Cluster Computing

Dynamic Load Balancing on Heterogeneous Clusters for Parallel Ant Colony Optimization

--Manuscript Draft--

Manuscript Number:	CLUS-D-15-00071R1
Article Type:	S.I. : CARLA 2014
Keywords:	Heterogeneous Computing; Ant Colony Optimization; CUDA; power-aware systems
Corresponding Author:	Manuel Ujaldón, Ph.D. Universidad de Malaga Malaga, Malaga SPAIN
First Author:	Antonio Llanes, M.Sc.
Order of Authors:	Antonio Llanes, M.Sc.
	José M. Cecilia, Ph.D.
	Antonia Sánchez, M.Sc.
	José M. García, Ph.D.
	Martyn Amos, Ph.D.
	Manuel Ujaldón, Ph.D.
Abstract:	Ant Colony Optimisation (ACO) is a nature-inspired, population-based metaheuristic that has been used to solve a wide variety of computationally hard problems. In order to take full advantage of the inherently stochastic and distributed nature of the method, we describe a parallelization strategy that leverages these features on heterogeneous and large-scale, massively-parallel hardware systems. Our approach balances workload effectively, by dynamically assigning jobs to heterogeneous resources which then run ACO implementations using different search strategies. Our experimental results confirm that we can obtain significant improvements in terms of both solution quality and energy expenditure, thus opening up new possibilities for the development of metaheuristic-based solutions to ``real world'' problems on high-performance, energy-efficient contemporary heterogeneous computing platforms.

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

Dynamic Load Balancing on Heterogeneous Clusters for Parallel Ant Colony Optimization

Antonio Llanes[1] · **José M. Cecilia**[1] · **Antonia Sánchez**[1] · **José M. García**[2] · **Martyn Amos**[3] · **Manuel Ujaldón**[4]

Abstract Ant Colony Optimisation (ACO) is a nature-inspired, population-based metaheuristic that has been used to solve a wide variety of computationally hard problems. In order to take full advantage of the inherently stochastic and distributed nature of the method, we describe a parallelization strategy that leverages these features on heterogeneous and large-scale, massively-parallel hardware systems. Our approach balances workload effectively, by dynamically assigning jobs to heterogeneous resources which then run ACO implementations using different search strategies. Our experimental results confirm that we can obtain significant improvements in terms of both solution quality and energy expenditure, thus opening up new possibilities for the development of metaheuristic-based solutions to "real world" problems on high-performance, energy-efficient contemporary heterogeneous computing platforms.

Keywords Heterogeneous Computing · Ant Colony Optimization · CUDA · power-aware systems

1 Introduction

Heterogeneous systems combine different types of processor, and computing nodes may use a combination of traditional multicore architectures (CPUs) and accelerators (mostly Nvidia GPUs [34] or Intel Xeon Phi cards [37]). Although such systems are becoming more common [4], they present a new set of specific challenges, such as scalability, energy efficiency, data management, programmability and reliability [6].

The role of the software developer will be increasingly important as such systems grow in popularity. They will be expected to manage the inherent tension between performance and power consumption, exploit the most useful feature of each component type, and be able to handle the complexity implied by combinations of hardware, instruction sets and programming models. So far, the efficient mapping of system components to computations within heterogeneous systems is largely the responsibility of the programmer (that is, the ability of the run-time system to achieve this is relatively immature).

The *hardware/software co-design* methodology has emerged since the 1990s as an approach to providing both *analysis* methods (which allow developers to assess whether or not a system meets its goals in terms of performance, power usage, etc.), and *synthesis* methods (which allow developers and researchers to rapidly explore the space of design methodologies) [12], [44].

This approach has facilitated significant advances in high-performance computing, which has, in turn, allowed for developments in computational modelling, image analysis, and many other areas [29], [40].

A particular application domain of interest to us is *metaheuristics*; specifically, algorithms inspired by *natural* processes or phenomena [39]. Many of these methods (such as the genetic algorithm [22], or particle swarm optimization [27]) are *population-based*: they maintain a *collection* of individual solutions which "evolves" in some way as the computation proceeds. These algorithms are generally stochastic, as they tend to rely on randomized search techniques. Additionally, they are inherently parallel, and many such variants have been described [5].

Affiliations:
[1] Department of Computer Science, Universidad Católica San Antonio de Murcia (UCAM). 30107 Murcia (Spain).
[2] Department of Computer Engineering, University of Murcia. 30080 Murcia (Spain).
[3] School of Computing, Mathematics and Digital Technology, Manchester Metropolitan University. Manchester (UK).
[4] Department of Computer Architecture, University of Málaga. 29071 Málaga (Spain).

One nature-based method of particular interest is *Ant Colony Optimization* (ACO) [15,16,20]. This algorithm is based on foraging behavior observed in colonies of ants, and has been applied to a wide variety of problems, including vehicle routing [45], feature selection [11] and autonomous robot navigation [21]. The method relies on "ants" (i.e., mobile agents) constructing paths on a graph representing a particular problem, where the paths represent a given solution. Paths are assessed according to the quality of the solution that they represent, and ants then deposit "pheromone" (i.e., signalling chemicals) accordingly (the better the solution, the higher the pheromone concentration). The algorithm takes advantage of positive feedback behaviour that emerges from the multi-agent system, where distributed selection quickly drives the population to high quality solutions.

The original ACO method (called the *Ant System* [17]) was developed by Dorigo in the 1990s, and this version (or slight variants thereof, such as the MAX-MIN Ant System (MMAS) [43]) is still in regular use [10,26, 28]. Parallel versions of the Ant System have been developed [13,31,41,46] (see also [35] for a survey), and, in recent work, we have presented a GPU-based version of ACO that, for the first time, parallelizes *both* main phases of the algorithm (that is, tour construction *and* pheromone deposition)[7,8].

The initial version of our ACO algorithm [7,8] was implemented in CUDA (Compute Unified Device Architecture) and written in C, which gave access to the parallel processing capabilities of the GPU. This paper extends our framework to encompass large-scale supercomputers, thus enabling its implementation in MPI and OpenMP (in addition to CUDA), and also incorporating different generations of Nvidia GPUs.

Since the advent of CUDA in 2006, at least four different generations of GPUs have been released: Tesla, Fermi, Kepler and Maxwell. Our algorithmic design investigates the potential to deploy a load-balancing strategy across several generations of Nvidia GPUs, for maximum performance and minimum power consumption. In what follows, we use our well-established ACO based metaheuristic as a both a benchmarking application and an illustration of the long-term potential for this method. Our experimental study covers a wide range of computing systems, from consumer-market devices to high-end servers.

This paper is organized as follows. Section 2 reviews the ACO method, the CUDA programming model and our ACO-based algorithm. Section 3 describes our parallelization techniques to enhance ACO simulation on GPU-based heterogeneous clusters, which form the main contribution of this work. Section 4 focuses on the experimental results, Section 5 gives a performance analysis, and we conclude in Section 6 with an overall assessment and suggestions for future work.

2 Background

2.1 Ant Colony Optimisation for the Traveling Salesman Problem

In what follows, we reprise our description of the algorithm, which was first given in [9]. The Traveling Salesman Problem (TSP)[30] involves finding the shortest (or "cheapest") round-trip route that visits each of a number of "cities" exactly once. The symmetric TSP on n cities may be represented as a complete weighted graph, G, with n nodes, with each weighted edge, $e_{i,j}$, representing the inter-city distance $d_{i,j} = d_{j,i}$ between cities i and j. The TSP is a well-known NP-hard optimisation problem, and is used as a standard benchmark for many heuristic algorithms [25].

The TSP was the first problem solved by Ant Colony Optimisation (ACO) [18,14]. This method uses a number of simulated "ants" (or *agents*), which perform distributed search on a graph. Each ant moves through on the graph until it completes a tour, and then offers this tour as its suggested solution. In order to do this, each ant may drop "pheromone" on the edges contained in its proposed solution. The amount of pheromone dropped, if any, is determined by the *quality* of the ant's solution relative to those obtained by the other ants. The ants probabilistically choose the next city to visit, based on *heuristic information* obtained from inter-city distances and the net pheromone trail. Although such heuristic information drives the ants towards an optimal solution, a process of "evaporation" is also applied in order to prevent the process stalling in a local minimum.

The Ant System (AS) is an early variant of ACO, first proposed by Dorigo [14]. The AS algorithm is divided into two main stages: *Tour construction* and *Pheromone update*. Tour construction is based on m ants building tours in parallel. Initially, ants are randomly placed. At each construction step, each ant applies a probabilistic action choice rule, called the *random proportional rule*, in order to decide which city to visit next. The probability for ant k, placed at city i, of visiting city j is given by the equation 1

$$p_{i,j}^{k} = \frac{[\tau_{i,j}]^{\alpha} [\eta_{i,j}]^{\beta}}{\sum_{l \in N_i^k} [\tau_{i,l}]^{\alpha} [\eta_{i,l}]^{\beta}}, \qquad if\ j \in N_i^k, \tag{1}$$

where $\eta_{i,j} = 1/d_{i,j}$ is a heuristic value that is available *a priori*, α and β are two parameters which deter-

mine the relative *influences* of the pheromone trail and the heuristic information respectively, and N_i^k is the feasible neighbourhood of ant k when at city i. This latter set represents the set of cities that ant k has not yet visited; the probability of choosing a city outside N_i^k is zero (this prevents an ant returning to a city, which is not allowed in the TSP). By this probabilistic rule, the probability of choosing a particular edge (i, j) increases with the value of the associated pheromone trail $\tau_{i,j}$ and of the heuristic information value $\eta_{i,j}$. The numerator of the equation 1 is pretty much the same for every ant in a single run, thus, computation times can be saved by storing this information in additional matrix, called *choice_info matrix* as showed in [19]. The random propotional rule ends with a selection procedure, which is done analogously to the *roulette wheel* selection procedure of evolutionary computation (for more detail see [19], [23]). Each value $choice\_info[current\_city][j]$ of a city j that ant k has not visited yet determines a slice on a circular roulette wheel, the size of the slice being proportional to the weight of the associated choice. Next, the wheel is spun and the city to which the marker points is chosen as the next city for ant k. Furthermore, each ant k maintains a memory, M^k, called the *tabu list*, which contains the cities already visited, in the order they were visited. This memory is used to define the feasible neighbourhood, and also allows an ant to both to compute the length of the tour T^k it generated, and to retrace the path to deposit pheromone.

After all ants have constructed their tours, the pheromone trails are updated. This is achieved by first lowering the pheromone value on all edges by a constant factor, and then adding pheromone on edges that ants have crossed in their tours. Pheromone evaporation is implemented by

$$\tau_{i,j} \leftarrow (1-\rho)\tau_{i,j}, \qquad \forall (i,j) \in L, \tag{2}$$

where $0 < \rho \leq 1$ is the pheromone evaporation rate. After evaporation, all ants deposit pheromone on their visited edges:

$$\tau_{i,j} \leftarrow \tau_{i,j} + \sum_{k=1}^{m} \Delta\tau_{i,j}^k, \qquad \forall (i,j) \in L, \tag{3}$$

where $\Delta\tau_{ij}$ is the amount of pheromone ant k deposits. This is defined as follows:

$$\Delta\tau_{i,j}^k = \begin{cases} 1/C^k \text{ if } e(i,j)^k & \text{belongs to } T^k \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

where C^k, the length of the tour T^k built by the k-th ant, is computed as the sum of the lengths of the edges belonging to T^k . According to equation 4, the better an ant's tour, the more pheromone the edges belonging to this tour receive. In general, edges that are used by many ants (and which are part of short tours), receive more pheromone, and are therefore more likely to be chosen by ants in future iterations of the algorithm.

2.2 The CUDA programming model

Compute Unified Device Architecture (CUDA) [33] is a platform for Graphics Processing Units (GPUs), covering both hardware and software. On the hardware side, the GPU consists of N multiprocessors which are replicated within the silicon area, each endowed with M cores sharing the control unit, and a shared memory (a small cache explicitly managed by the programmer). Each GPU generation has increased CUDA Compute Capabilities (CCC), as well as increasing the number of cores and shared memory size (see Table 1). In conjunction with these developments, power consumption has been reduced by a factor of 2 at each new generation.

The CUDA software paradigm is based on a hierarchy of abstraction layers: the *thread* is the basic execution unit; threads are grouped into *blocks*, and blocks are mapped to multiprocessors. C language procedures to be ported to GPUs are transformed into CUDA *kernels*, mapped to many-cores in a SIMD (Single Instruction Multiple Data) fashion (that is, with all threads running the same code but having different IDs). The programmer deploys parallelism by declaring a *grid* composed of blocks equally distributed among all multiprocessors. A kernel is therefore executed by a grid of thread blocks, where threads run simultaneously grouped in batches called *warps*, which are the main scheduling units.

2.3 Our initial CUDA implementation

In previous work, we developed a CUDA-based ACO implementation, with an emphasis on *data parallelism* [7]. We now summarize this algorithm, as it provides the foundation of the current work.

Recall that our ACO implementation involves ants moving on a graph, deciding where to move next based on simulated pheromone concentrations. When an ant makes a decision on which city/node to visit next, it must calculate heuristic values which are the same for all ants at any one time step (that is, the heuristic information constitutes information on nodes, which must be consistent and accessible to all ants). It makes sense, therefore, to split the computation of heuristic values

Table 1 CUDA summary by hardware generation since its inception (four generations up to 2015).

Hardware generation and starting year	Tesla 2007	Fermi 2010	Kepler 2012	Maxwell 2014
Multiprocessors per die (up to)	30	16	15	16
Cores per multiprocessor	8	32	192	128
Total number of cores (up to)	240	512	2880	2048
Shared memory size (maximum in Kbytes, per multiprocessor)	16	48	48	96
CUDA Compute Capabilities (CCC)	1.3	2.1	3.5	5.2
Peak single-precision performance (GFLOPS)	672	1178	4290	4980
Performance per watt (approximated and normalized)	1	2	6	12

into a separate *heuristic info kernel*, which is then executed prior to tour construction. Transition probabilities are stored in a two-dimensional *choice matrix*, which is used to inform "roulette wheel" (Monte Carlo) selection by each ant.

In the *tour construction* kernel, each ant is associated with a *thread block*, such that each thread represents a city (or cities) that the ant may visit. This avoids the problem of warp divergences, and enhances data parallelism, as all threads within a block may *cooperate*. The degree of parallelism improves by a factor of $1 : w$, where w is the number of CUDA threads per block.

Finally, the *pheromone kernel* performs evaporation and deposition. Evaporation is straightforward, as a single thread can independently lower each entry in the pheromone matrix by a constant factor. Deposition is more challenging, since each ant generates its own private tour in parallel, and will eventually visit the same edge as another ant. In order to prevent race conditions, we require the use of CUDA atomic operations when accessing the pheromone matrix in this stage.

3 Scaling to heterogeneous clusters

Traditional parallel implementations are not always efficient when ported to heterogeneus systems. They are often inherited from scalable supercomputers, where all nodes in the cluster have the same compute capabilities, and they therefore lack the ability to distinguish computational devices with assymmetric computational power and energy consumption. Differences are not limited to fundamental hardware design (CPUs vs. GPUs), but also occur within the same family of processors. For example, the Kepler family (see Table 1) includes Tesla K20, K20X and K40 models, endowed with 13, 14 and 15 multiprocessors, respectively (the K80 model even reaches 30 multiprocessors split into two chips). Figure 1 shows a heterogeneous cluster which, nowadays, may include different Nvidia GPU generations, even within the same node.

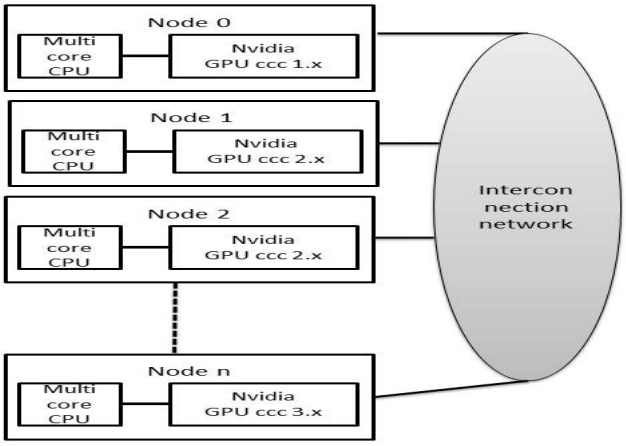


Fig. 1 Heterogeneous system based on different Nvidia GPU generations.

With this scenario in mind, we introduce a heterogeneity-aware parallelization of Ant Colony Optimisation applied to the Travelling Salesman Problem as introduced in Section 2.1. Our departure point is (1) the CUDA-based implementation of ACO described in Section 2.3, and (2) the parallelization strategy proposed by Stützle [42], where independent instances of the ACO algorithm are run on different processors (GPUs in our case, having assorted CUDA Compute Capabilities).

Parallel runs do not incur any communication overhead, and the final solution is chosen across all independent executions, taking advantage of the stochastic nature of ACO algorithms. The execution time of each independent execution may differ, as it depends on (1) the underlying GPU each ACO instance runs on, which is actually unknown at compile-time, and (2) the TSP instance size (the same in principle for all processors, but affected by GPU heterogeneity). Given that the slowest GPU will determine the overall execution time, our mission is to make use of the idle time offered by the most powerful GPUs. Performance and energy differences shown in the last two rows of Table 1 lead us to believe that there is ample room for improvement here.

We have designed an implementation with three main focuses: (1) Resources accounting through MPI processes, (2) performance monitoring via OpenMP threads and, (3) power consumption balance using GPU Boost. We now expand on each of these in the following subsections.

3.1 Resources accounting

First, our algorithm defines a MPI thread for each existing node in the cluster where we run our simulation. Heuristic information about inter-city distances is sent to each node, where supporting data structures are also created to avoid communication overhead. Then each MPI thread creates as many OpenMP threads as GPUs are available on a node, which is easily attained by querying the GPU properties at runtime (using `cudaGetDeviceCount` from the CUDA API) and NVML (Nvidia Management Library).

3.2 Performance monitoring

Secondly, a *warm-up* phase is performed to establish performance differences among all targeted GPUs running the particular TSP instance to be solved. This phase measures, at run-time, the execution time of a small number of iterations of the ACO algorithm (five to ten) in order to detect these differences. Importantly, at this stage, the algorithm is not trying to *solve* the TSP problem in any meaningful sense (five to ten iterations is not enough to do so) but these runs allow us to calculate the performance differences between GPUs. The execution times spent at this *warm-up* phase on all GPUs are reduced to obtain the maximum value using `MPI_Allreduce`. Thus, the *Percent* parameter is eventually determined according to equation 5. The slowest GPU will have $Percent = 1$, a GPU two times faster than slowest GPU would have $Percent = 0.5$, and so on.

$$Percent = \frac{Ex.time_{actualGPU}}{Ex.time_{slowestGPU}}/ \tag{5}$$

We then establish the *time-budget*, which is a threshold that determines the maximum completion time for that ACO algorithm on every GPU. It corresponds to the execution time required to perform a number of iterations of ACO on the slowest GPU available. This number of iterations (referred to as δ from now on) is a configuration parameter of our algorithm, and is known by all nodes in the simulation. It is empirically determined to be good enough to find out a good solution to the TSP on our CUDA implementation of ACO. For instance, in our experimental section δ is set to 1000 iterations.

Each OpenMP thread then calculates the slot that it can use for the simulation (γ, with $\gamma > \delta$). This slot can be used for a deeper search (thus computing additional iterations of ACO), or for reducing the power consumption (by relaxing the clock rate in GPU cores). In addition, when $\gamma \geq \delta/2$, the algorithm can even do a restart to avoid becoming "trapped" in a local minimum.

Additional iterations (γ) are obtained by equation 6.

$$\gamma = \delta * (1/percent); \tag{6}$$

where "percent" is the performance difference identified among GPUs at warm-up stage, which we have previously explained.

The number of restarts or additional iterations that each GPU may perform is calculated by equation 7

$$\gamma = 1/percent; \tag{7}$$

as the numerator represents the percent for the slowest GPU, which is always set to 1.

Finally, if we wish to reduce the overall *power consumption* of our simulation, we may use GPU Boost^{TM}, which is a new hardware feature introduced by Nvidia from the K40 Kepler GPU onwards. GPU Boost manipulates the clock rate of the GPU cores to trade performance by energy. The idea is to sacrifice time in favour of power consumption when the latter is more critical. Developers can use the `nvidia-smi` shell command to set up the frequency in the GPU, usually exceeding/reducing the nominal value around 20%. To prevent excessive thermal stress, Nvidia does not allow developers to change this parameter at run-time or within an application, as the Intel SpeedStep^{TM}does. Moreover, the GPU is required to work in *Persistence Mode*, which ensures that driver stays loaded even when the GPU has no work to run on it. The range of clocks supported can be queried by the `nvidia-smi -d SUPPORTED_CLOCKS` command, and changed with the `-ac` option (see [1] for more details and a full list of commands). Clock changes require superuser privileges, or developers can use the NVIDIA Management Library (NVML) [3] instead. NVML is a C-based API for monitoring and managing diverse states of NVIDIA GPU devices (including clock settings), without requiring the user to run `nvidia-smi` prior to launching the application on the GPU. The real-time power consumption measurement of individual GPU components using a software

Table 2 Hardware resources and experimental setup used during our executions.

	Vendor and type	Intel CPU	Nvidia GPUs			
	Family	Haswell	Fermi	Kepler	Kepler	Maxwell
	Class	Xeon	Tesla	Tesla	Tesla	GeForce
	Model	X7550	C2050	K20c	K40c	GTX 980
	Year	2015	2012	2013	2014	2015
Processing elements	Cores per multiprocessor	(does not	32	192	192	128
	Number of multiprocessors	apply)	14	13	15	16
	Total number of cores	8	448	2496	2880	2048
	Clock frequency (MHz)	2000	1147	706	745	1216
Maximum number of GPU threads	Per multiprocessor	(does	1536	2048	2048	2048
	Per block	not	1024	1024	1024	1024
	Per warp	apply)	32	32	32	32
Register file	32-bit registers (per multiprocessor)		32768	65536	65536	65536
SRAM memory (per multiproc. on GPUs)	Shared (only GPUs)	(32 KB L1D	16 or 48 KB	16 or 48 KB	16 or 48 KB	96 KB
	L1 cache	and	48 or 16 KB	48 or 16 KB	48 or 16 KB	(48 KB
	(Shared + L1)	32 KB L1I)	64 KB	64 KB	64 KB	per block)
L2 cache	(shared by	256 KB	768 KB	1280 KB	1536 KB	2048 KB
L3 cache	all cores)	16 MB	(does not apply)			
DRAM memory	Size (Megabytes)	131072	2687	4800	11520	4096
	Speed (MHz)	2x666	2x1546	2x2600	2x3004	2x3505
	Width (bits)	256	384	320	384	256
	Bandwidth (Gbytes/s)	42.66	148.41	208	288.38	224.32
	Technology	DDR3	GDDR5	GDDR5	GDDR5	GDDR5
CUDA Compute Capabilities		(d.n.a.)	2.0	3.5	3.5	5.2

approach is only supported by the Nvidia Kepler architecture GPU. This is also done by using NVML, which reports the GPU power usage at real-time. We use `nvmlDeviceGetPowerUsage` command to obtain power usage.

4 Experimental setup

4.1 Hardware environment

For this experimental study, we used the following platforms:

- **On the CPU side:** Four Intel Xeon X7550 processors running at 2 GHz and plugged into a quad-channel motherboard endowed with 128 Gigabytes of DDR3 memory.
- **On the GPU side:** Four GPUs, starting with an Tesla C2050 (Fermi generation, approximately 4 years old) and ending with a brand new GeForce GTX 980 (Maxwell generation), with two Kepler models in between (K20 and K40), all sharing the motherboard space with PCI-e 3.0 slots to communicate with the CPUs.

Table 2 gives a detailed descriptions of all these platforms. We use gcc 4.8.2 with the -O3 flag to compile on the CPU, and the CUDA compiler/driver/runtime version 6.5 to compile and run on the GPU.

Table 3 Description of benchmark instances from TSPLIB library (EUC_2D stands for 2D euclidean distance).

Name	Cities	Type	Best tour length
d198	198	EUC_2D	15780
a280	280	EUC_2D	2579
lin318	318	EUC_2D	42029
pcb442	442	EUC_2D	50778
rat783	783	EUC_2D	8806
pr1002	1002	EUC_2D	259045

4.2 Benchmarking

We test our designs using a set of benchmark instances from the well-known TSPLIB library [38] [2]. All benchmark instances are defined on a complete graph, and all distances are defined as integer numbers. Table 3 shows a list of all targeted benchmark instances with information on the number of cities, the type of distance and the length of optimal tours.

ACO parameters such as the number of ants (m), and those values to set up their behaviour, like α, β, ρ, and so on, are set according to the values recommended in [19]. In particular, $m = n$ (being n the number of cities), $\alpha = 1$, $\beta = 2$ and $\rho = 0.5$.

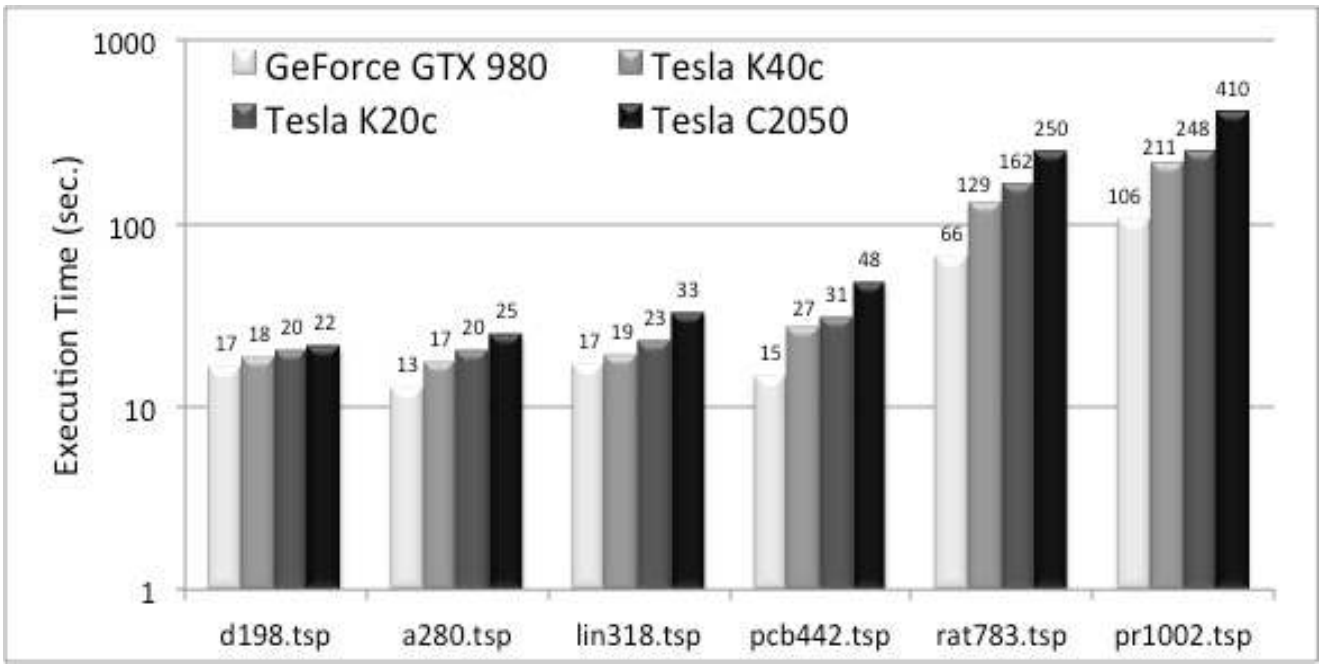


Fig. 2 Execution times in seconds on different Nvidia GPU generations for several TSP instances. Although we have used a Tesla s2050 in our experiments, the figure only shows the performance of a single GPU of the S2050 server (i.e. Tesla C2050).

5 Experimental results

Given the fact that our techniques establish the experimental setup dynamically, results shown below are platform dependent.

5.1 Performance and workload balance

Figure 2 shows performance differences across different GPU generations when they run several TSP instances. Results are recorded for 1000 iterations, and averaged over 10 different runs. The fastest GPU belongs to the latest generation (Maxwell-based GeForce GTX 980), outperforming the slowest GPU by up to a 4.2x factor. This slowest GPU is the Tesla C2050, which determines the *time-budget* for the entire execution. Tesla K20c, the Kepler model, obtains intermediate results, with up to 1.6x gain versus the Tesla C2050.

Results are measured statically for the sake of showing performance differences in a real scenario. However, as described, our methodology includes a *warm-up stage* to calculate these differences at run-time. In previous work [7], more details about performance analysis are given; in particular, we reported up to 20x speed-up factor on average for a Tesla C2050 versus a single-threaded CPU.

We now enhance our parallelization strategy to take advantage of the time that Kepler and Maxwell GPUs are idle, in order to improve the quality of the results. One idea, which we call **Deep Search**, is to increase the *number of iterations* in order to perform a deeper search within the same time budget. For instance, GeForce GTX 980 carries out 4102 iterations, Tesla K40 carries out 1946 iterations, Tesla K20c carries out 1654 iterations, and Tesla C2050 just 1000 iterations (the time-budget established for this simulation).

Another possibility is to include a restart to avoid being trapped in a local minimum. That is possible if and only if the performance gap is at least twice the slowest GPU performance. These two goals can be merged to create a hybrid approach which we call **Deep Search + Restart**. Driven by this combination, GeForce GTX 980 may perform up to four restarts of 1000 iterations each (as its percent value is 0.24 on pr1002 TSP instance), whereas Tesla K40 and Tesla K20c only perform a single phase with a deeper search involving 1946 and 1657 iterations, respectively (0.51 and 0.60 percent values are not enough to complete two restarts).

Figure 3 shows a tour quality comparison across the sequential run and all parallel strategies for a variety of benchmarks normalized by the optimal solution. The first bar represents the sequential code, written in ANSI C, provided by Stuzle in [19]. This code runs for 1000 ACO iterations on a single-threaded CPU. The second bar is the result quality for our GPU version over 1000 ACO iterations. Figures show that the quality of solutions obtained for these two versions are relatively similar to each other.

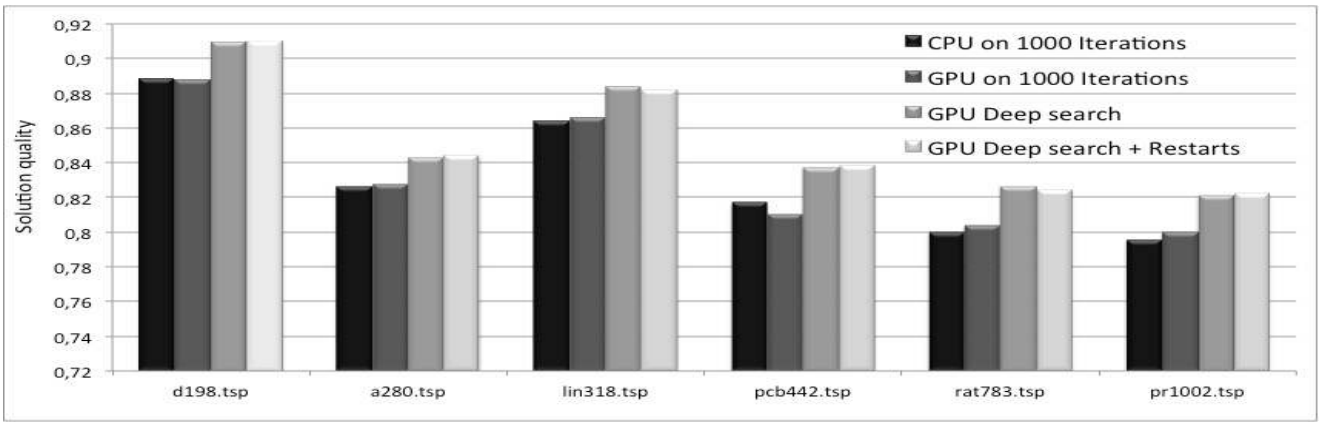


Fig. 3 Quality of the results obtained for different TSP Lib instances, normalized to the optimal solution.

The third bar shows our GPU Deep Search strategy, and the fourth bar represents Deep Search + Restart. These two last versions improve results by significant margin within the same time-budget, with a small advantage for Deep Search on average. Note that Deep Search performs restarts implicitly, as different searches are executed on different GPUs, whereas Deep Search + Restarts includes restarts explicitly on the same GPU.

5.2 Power consumption

Figure 4 shows the power budget for our simulation under different clock settings. Performance gains reflect up to 1.3x speed-up factor, in line with the 31% increment in the clock rate (frequency raises from 666 MHz to 875 MHz).

Figure 5 outlines power consumption in milliwatts for different clock rates. As expected, power consumption raises with higher clock frequencies.

The overall power budget is correlated to the total execution time of the application (see Figure 6.a). However, the 745 MHz clock setting - which is actually set by default on Nvidia's driver for the Tesla K40 - is the most energy efficient.

5.3 Power-aware performance metrics

Researchers have proposed metrics combining performance and power measures into a single index. The most popular in low-power circuit design is in the form of ED^n [36], where E is the energy, D is the circuit delay, and n is a nonnegative integer. The power-delay product (PDP), the energy-delay product (EDP) [24] and the energy-delay-squared product (ED^2P) [32] are all special cases of ED^n with $n = 0, 1, 2$, respectively.

Intuitively, ED^n captures the energy usage per operation, with a lower value reflecting the fact that power is more efficiently translated into the speed of operation. The parameter n implies that a 1% reduction in circuit delay is worth paying an n% increase in energy usage; thus, different n values represent varying degrees of emphasis on deliverable performance over power consumption.

Figure 6.b shows the Energy Delay Product (EDP) for our ACO simulation, and Figure 6.c the Energy Delay Square Product (triple weight on performance). These couple of metrics prioritize performance over energy. Figure 4 shows that performance differences among different clock frequencies are remarkable, to benefit fastest settings.

6 Conclusions and future work

We present a parallelization strategy tailored to heterogeneous and massively parallel systems. Heterogeneity may limit acceleration and waste energy unless programmers develop smarter applications to wisely control those features on the road towards an optimal performance/watt ratio. Our proposal cares about accuracy, joules and time equally, deploying those magnitudes on an equilateral triangle managed by a cooperative scheduling of jobs to attain an optimal balance among them at run-time. This makes our strategy particularly useful for non-deterministic algorithms and stochastics behaviours where real-time and/or energy contraints must be fulfilled. With the user setting up those constraints properly, our method may even grant priority to any of the goals composing the metaheuristic.

In a preliminary stage of development, we have illustrated our ideas using Ant Colony Optimization as case study. Given the scalability demonstrated along

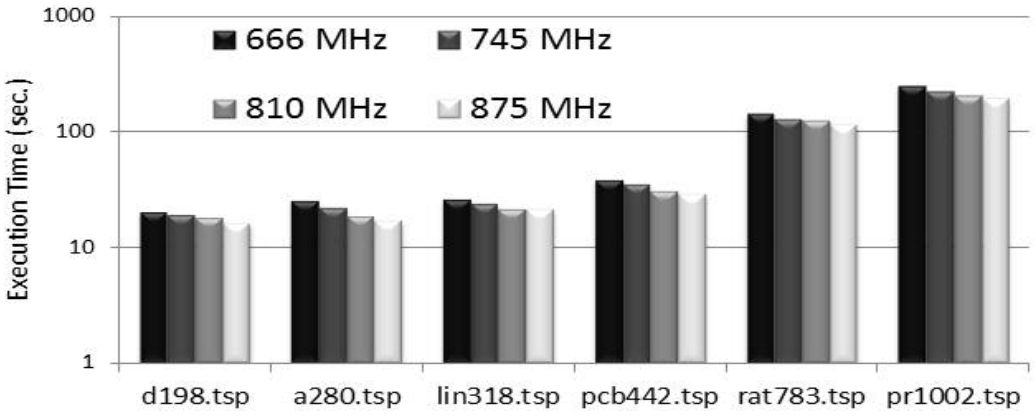


Fig. 4 Execution times in seconds on a Tesla K40 GPU for several TSP instances using different clock frequencies.

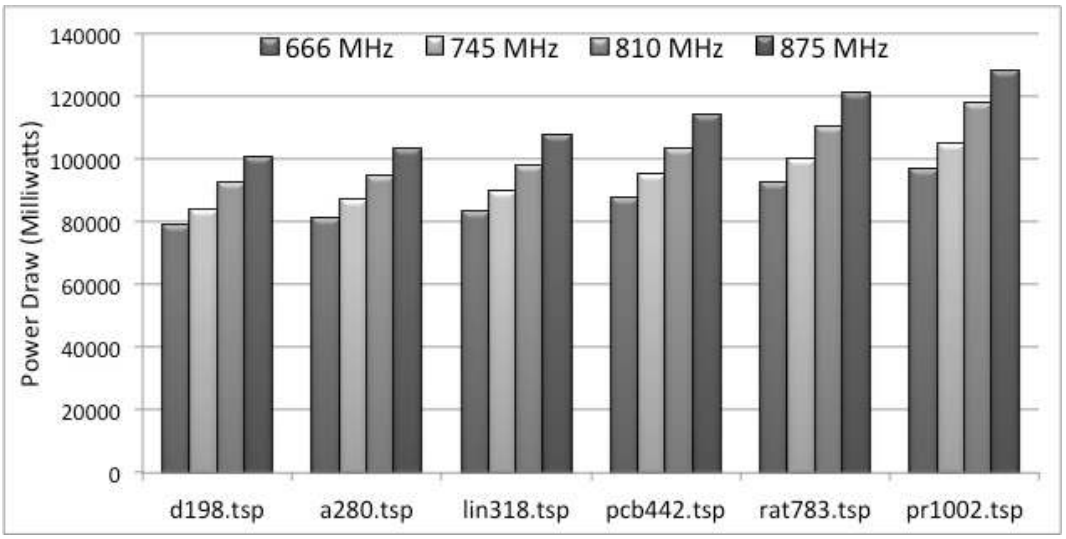
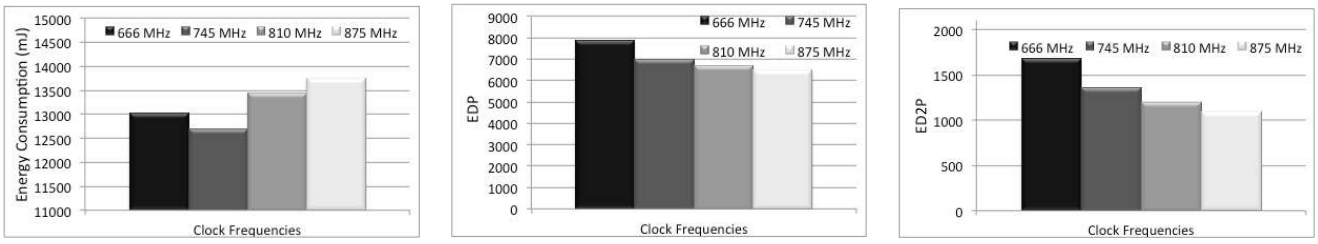


Fig. 5 Power consumption (in milliwatts) measured for the Tesla K40 GPU on different clock frequencies and TSP instances.



(a) Total energy. (b) Energy Delay Product (EDP). (c) Energy Delay Square Product.

Fig. 6 Energy consumption in *Joules*/1000 (mJ) measured on different clock frequencies for the Tesla K40 GPU. Measurements are taken for the execution on all targeted TSP instances, and averaged over 10 launches.

our experimental study, we foresee an immense potential to extend and refine our methods in future heterogeneous systems. In particular, queries to measure energies and temperatures within the GPU are weak and almost non-existing on low-power devices like Tegra heterogeneous plaforms. Given the long way ahead for improvement and how vendors are enthusiastically endorsing low-power devices, we believe the ideas presented here will greatly benefit from incoming sensors, hardware counters, middleware, libraries and tools, to provide the research community solid pillars to face the expected growth of heterogeneous systems in a much better power-aware manner.

Acknowledgments

This work is jointly supported by the Fundación Séneca (Agencia Regional de Ciencia y Tecnología, Región de Murcia) under grants 15290/PI/2010 and 18946/JLI/13, by the Spanish MEC under grants TIN2012-31345 and TIN2013-42253-P, by the Nils Coordinated Mobility under grant 012-ABEL-CM-2014A, in part financed by the European Regional Development Fund (ERDF), and by the Junta de Andalucía under Project of Excellence P12-TIC-1741. We also thank Nvidia for hardware donations within UCAM and UMA CUDA Teaching and Research Centers awards.

References

1. Parallel forall blog. Nvidia CUDA Zone. http://devblogs.nvidia.com/parallelforall/increase-performance-gpu-boost-k80-autoboost/ [11 March 2015]
2. TSPLIB Webpage (2011). http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/
3. Nvidia Corporation. NVML API Reference ([last accesed 15 November 2014]). http://developer.download.Nvidia.com/assets/cuda/files/CUDADownloads/NVML/nvml.pdf
4. Top 500 supercomputer site ([last accesed 15 November 2014]). http://www.top500.org/
5. Alba, E., Luque, G., Nesmachnow, S.: Parallel metaheuristics: recent advances and new trends. International Transactions in Operational Research **20**(1), 1–48 (2013). DOI 10.1111/j.1475-3995.2012.00862.x
6. Carretero, J., Garcia-Blas, J., Singh, D.E., Isaila, F., Fahringer, T., Prodan, R., Bosilca, G., Lastovetsky, A., Symeonidou, C., Perez-Sanchez, H., et al.: Optimizations to enhance sustainability of mpi applications. In: Proceedings of the 21st European MPI Users' Group Meeting, p. 145. ACM (2014)
7. Cecilia, J.M., Garcia, J.M., Nisbet, A., Amos, M., Ujaldón, M.: Enhancing data parallelism for ant colony optimization on GPUs. Journal of Parallel and Distributed Computing **73**(1), 42–51 (2013)
8. Cecilia, J.M., Garcia, J.M., Ujaldon, M., Nisbet, A., Amos, M.: Parallelization strategies for ant colony optimisation on GPUs. In: Proceedings of the 2011 IEEE International Symposium on Parallel and Distributed Processing, pp. 339–346. IEEE (2011)
9. Cecilia, J.M., Nisbet, A., Amos, M., Garcia, J.M., Ujaldón, M.: Enhancing GPU parallelism in nature-inspired algorithms. Journal of Supercomputing **63**(3), 773–789 (2013)
10. Chang, R.S..S., Chang, J.S..S., Lin, P.S..S.: An ant algorithm for balanced job scheduling in grids. Future Generation Computer Systems **25**(1), 20–27 (2009). DOI 10.1016/j.future.2008.06.004
11. Chen, Y., Miao, D., Wang, R.: A rough set approach to feature selection based on ant colony optimization. Pattern Recognition Letters **31**(3), 226–233 (2010). DOI 10.1016/j.patrec.2009.10.013
12. De Michell, G., Gupta, R.K.: Hardware/software co-design. Proceedings of the IEEE **85**(3), 349–365 (1997)
13. Delévacq, A., Delisle, P., Gravel, M., Krajecki, M.: Parallel ant colony optimization on graphics processing units. Journal of Parallel and Distributed Computing **73**(1), 52–61 (2013). DOI 10.1016/j.jpdc.2012.01.003
14. Dorigo, M.: Optimization, learning and natural algorithms. Ph.D. thesis, Politecnico di Milano, Italy (1992)
15. Dorigo, M., Birattari, M., Stutzle, T.: Ant colony optimization. Computational Intelligence Magazine, IEEE **1**(4), 28–39 (2006)
16. Dorigo, M., Di Caro, G.: Ant colony optimization: A new meta-heuristic. In: Proceedings of the 1999 Congress on Evolutionary Computation (CEC'99), pp. 1470–1477. IEEE Press (1999)
17. Dorigo, M., Maniezzo, V., Colorni, A.: Ant system: Optimization by a colony of cooperating agents. IEEE Transactions on Systems, Man and Cybernetics B **26**(1), 29–41 (1996)
18. Dorigo, M., Maniezzo, V., Colorni, A.: The ant system: Optimization by a colony of cooperating agents. IEEE Transactions on Systems, Man, and Cybernetics-Part B **26**, 29–41 (1996)
19. Dorigo, M., Stutzle, T.: Ant Colony Optimization. Bradford Company (2004)
20. Dorigo, M., Stützle, T.: Ant colony optimization: overview and recent advances. In: Handbook of metaheuristics, pp. 227–263. Springer (2010)
21. Garcia, M.P., Montiel, O., Castillo, O., Sepúlveda, R., Melin, P.: Path planning for autonomous mobile robot navigation with ant colony optimization and fuzzy cost function evaluation. Applied Soft Computing **9**(3), 1102–1110 (2009). DOI 10.1016/j.asoc.2009.02.014
22. Goldberg, D.E.: Genetic algorithms in search, optimization, and machine learning. Addison-Wesley Professional (1989)
23. Goldberg, D.E.: Genetic Algorithms in Search, Optimization and Machine Learning, 1st edn. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1989)
24. González, R., Horowitz, M.: Energy dissipation in general purpose microprocessors. IEEE Journal of Solid-State Circuits **31**(9) (1996)
25. Johnson, David S., Mcgeoch, Lyle A.: The Traveling Salesman Problem: A Case Study in Local Optimization (1997)
26. Ke, B.R., Chen, M.C., Lin, C.L.: Block-layout design using max-min ant system for saving energy on mass rapid transit systems. IEEE Transactions on Intelligent Transportation Systems **10**(2), 226–235 (2009). DOI 10.1109/TITS.2009.2018324
27. Kennedy, J., Eberhart, R.: Particle swarm optimization. In: Neural Networks, 1995. Proceedings., IEEE International Conference on, vol. 4, pp. 1942–1948. IEEE (1995)
28. Komarudin, Wong, K.Y.: Applying ant system for solving unequal area facility layout problems. European Journal of Operational Research **202**(3), 730–746 (2010). DOI 10.1016/j.ejor.2009.06.016
29. Krueger, J., Donofrio, D., Shalf, J., Mohiyuddin, M., Williams, S., Oliker, L., Pfreund, F.J.: Hardware/software co-design for energy-efficient seismic modeling. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, p. 73. ACM (2011)
30. Lawler, E., Lenstra, J., Kan, A., Shmoys, D.: The traveling salesman problem. Wiley New York (1987)
31. Manfrin, M., Birattari, M., Stützle, T., Dorigo, M.: Parallel ant colony optimization for the traveling salesman problem. In: Ant Colony Optimization and Swarm Intelligence, pp. 224–234. Springer (2006)

32. Martin, A.: Towards an energy complexity of computations. Information Processing Letters **77**, 181–187 (2001)
33. Nickolls, J., Buck, I., Garland, M., Skadron, K.: Scalable parallel programming with cuda. Queue **6**(2), 40–53 (2008)
34. NVIDIA: NVIDIA CUDA C Programming Guide 6.5 (2014)
35. Pedemonte, M., Nesmachnow, S., Cancela, H.: A survey on parallel ant colony optimization. Applied Soft Computing **11**(8), 5181–5197 (2011). DOI 10.1016/j.asoc.2011.05.042
36. Pénzes, P., Martin, A.: Energy-delay efficiency of vlsi computations. In: Proceedings of the ACM Great Lakes Symposium on VLSI (GLSVLSI). IEEE (2002)
37. Rahman, R.: Xeon phi system software. In: Intel® Xeon Phi Coprocessor Architecture and Tools, pp. 97–112. Springer (2013)
38. Reinelt, G.: TSPLIB— a traveling salesman problem library. ORSA Journal on Computing **3**(4), 376–384 (1991)
39. Rozenberg, G., Bäck, T., Kok, J.N.: Handbook of Natural Computing. Springer (2011)
40. Shalf, J., Quinlan, D., Janssen, C.: Rethinking hardware-software codesign for exascale systems. Computer **44**(11), 22–30 (2011)
41. Stützle, T.: Parallelization strategies for ant colony optimization. In: Parallel Problem Solving from Nature (PPSN V), pp. 722–731. Springer (1998)
42. Stützle, T.: Parallelization strategies for ant colony optimization. In: PPSN V: Proceedings of the 5th International Conference on Parallel Problem Solving from Nature, pp. 722–731. Springer-Verlag, London, UK (1998)
43. Stutzle, T., Hoos, H.H.: MAX-MIN ant system. Future Generation Computer Systems **16**(8), 889–914 (2000)
44. Wolf, W.: A decade of hardware/software codesign. Computer **36**(4), 38–43 (2003)
45. Yu, B., Yang, Z.Z., Yao, B.: An improved ant colony optimization for vehicle routing problem. European Journal of Operational Research **196**(1), 171–176 (2009). DOI 10.1016/j.ejor.2008.02.028
46. Zhu, W., Curry, J.: Parallel ant colony for nonlinear function optimization with graphics hardware acceleration. In: Systems, Man and Cybernetics, 2009. SMC 2009. IEEE International Conference on, pp. 1803–1808. IEEE (2009)

Figure 1
Click here to download Figure: Fig1.jpg

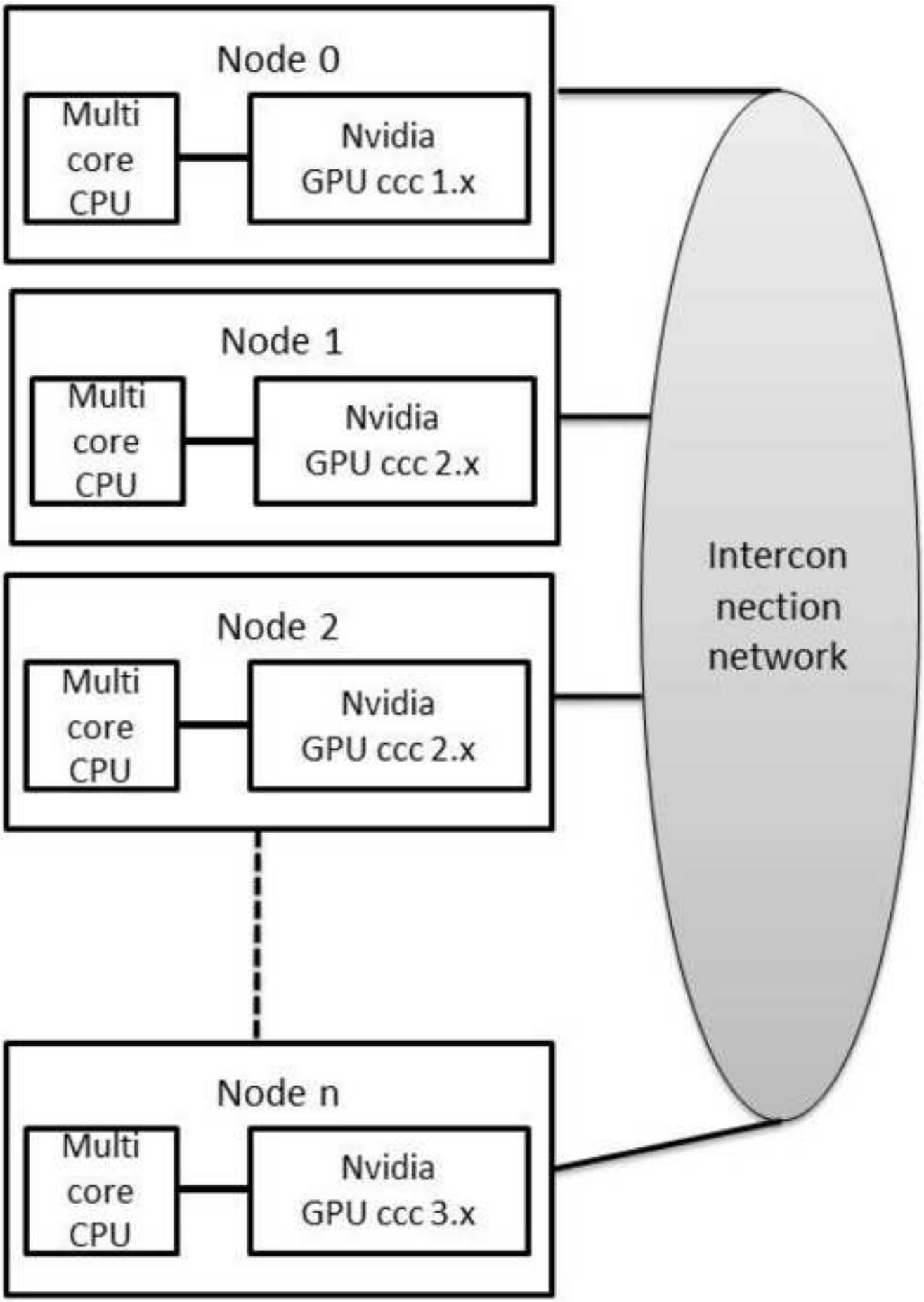


Figure 2
Click here to download Figure: Fig2.jpg

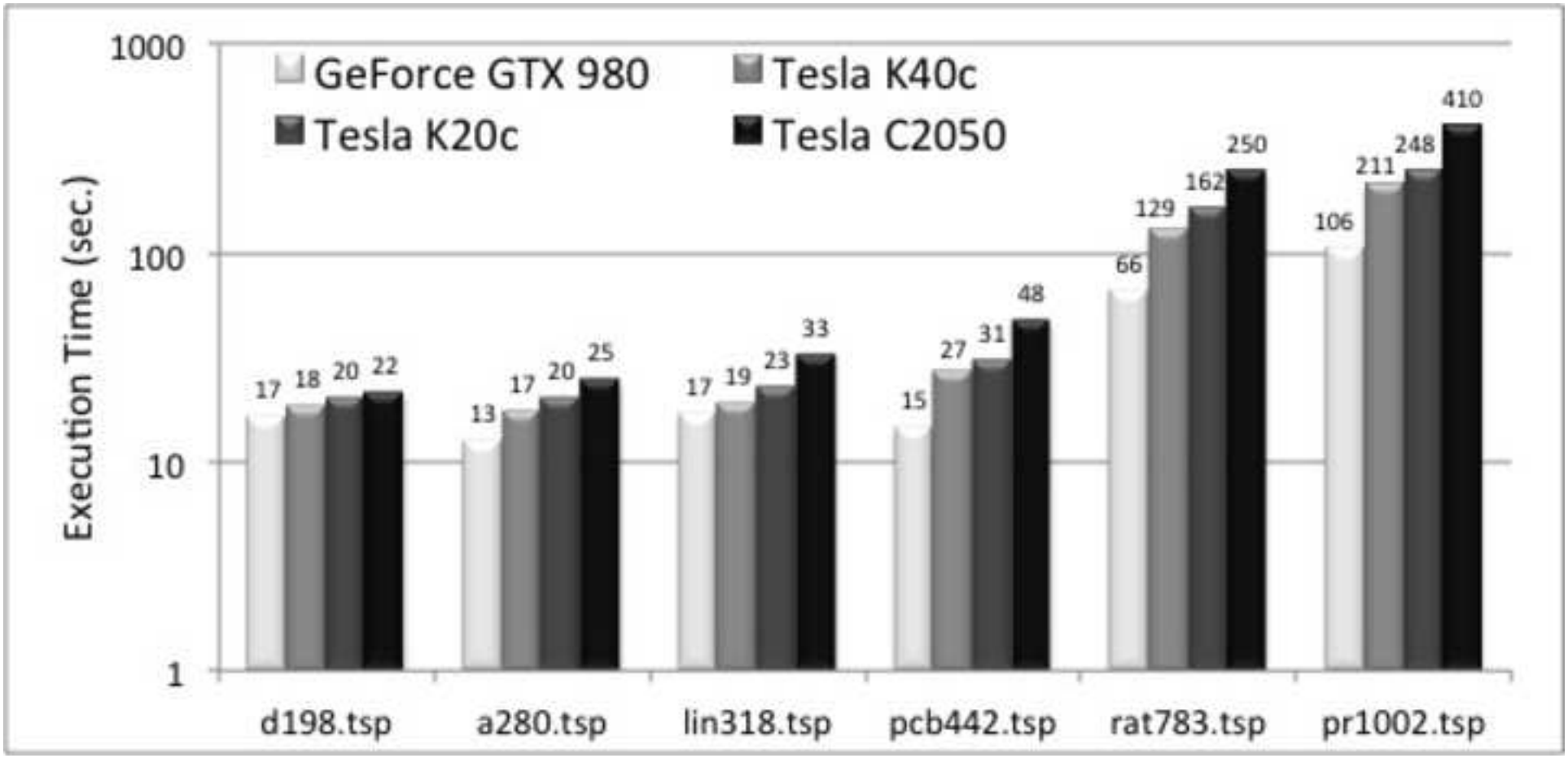


Figure 3
Click here to download Figure: Fig3.jpg

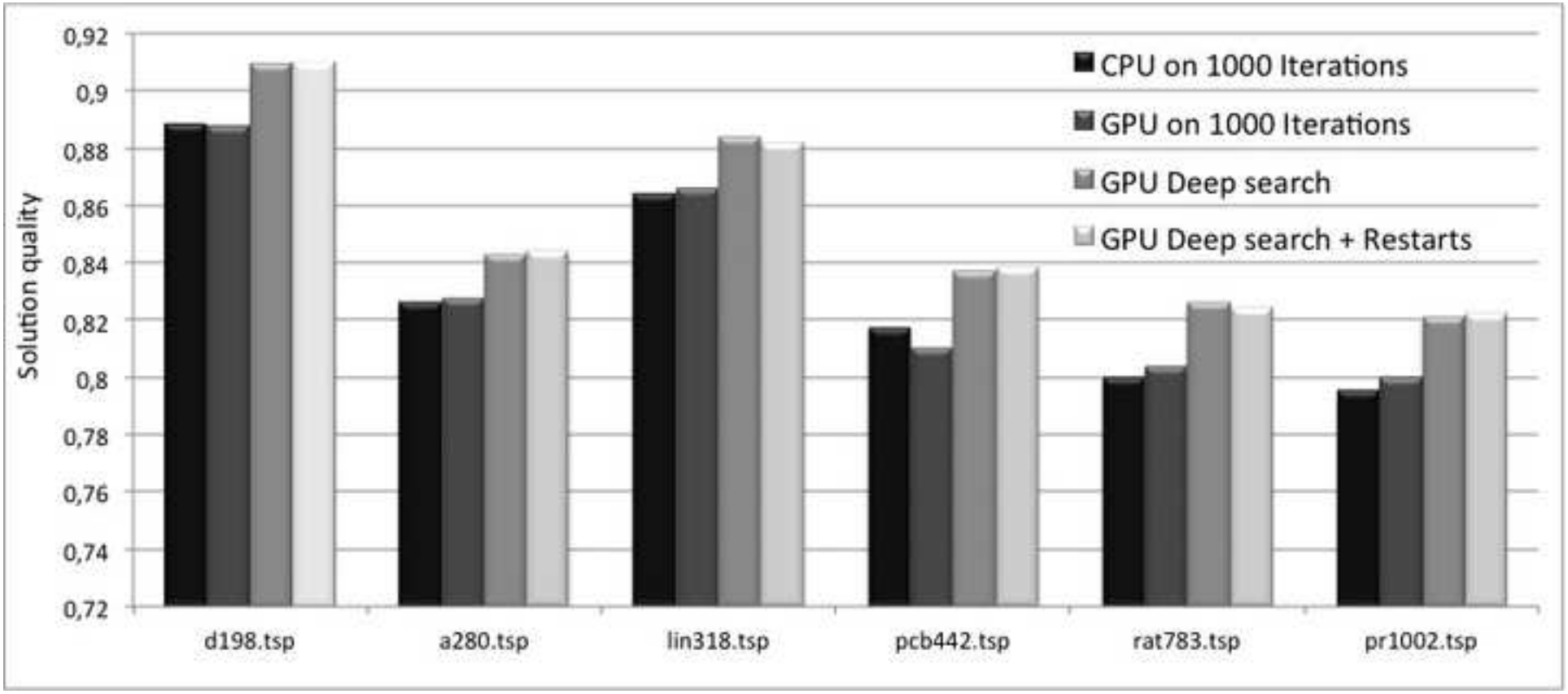


Figure 4
Click here to download Figure: Fig4.jpg

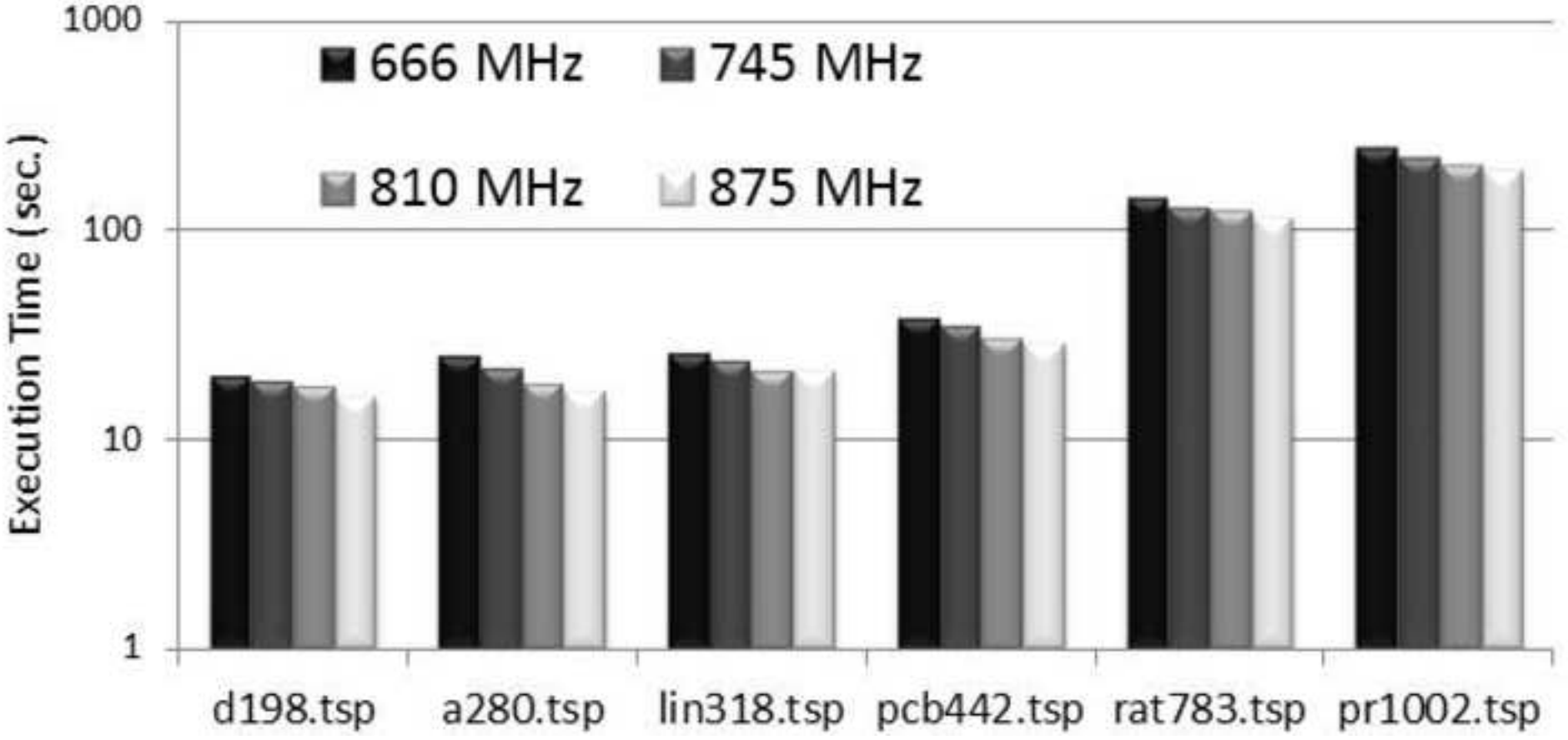


Figure 5
Click here to download Figure: Fig5.jpg

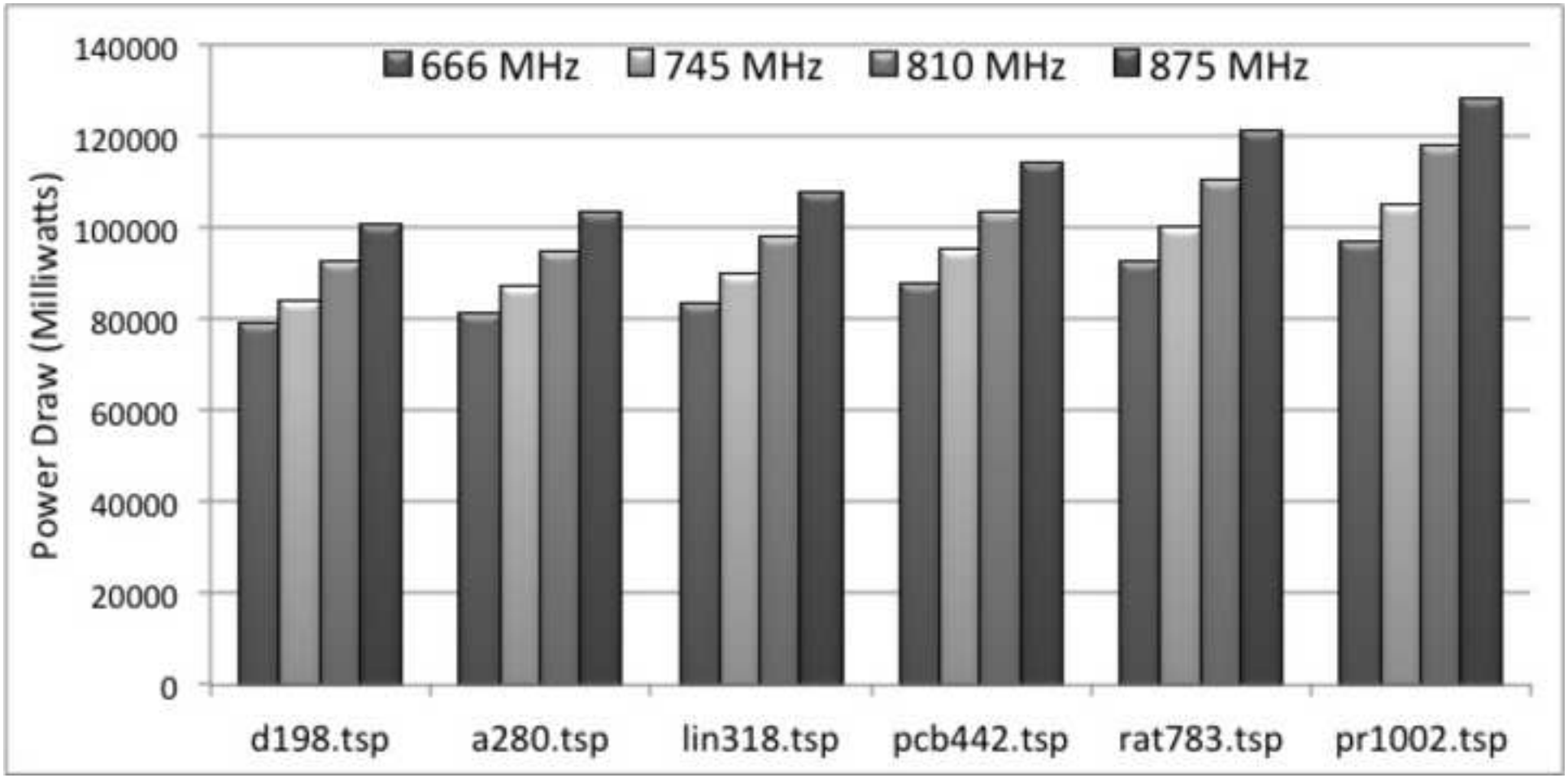


Figure 6a
Click here to download Figure: Fig6a.jpg

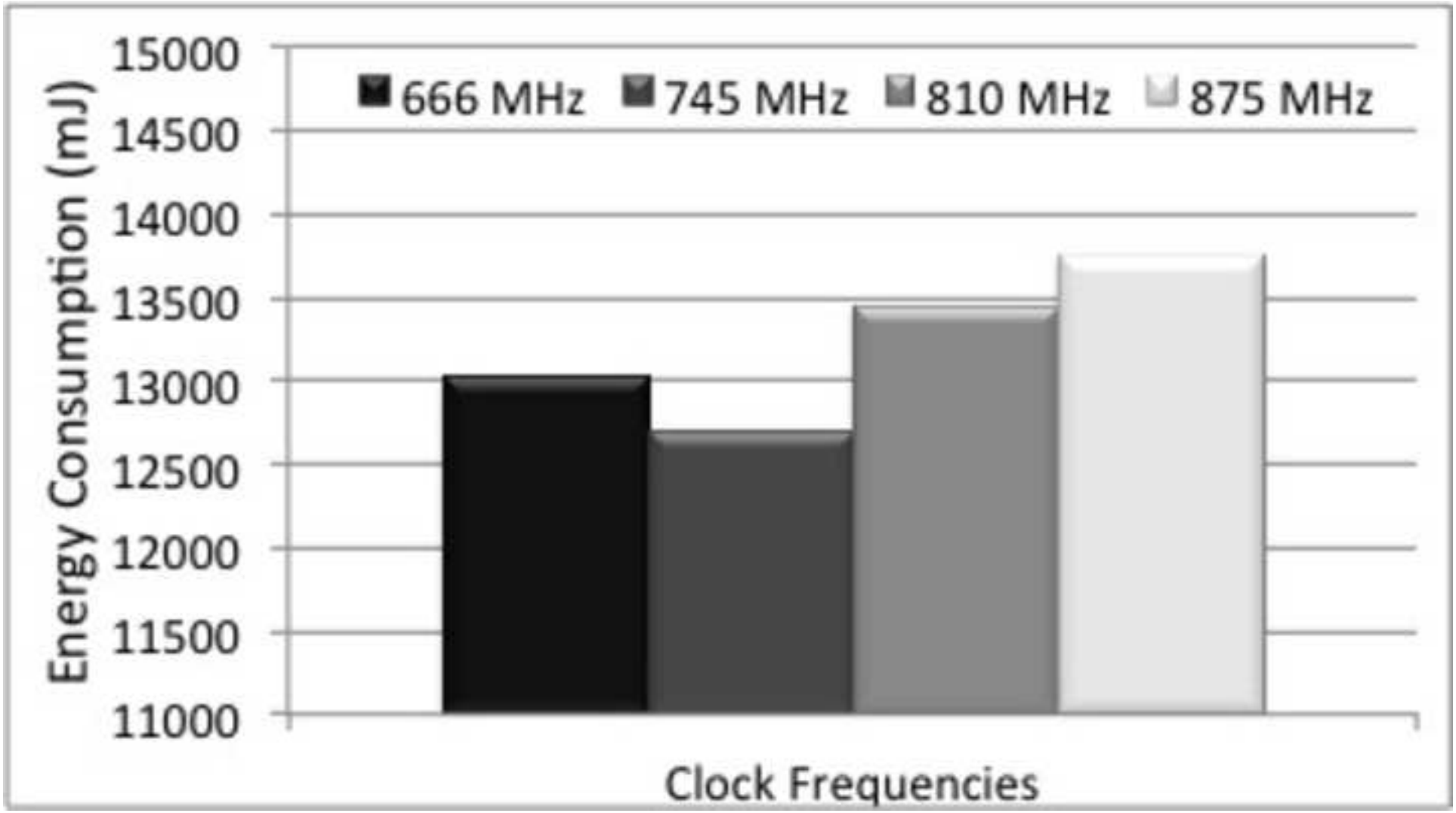


Figure 6b
Click here to download Figure: Fig6b.jpg

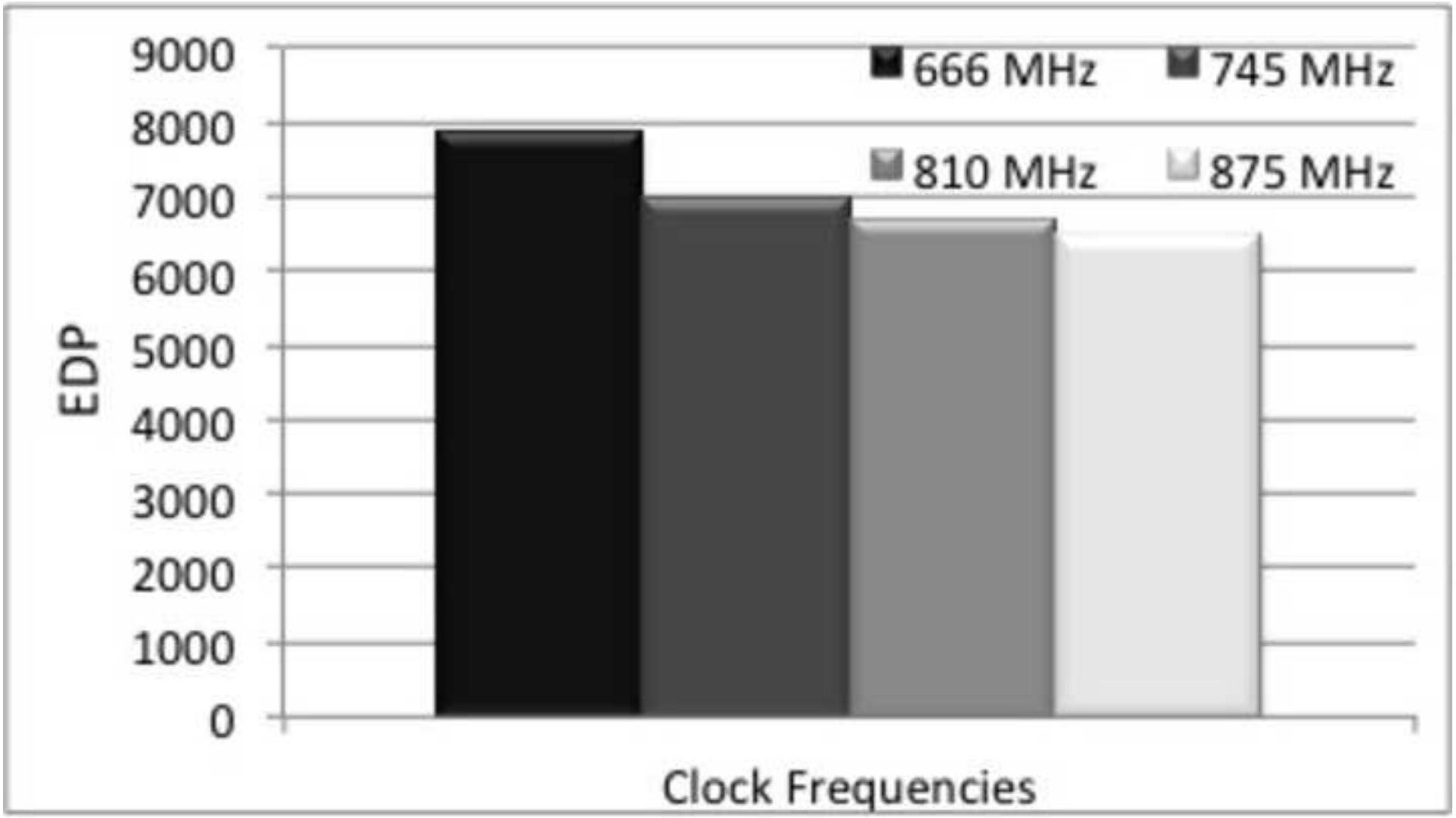


Figure 6c
Click here to download Figure: Fig6c.jpg