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

Comparative evaluation of platforms for parallel Ant Colony Optimization

Ginés D. Guerrero · José M. Cecilia · Antonio Llanes · José M. García · Martyn Amos · Manuel Ujaldón

Abstract The rapidly growing field of *nature-inspired computing* concerns the development and application of algorithms and methods based on biological or physical principles. This approach is particularly compelling for practitioners in high-performance computing, as natural algorithms are often *inherently parallel* in nature (for example, they may be based on a "swarm"-like model that uses a population of agents to optimize a function). Coupled with rising interest in nature-based algorithms is the growth in *heterogenous computing*; systems that use more than one kind of processor. We are therefore interested in the performance characteristics of nature-inspired algorithms on a *number* of different platforms. To this end, we present a new OpenCL-based implementation of the Ant Colony Optimization algorithm, and use it as the basis of extensive experimental tests. We benchmark the algorithm against existing implementations, on a wide variety of hardware platforms, and offer extensive analysis. This work provides rigorous foundations for future investigations of Ant Colony Optimization on high-performance platforms.

Keywords Heterogeneous Computing, Ant Colony Optimization, CUDA, OpenCL, APU, GPU.

Ginés D. Guerrero
National Laboratory for High Performance Computing, University of Chile (Chile).
E-mail: gguerrero@nlhpc.cl

José M. Cecilia, Antonio Llanes
Computer Science Department, Universidad Católica San Antonio de Murcia (Spain).
E-mail: {jmcecilia, allanes}@ucam.edu

José M. García
Computer Engineering Department, University of Murcia (Spain).
E-mail: jmgarcia@ditec.um.es

Martyn Amos
School of Computing, Mathematics and Digital Technology, Manchester Metropolitan University (UK).
E-mail: m.amos@mmu.ac.uk

Manuel Ujaldón
Computer Architecture Department, University of Malaga (Spain).
E-mail: ujaldon@uma.es

1 Introduction

Algorithms inspired by *natural* processes are gaining increasing acceptance, and are now used in a wide variety of application domains [28]. Many nature-inspired methods (such as the genetic algorithm [16], or particle swarm optimization [20]) are *population-based*, meaning that they maintain a *collection* of individual solutions which evolves or is modified as the computation proceeds. This structure naturally lends itself to *parallelisation*, and many parallel versions of such algorithms now exist [1].

One nature-based method that is proving to be increasingly popular is *ant colony optimization* (ACO) [8, 10,13]. This algorithm is based on foraging behaviour observed in colonies of real ants, and has been applied to a wide variety of problems, including vehicle routing [32], feature selection [6] and autonomous robot navigation [15]. The method generally uses simulated "ants" (i.e., mobile agents), which first construct tours or paths on a network structure (corresponding to solutions to a problem), and then deposit "pheromone" (i.e., signalling chemicals) according to the quality of the solution generated. The algorithm takes advantage of emergent properties of the multi-agent system, in that positive feedback (facilitated by pheromone deposition) quickly drives the population to high-quality solutions.

The original ACO method (called the *Ant System* [11]) was developed by Dorigo in the 1990s, and this

version (or slight variants thereof, such as the MAX-MIN Ant System (MMAS) [31]) is still in regular use [5,19,21]. Parallel versions of the Ant System have been developed [7,23,30,33] (see also [26] for a survey), and, in recent work, we present a graphics processor unit (GPU)-based version of ACO that, for the first time, parallelizes *both* main phases of the algorithm (that is, tour construction *and* pheromone deposition) [3,4].

The original version of our algorithm was developed for the CUDA (Compute Unified Device Architecture) platform[1], which offers easy access to the parallel processing capabilities of GPUs (thus facilitating so-called "GPGPU" or "general purpose GPU" computation). However, although it laid the foundations for general GPU-based computing, CUDA is proprietary to Nvidia, one of the dominant manufacturers in the GPU market. With that in mind, an alternative open standard was developed, which became known as *OpenCL* (Open Computing Language) [29]. This standard provides a common language, programming interfaces and hardware abstractions over a *wide range* of devices (CPUs, GPUs and other accelerators), and has contributed significantly to the growth of *heterogeneous computing* [2]. Importantly, OpenCL offers *portability* across combinations of operating system, GPU and other processors, which, in turn, have their own hardware costs and performance characteristics. It is therefore possible to write a portable, parallel algorithm for a specific problem, which may be run on a hardware/software combination that meets multiple constraints (cost, performance, and so on).

With that in mind, we present a new OpenCL-based version of our ACO algorithm, which may be run on a variety of platforms (from laptops to high-end servers). Our aim is to demonstrate how such an implementation may be used as the foundation for *high-performance, portable* ACO-based solutions. We benchmark our algorithm on a range of platforms and give an analysis about its scalability on high-end platforms.

The paper is organized as follows: in Section 2 we briefly describe our ACO-based algorithm and the process of migrating it to OpenCL. We then present the results of experimental investigations in Section 3, offer some analysis in Section 4, and then conclude in Section 5 with a brief discussion of our findings.

2 ACO algorithm

Our ACO-based solution to the Travelling Salesman Problem (TSP) is described in detail in [3,4], so here we simply give a brief overview in order to highlight specific issues arising from the migration to OpenCL.

The TSP is a well-known NP-hard optimization problem, and is often used as a standard benchmark for heuristic algorithms [18]. Indeed, it was the first problem to be solved using ACO [11], and our own work is a natural development of this. Briefly, the TSP involves finding the shortest ("cheapest") round-trip route that visits each of a number of "cities" exactly once. In what follows, we address the symmetric TSP on n cities, which may be represented as a complete weighted graph, G, of 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 ACO algorithm for TSP uses a number of simulated "ants" (or *agents*), which perform distributed search on a graph. Each ant moves on the graph until it completes a tour, and then offers this tour as its suggested solution. In order to achieve this latter step, each ant deposits "pheromone" on the edges that it visits during its tour. The quantity of pheromone deposited, if any, is determined by the *quality* of the solution relative to those obtained by the other ants. Pheromone levels on each edge "evaporate" over time (i.e., they are gradually reduced), in order to prevent the algorithm from being locked into sub-optimal solutions.

While building a tour, each ant probabilistically chooses the next city to visit based on two different sources of information: (1) heuristic information, obtained from inter-city distances, and (2) the pheromone trail, which facilitates indirect communication between ants via their *environment* (a process known as *stigmergy* [9]). The combination of local search and global signalling enables a process of directed positive feedback, by which the population quickly converges to a high-quality solution to the problem. The main body of the algorithm therefore has two main phases: (1) *tour construction*, and (2) *pheromone deposition*.

During tour construction, a number of ants build tours in parallel. Ants are initially placed at random, and they then repeatedly apply a probabilistic action choice rule in order to decide which city to visit next. Pheromone deposition occurs once all ants have constructed their tours; first, the pheromone levels on all edges are reduced by a constant factor (in order to simulate evaporation), and then pheromone is deposited on edges that ants have included in their tours (the precise amount for each edge in a particular tour being inversely proportional to the tour's length). In this way, edges that are used by many ants (and which are part of short tours) receive more pheromone, and are therefore more likely to be selected by ants in subsequent rounds

[1] Full technical details at http://docs.nvidia.com/cuda/index.html

(thus implementing the positive feedback process that we have already described).

2.1 Original CUDA implementation

We first briefly review the main characteristics of CUDA [24], for the benefit of readers who are unfamiliar with the programming model. CUDA is based on a hierarchy of abstraction layers; the *thread* is the basic execution unit; threads are grouped into *blocks*, each of which run on a single multiprocessor, where they can share data on a small but extremely fast memory. A *grid* is composed of blocks, which are equally distributed and scheduled among all multiprocessors. The parallel sections of an application are executed as *kernels* in a SIMD (Single Instruction Multiple Data) fashion, that is, with all threads running the same code. A kernel is therefore executed by a grid of thread blocks, where threads run simultaneously grouped in batches called *warps*, which are the scheduling units.

We now consider the implementation of each phase of the algorithm. The "traditional" task-based parallelism approach is based on the observation that ants run in parallel while searching for the best tour [4] (that is, parallelism is expressed at the level of individual ants). Within the basic model, each ant is associated with an individual thread, but this approach has three main drawbacks:

1. **Low degree of parallelism**. Because the number of ants used is generally a (linear) function of the problem size, the number of threads required is generally too low to fully exploit the resources of the GPU.
2. **Control dependencies**. *Warp divergences* (a situation where threads take different control-flow paths) can often arise when ants check the so-called *tabu list* - the record of cities already visited. Put simply, different threads in a warp may need to do different things, depending on which cities the different ants have visited, and this is expensive.
3. **Irregular memory access**. Because the ACO algorithm is inherently stochastic, this can produce an unpredictable *memory access pattern*. This prevents the GPU from taking advantage of *caching* schemes and other techniques for reducing memory access latency.

In previous work, we developed an alternative approach that places more emphasis on *data parallelism* [3]. We now briefly describe this algorithm, in order to establish the differences between the CUDA and OpenCL implementations.

When an ant makes a decision on which city to visit next, it must calculate heuristic information, as previously described. The heuristic information available to any one ant at a given time is *the same*, regardless of which ant is making the query, so it makes sense to separate out the computation of heuristic values 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, as described earlier. Evaporation is straightforward, as a single thread can independently lower each entry in the pheromone matrix by a constant factor. Deposition is more problematic, as each ant generates its own private tour in parallel, and will eventually visit the same edge as another ant. In order, therefore, to prevent race conditions, we require the use of CUDA atomic operations when accessing the pheromone matrix.

The efficiency of a parallel implementation is also affected by the *types* of operation on which it relies; in our code, scatter/gather operations [17] predominate (i.e., those which either write or read a large number of data items). As Table 2 reflects, the vast majority of operations are of the "gather" type; algorithms of this type are memory bounded and amenable to optimization via methods such as *coalescing* (Nvidia GPUs) and the use of SSE *vector instructions* (Intel CPUs). A comparative study [22] of these optimisations reveals similar impact on performance across platforms, which suggests that the experimental sections of the current paper will not suffer too much from platform-specific biases.

2.2 OpenCL migration

In this Section we briefly describe various issues that arose during the migration from CUDA to OpenCL. The foundations of OpenCL are based on the CUDA threading model, but with differences in terms of naming schemes and identifiers. We therefore used source-to-source translation in order to migrate our CUDA-based kernels to OpenCL. This mapping requires in-depth knowledge of both application programming in-

terface (API) models, as it is considerably more complex than simple instruction conversion. Also, OpenCL is still relatively young compared to CUDA, and does not provide the same functionality offered by its more mature partner.

The process of setting up a device for kernel execution differs *substantially* between CUDA and OpenCL. The APIs for context creation and data copying use different conventions for mapping the kernel onto the device processing elements, which may substantially affect the programming effort required to code and debug a parallel application. CUDA provides several libraries to enhance the functionality of its API. For example, our ACO algorithm uses the CURAND library [25] to generate pseudo-random numbers. This library is not directly implemented in OpenCL, where the main alternative is an implementation of the RANLUX pseudo-random number generator, called RANLUXCL [2]. Unfortunately, we found this library to be fairly wasteful in terms of memory, so we decided to implement our own, taking a C counterpart as a departure point [14].

3 Experimental results

In this Section we give the results of extensive comparative evaluations of ACO-based solutions to the TSP on different CPU, APU and GPU platforms. The underlying hardware platforms we tested are specified in Table 1.

For validation purposes, we use a baseline comparison with the sequential ANSI C code provided in [12]. The experimental setup (in terms of hardware/software) is listed in Table 3. We run our three ACO implementations (ANSI C, CUDA and OpenCL) on selected benchmark TSP instances from the well-known TSPLIB library [27]. All instances are defined on a complete graph, and distances are given as integers. Table 4 specifies the instances used; they were selected in order to ensure a representative sample, from "small" to "medium" and "large" (for reasons of practicality, we test only the high-end platforms on *pr*2392; these results are used for the later scalability analysis). Importantly, we note that our methods solve all instances *to optimality*; for the purposes of this paper, we are less interested in the *quality* of solutions produced, so in order to ensure a fair comparison we use instances that are solvable to optimality by our implementations as described in [3].

For all runs, we set the ACO parameters according to the values recommended in [12]; $\alpha = 1$, $\beta = 2$, $\rho = 0.5$, and $m = n$, meaning that the number of ants, m, is equal to the number of cities, n. We run each algorithm for 1000 iterations, and average timings over 1000 runs. CUDA times are obtained with a block size of 128 threads, and OpenCL local size is also set to 128.

Before discussing the results of our experiments, we consider several issues with respect to performance. Firstly, APUs are much more limited in terms of thermal design power, as they must also include the CPU. This means that execution units will need to be removed in order to keep power consumption down. Secondly, because the APU is a cost-effective solution, it does not have its own dedicated global memory, but instead it relies on an emulated global memory located in system memory. While this is good for performance when transferring data directly between the CPU and GPU, it means that it will also suffer in terms of overall bandwidth, as even low-end GPUs have more memory bandwidth.

We present a summary of our results in Figure 1. For each row (i.e., each platform, or hardware/software combination), we show execution times averaged over the small (top bar) and medium/large (bottom bar) instances. Note that times are measured in milliseconds (ms), and represent the elapsed time for a *single iteration* of the platform-specific algorithm, averaged over 100 runs of 1000 iterations each (as opposed to the average run time for the whole algorithm). We focus on the average time for a single iteration precisely because we are interested in the *overall kernel performance* on each platform, so this fine-grained approach gives us the insights that we require.

4 Analysis

We now give an analysis of the performance of each category of hardware platform.

4.1 Desktop PCs

Beginning with the E-350 APU, we see that the CPU does not perform particularly well. This is expected, based on the architecture's emphasis on power consumption over performance for this consumer market. However, when moving to the GPU we see that, for small problem instances, it actually scales better in terms of overall computational time than the FirePro V8800 for the same base architecture.

Looking closely at the numbers, the E-350 APU, which is outclassed by factors of 37 and 17 for computational power (*execution resources* × *clock speed*) and memory bandwidth respectively, manages to only perform at roughly 1/10th the speed. We attribute this to the APU's ability to quickly transfer data to and

[2] See https://bitbucket.org/ivarun/ranluxcl/

Table 1: Summary of hardware features for the CPUs, APUs and GPUs used during our experimental survey.

(a) Processors found in high-end servers.

	CPU	GPU	GPU
Release date	Q4 2009	Q4 2009	Q1 2010
Codename	Intel Westmere	Nvidia Fermi	ATI Cypress
Commercial model	Xeon E5620	Tesla C2050	FirePro V8800
No. cores @ speed	4 @ 2.4 GHz	-	-
No. stream processors	-	448 @ 1.15 GHz	1600 @ 925 MHz
L2 cache size	12 MB.	768 KB.	512 KB.
DRAM memory size	16 GB.	3 GB.	2 GB.
DRAM type	DDR3	GDDR5	GDDR5
Memory bus width	128 bits	384 bits	256 bits
Memory clock	1066 MHz	2 x 1.5 GHz	4 x 1.15 GHz
Memory bandwidth	17 GB/s	144 GB/s	147.2 GB/s

(b) Processors found in desktop PCs.

	CPU on APU	GPU on APU
Release date	Q1 2010	Q1 2010
Codename	AMD Llano	ATI Redwood
Commercial model	E-350	ATI HD 6310
No. cores @ speed	2 @ 1.6 GHz	-
No. stream processors	-	80 @ 492 MHz
L2 cache size	2 x 512 KB.	-
DRAM memory size	4 GB. (shared)	4 GB. (shared)
DRAM type	DDR3	DDR3
Memory bus width	64 bits	64 bits
Memory clock	1066 MHz	1066 MHz
Memory bandwidth	8.5 GB/s	8.5 GB/s

(c) Processors found in laptops.

	CPU on APU	GPU on APU	GPU
Release date	Q2 2011	Q2 2011	Q1 2011
Codename	AMD Llano	ATI Redwood	ATI Redwood
Commercial model	A6-3420	Radeon HD 6520	Radeon HD 6650M
No. cores @ speed	4 @ 1.4 GHz	-	-
No. stream processors	-	320 @ 400 MHz	480 @ 600 MHz
L2 cache size	4 MB.	-	-
DRAM memory size	4 GB. (shared)	4 GB. (shared)	1 GB. (exclusive)
DRAM type	DDR3	DDR3	DDR3
Memory bus width	64 bits	64 bits	128 bits
Memory clock	1333 MHz	1333 MHz	900 MHz
Memory bandwidth	10.6 GB/s	10.6 GB/s	14.4 GB/s

Table 2: Characterization of the stages involved in our ACO implementation on GPUs.

Algorithm stage	Operator	Key features	CUDA kernel
Generation of `choice_info` array	Gather	Data parallelism fully exploited	`choice_info`
Tour construction	Gather	Optimized via `choice_info` array	`Next_tour`
Tabu list update	Scatter	Optimized via an array in register file	`Next_tour`
Pheromone evaporation	Scatter	Concurrent updates, no queries	`Pheromone`
Pheromone deposit	Gather	Single update using atomic operations	`Pheromone`

Table 3: Software resources used for each hardware platform in our experimental study.

Target hardware	*Software tools*
Intel Xeon CPU	gcc compiler, 4.3.4 version with the -O3 flag set
Nvidia Tesla GPU	CUDA compilation tools, release 4.0
ATI FirePro GPU	Software Suite 8.85.7.2 and OpenCL runtime v831.4
AMD APUs and dedicated GPUs	AMD's APP SDK 2.6, Catalyst driver 11.12, OpenCL runtime version 793.1

from the CPU to the GPU. However, as the input size increases this advantage disappears, as raw computational throughput and bandwidth become more important than latency. Comparing these results to the Tesla C2050 GPU, the APU is at an even greater disadvantage, due to its VLIW architecture (compared to the scalar and compute-oriented architecture of the C2050). This should change, however, with AMD future generations of APUs, which consider a GPU based on their newly-released Graphics Core Next (GCN) architecture. GCN greatly improves computational throughput, by moving scheduling from the compiler to the hardware.

4.2 Laptop computers

Moving to the A6-3420M APU, we see very similar results as with the E-350 APU. Here, our integrated GPU (iGPU) has roughly 3 times the amount of computational resources, but only 1/4 more bandwidth. This is evident in the scaling of the algorithm, as we go from 200 ms. with the E-350 APU to 148 ms. with the iGPU, a near exact scaling of the bandwidth advantage that the iGPU possesses.

As we increase the size of the problem instance, we then see that performance becomes constrained more by computational resources than by bandwidth. Our sim-

Table 4: TSP instances used in our study.

	Small dataset				*Medium/Large dataset*				
Graph name	d198	a280	lin318	pcb442	rat783	pr1002	pcb1173	d1291	pr2392
Number of cities	198	280	318	442	783	1002	1173	1291	2392
Best tour length	15780	2579	42029	50778	8806	259045	56892	50801	378032

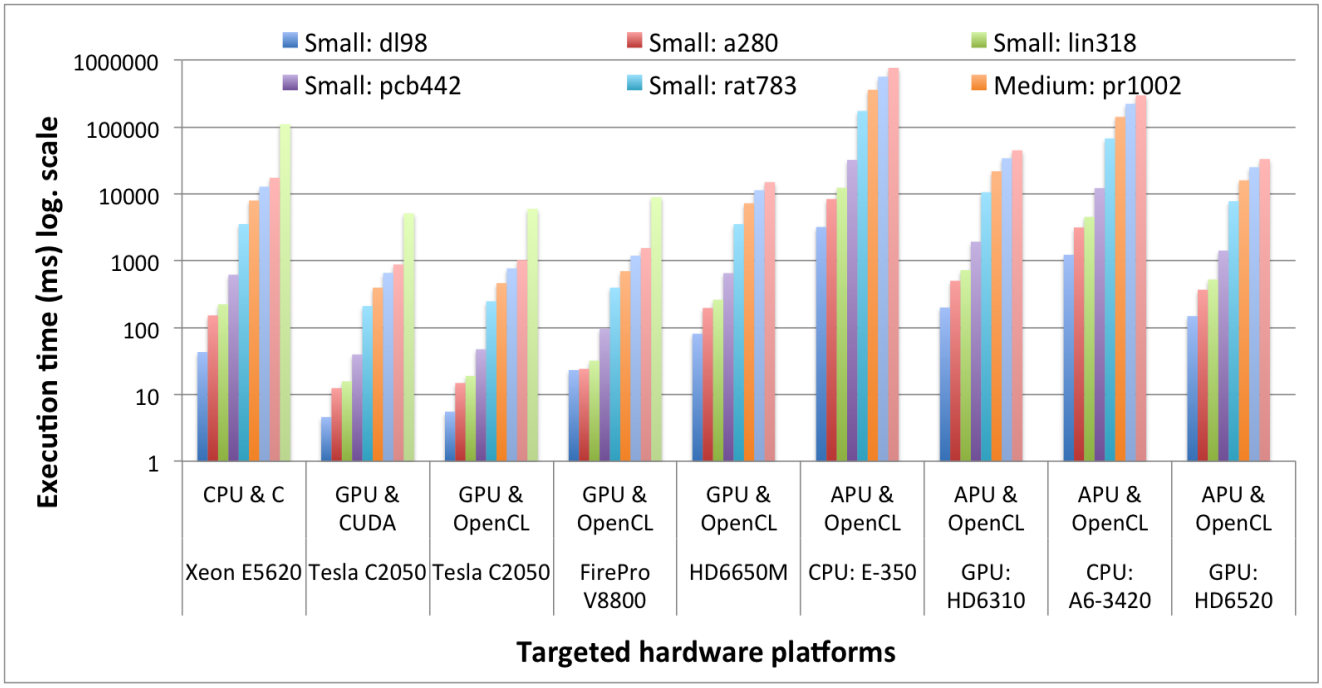

Fig. 1: Summary of experimental results. X axis shows each platform, Y axis (logarithmic plot) shows execution time (ms) for one iteration. Bars are ordered from smallest (Left) to largest (Right) instances.

plest comparison is to the A6-3420M's dedicated GPU (which has 2.25 times the computational power), where, as the input size increases, the difference between the two solutions approaches this limitation. This shows that, while the memory system influences ACO performance, computational resources become the dominating factor in overall performance. For PCI-express 2.0, the maximum bandwidth (unidirectional) is 8 GB/s, while using zero copy the APU is able to reach nearly 16 GB/s. If this had been taken into account, the results for the APU and dedicated GPUs would in fact be much closer, as this type of workload/data transfer is playing to the APU's strength.

Ending with the dedicated GPU (dGPU), we see a similar speedup increase, just as we did for the E-350 and A6-3420M APUs. Again, for small input sizes, latency and bandwidth are much more important than the computational abilities of the device, as there are fewer threads to interleave in order to hide memory accesses. This is visible in the dGPU, which has 7 and 2 times the amount of computational power and memory bandwidth as the E-350 APU, while performing just over twice as quickly for the d198 dataset.

As we increase the complexity of the workload, we again see that memory bandwidth becomes a less important issue, and computational power becomes the main contributing factor for overall performance. Comparing once again to the Tesla C2050, the APU solution does not perform as well as we had hoped.

4.3 High-end platforms

High-end processors usually cover large-scale applications, and our performance analysis emphasises *scalability*. Table 5 shows the behaviour of the execution time when the problem size increases. We compare execution times on small, medium and large instances, and obtain the coefficient or multiplier which separates them. The larger this coefficient is for a given processor, the poorer the degree of scalability.

Table 5: Scalability on high-end-platforms depending on hardware and programming methods. FirePro behaves better on larger problem instances, followed by Tesla using OpenCL (with CUDA very close), and finally Xeon using C.

	Scalability →	Short range	Mid range	Long range
Language/API	HW platform	Time(pr2392)/ Time(rat783)	Time(rat783)/ Time(d198)	Time(pr2392)/ Time(d198)
C	CPU Xeon	31.24x	82.30x	2571.46x
CUDA	GPU Tesla	24.43x	45.74x	1117.88x
OpenCL	GPU Tesla	24.16x	44.83x	1083.52x
OpenCL	GPU FirePro	22.70x	17.09x	388.12x

Looking at those numbers, we see that when comparing Tesla versus FirePro (GPUs running the same OpenCL code), Tesla is 1.5x-2x *faster*, but FirePro *scales* better. Also, comparing languages on the same Tesla hardware, CUDA is 1.15x-1.20x faster, but OpenCL scales slightly better. Finally, comparing GPU results with numbers on the CPU, the GPU is faster and scales better: The speed-up factor ranges 9x-15x on four small data sets, 17x-20x on four medium data sets, and finally 21.5x on the large data set.

5 Conclusions

In this paper we presented a comprehensive performance review of different platforms for Ant Colony Optimization, an emerging and fast-growing nature-inspired algorithm. We discussed the translation of our previous algorithm from CUDA to OpenCL, and highlighted certain issues that may be faced by other practitioners in future. We then performed a performance analysis of three variants of the ACO algorithm, using the Travelling Salesman Problem as a benchmark, and focussed on issues of scalability.

In general, GPUs are superior to CPUs on the high-end segment: they yield twenty times faster execution on large problem instances. The GPU-CPU difference is similar on desktops and laptops, 10-20x in favor of GPUs. At an early stage of its evolution, the APU offers a low-cost platform, without powerful computational units nor swift memory data paths. Our results demonstrate that these two issues have a severe impact on performance.

The growth of heterogeneous systems represents a solid trend in modern systems, and we believe that future work on Ant Colony Optimization in this domain can benefit from the promising insights into scalability demonstrated by our experimental study.

Acknowledgements This work is jointly supported by the Fundación Séneca (Agencia Regional de Ciencia y Tecnología, Región de Murcia) under grant 15290/PI/2010, by the Spanish MEC and European Commission FEDER under grant TIN2012-31345, by the UCAM under grant PMAFI/26/12, by the Junta de Andalucía under Project of Excellence P12-TIC-1741 and by the supercomputing infrastructure of the NLHPC (ECM-02). We also thank NVIDIA for hardware donation under CUDA Teaching Center 2011-14, CUDA Research Center 2012-14 and CUDA Fellow 2012-14 Awards.

References

1. 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
2. Brodtkorb, A.R., Dyken, C., Hagen, T.R., Hjelmervik, J.M., Storaasli, O.O.: State-of-the-art in heterogeneous computing. Scientific Programming **18**(1), 1–33 (2010)
3. 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)
4. 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)
5. 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
6. 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
7. 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
8. Dorigo, M., Birattari, M., Stutzle, T.: Ant colony optimization. Computational Intelligence Magazine, IEEE **1**(4), 28–39 (2006)
9. Dorigo, M., Bonabeau, E., Theraulaz, G.: Ant algorithms and stigmergy. Future Generation Computer Systems **16**(8), 851–871 (2000)
10. 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)

11. 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)
12. Dorigo, M., Stutzle, T.: Ant Colony Optimization. Bradford Company (2004)
13. Dorigo, M., Stützle, T.: Ant colony optimization: overview and recent advances. In: Handbook of metaheuristics, pp. 227–263. Springer (2010)
14. Flannery, B.P., Press, W.H., Teukolsky, S.A., Vetterling, W.: Numerical recipes in c. Press Syndicate of the University of Cambridge, New York (1992)
15. 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
16. Goldberg, D.E.: Genetic algorithms in search, optimization, and machine learning. Addison-Wesley Professional (1989)
17. He, B., Govindaraju, N.K., Luo, Q., Smith, B.: Efficient gather and scatter operations on graphics processors. In: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing, pp. 46–57. ACM (2007)
18. Johnson, D.S., McGeoch, L.A.: The traveling salesman problem: A case study in local optimization. In: J. Lenstra, E. Aarts (eds.) Local Search in Combinatorial Optimization, pp. 215–310. John Wiley and Sons (1997)
19. 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
20. Kennedy, J., Eberhart, R.: Particle swarm optimization. In: Neural Networks, 1995. Proceedings., IEEE International Conference on, vol. 4, pp. 1942–1948. IEEE (1995)
21. 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
22. Lee, V.W., Kim, C., Chhugani, J., Deisher, M., Kim, D., Nguyen, A.D., Satish, N., Smelyanskiy, M., Chennupaty, S., Hammarlund, P.: Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu. In: ACM Intl. Symposium on Computer Architecture, pp. 451–460. ACM (2010)
23. 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)
24. Nickolls, J., Buck, I., Garland, M., Skadron, K.: Scalable parallel programming with cuda. Queue **6**(2), 40–53 (2008)
25. Nvidia: (2011). CUDA Toolkit 4.0 CURAND Guide. `http://developer.download.nvidia.com/compute/DevZone/docs/html/CUDALibraries/doc/CURAND\_Library.pdf`
26. 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
27. Reinelt, G.: TSPLIB - a Traveling Salesman Problem library. ORSA Journal on Computing **3**(4), 376–384 (1991). Library available at http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/
28. Rozenberg, G., Bäck, T., Kok, J.N.: Handbook of Natural Computing. Springer (2011)
29. Stone, J.E., Gohara, D., Shi, G.: OpenCL: A parallel programming standard for heterogeneous computing systems. Comput Sci Eng **12**(3), 66–72 (2010). DOI 10.1109/MCSE.2010.69
30. Stützle, T.: Parallelization strategies for ant colony optimization. In: Parallel Problem Solving from Nature (PPSN V), pp. 722–731. Springer (1998)
31. Stutzle, T., Hoos, H.H.: MAX-MIN ant system. Future Generation Computer Systems **16**(8), 889–914 (2000)
32. 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
33. 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)