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

GPU Computing in Discrete Optimization
Part II: Survey Focused on Routing Problems

Christian Schulz · Geir Hasle · André R. Brodtkorb · Trond R. Hagen

Received: date / Accepted: date

Abstract In many cases there is still a large gap between the performance of current optimization technology and the requirements of real world applications. As in the past, performance will improve through a combination of more powerful solution methods and a general performance increase of computers. These factors are not independent. Due to physical limits, hardware development no longer results in higher speed for sequential algorithms, but rather in increased parallelism. Modern commodity PCs include a multi-core CPU and at least one GPU, providing a low cost, easily accessible heterogeneous environment for high performance computing. New solution methods that combine task parallelization and stream processing are needed to fully exploit modern computer architectures and profit from future hardware developments. This paper is the second in a series of two. Part I gives a tutorial style introduction to modern PC architectures and GPU programming. Part II gives a broad survey of the literature on parallel computing in discrete optimization targeted at modern PCs, with special focus on routing problems. We assume that the reader is familiar with GPU programming, and refer the interested reader to Part I. We conclude with lessons learnt, directions for future research, and prospects.

Keywords Discrete Optimization; Parallel Computing; Heterogeneous Computing; GPU; Survey; Introduction; Tutorial; Transportation, Travelling Salesman Problem; Vehicle Routing Problem

Corresponding author: Geir Hasle. E-mail: geir.hasle@sintef.no

SINTEF ICT, Dept. of Applied Mathematics, P.O. Box 124 Blindern, NO-0314 Oslo, Norway

1 Introduction

In Part I of this paper [9], we give a brief introduction to parallel computing in general and describe modern computer architectures with multi-core processors for task parallelism and accelerators for data parallelism (stream processing). A simple prototype of a GPU based local search procedure is presented to illustrate the execution model of GPUs. Strategies and guidelines for software development and performance optimization are given. On this background, we here, in Part II, give a survey of the existing literature on parallel computing in discrete optimization targeted at modern PC platforms. With few exceptions, the work reported focuses on GPU parallelization.

Section 2 contains the bulk of Part II. It starts with an overall description of our literature search before we refer to early work on non-GPU accelerators in 2.1. The rest of the section is structured according to type of optimization method. As a reflection of the number of publications, the first and most comprehensive part concerns metaheuristics. We give accounts of the literature on swarm intelligence, population based metaheuristics, and trajectory based metaheuristics in subsections 2.2, 2.3, and 2.4, respectively. For all optimization methods, we briefly describe the method in question, present a survey of papers, often also in tabular form, and synthesize the insights gained. An in-depth discussion of important papers on routing is given, if any. Subsection 2.5 discusses GPU-based implementations of shortest path algorithms. In Section 3 we give an overview over GPU implementations of metaheuristics applied to problems that are not related to routing, using the structure from Section 2. In 3.4, we discuss hybrid metaheuristics. As Linear Programming and Branch & Bound are important bases for methods in discrete optimization, we give a brief account of GPU implementations in 3.5. We conclude Part II with lessons learnt and directions for future research in Section 4, followed by summary and conclusions in Section 5.

2 Literature Survey with Focus on Routing Problems

Parallel methods to alleviate the computational hardness of discrete optimization problems (DOPs) are certainly older than the modern PC architecture. Parallelized heuristics, metaheuristics, and exact methods for DOP have been investigated since the 1980s and there is a voluminous literature, see for instance [87] and [4] for general surveys, and [21] for a survey focused on the VRP. Most of the work is based on task parallelism, but the idea of using massive data parallelism to speed up genetic algorithms dates back to the early 1990s, see for instance [85].

It should be clear that population based metaheuristics and methods from swarm intelligence such as Ant Colony Optimization lend themselves to different types of parallelization at several levels of granularity. Both task and data parallelization are possible, and within both types there are many alternative parallelization schemes. Also, the neighborhood exploration in local search that is the basis and the bottleneck of many trajectory based metaheuristics is inherently parallel. At a fine-grained level, the evaluation of objective components and constraints for a given neighbor may be executed in parallel. A more coarse-grained parallelization results from neighborhood splitting. What may be regarded as the simplest metaheuristic — multi-start local search — is embarrassingly parallel.

Again, both data and task parallelization may be envisaged, and there are many non-trivial design decisions to make including the parallelization scheme.

Branch & Bound and Branch & Cut are basic tree search algorithms in exact methods for DOP. At least conceptually, they are easy to parallelize, but load balancing and scaling are difficult issues. We refer to [22] and [72] for in-depth treatments. Commercial MILP solvers typically have task parallel versions that are well suited for multi-core processors. As far as we know there exists no commercial MILP solver that exploits stream processing accelerators. More sophisticated exact MILP methods such as Branch & Cut & Price are harder to parallelize [73].

In a literature search (2012), we found some 100 publications on GPU computing in discrete optimization. They span the decade 2002-2012. With only a few exceptions they discuss GPU implementation of well-known metaheuristics, or problem specific special algorithms. Very few address the combined utilization of the CPU and the GPU. Below, we give some overall characterizations of the publications found, before we structure and discuss the literature in some detail.

As for applications and DOPs studied, 28 papers describe research on routing problems, of which 9 focus on the Shortest Path Problem (SPP), 16 discuss the TSP, and only 3 study the VRP. As GPU computing for the SPP is peripheral to the goals of this paper, we only give a brief survey of the literature in Section 2.5. Also relevant to transportation there is a paper on route selection for car navigation [11], and one on route planning in aerial surveillance [79]. Bleiweiss [8] describes an efficient GPU implementation of parallel global pathfinding using the CUDA programming environment. The application is real time games where a major challenge is autonomous navigation and planning of thousands of agents in a scene with both static and dynamic moving obstacles. Rostrup et al. [78] describe a GPU implementation of Kruskal's algorithm for the Minimum Spanning Tree problem.

Among other DOPs and applications studied are:

- Allocation of tasks to heterogeneous processing units
- Task matching
- Flowshop scheduling
- Option pricing
- FPGA placement
- VLSI circuit optimization
- Protein sequence alignment in bioinformatics
- Sorting
- Learning
- Data mining
- Permutation perceptron problem
- Knapsack problem
- Quadratic Assignment Problem
- 3-SAT and Max-SAT
- Graph coloring

Not surprisingly, the bulk of the literature with some 80 papers discusses implementations of swarm intelligence methods and population based metaheuristics. Of the 41 swarm intelligence papers found, there are 23 on Ant Colony Optimization (ACO) and 18 on Particle Swarm Optimization (PSO). Most of the PSO publications focus on continuous optimization. The identified literature on popula-

tion based metaheuristics (Evolutionary Algorithms, Genetic Algorithms, Genetic Programming, Memetic Algorithms, and Differential Evolution) also consists of 41 publications. The remaining publications cover (number of publications):

- Metaheuristics in general (1)
- Immune Systems (2)
- Local Search (8)
- Simulated Annealing (3)
- Tabu Search (3)
- Special purpose algorithms (2)
- Linear Programming (4)

The most commonly used basis for justifying a GPU implementation is speed comparison with a CPU implementation. This is useful as a first indication, but it is not sufficient by itself. Important aspects such as the utilization of the GPU hardware, are typically not taken into consideration. Moreover, the CPU code used for comparison is normally unspecified, and thus unknown to the reader. We refer to Section 4 for a detailed discussion on speedup comparison. Often, an algorithm can be organized in different ways, which in turn can have a variety of GPU implementations, each using different GPU specifics such as shared memory. Only a few papers discuss and compare different algorithmic approaches on the GPU. A thorough investigation of hardware utilization, e.g., through profiling of the implemented kernels, is missing in nearly all of the papers. For these, we will simply quote the reported speedups. If a paper provides more information on the CPU implementation used, different approaches, or profiling, we will mention this explicitly.

2.1 Early works on non-GPU related accelerators

Early papers utilize hardware such as Field-Programmable Gate Arrays (FPGAs). Guntsch et al. [34] is the earliest paper in our survey. It appears in 2002 and proposes a design for an Ant Colony Optimization (ACO) variant, called Population-based ACO (P-ACO), that allows efficient FPGA implementation. In [81], an overlapping set of authors report from the actual implementation of the P-ACO design. They conduct experiments on random instances of the Single Machine Total Tardiness Problem (SMTTP) with number of jobs ranging from 40 to 320 and report moderate speedups between 1.6 and 10 relative to a software implementation. Scheuermann et al. continue their work on ACO for FGPAs in [80], where they propose a new ACO variant called Counter-based ACO. The algorithm is designed such that it can easily be mapped to FPGAs. In simulations they apply this new method to the TSP.

2.2 Swarm Intelligence Metaheuristics

The emergent collective behavior in nature, in particular the behavior of ants, birds, and fish is the inspiration behind Swarm intelligence metaheuristics. For an introduction to Swarm intelligence, see for instance [42]. Swarm intelligence metaheuristics are based on communication between many, but relatively simple,

agents. Hence, parallel implementation is a natural idea that has been investigated since the birth of these methods. However, there are non-trivial design issues regarding parallelization granularity and scheme. A major challenge is to avoid communication bottlenecks.

The methods of this category that we have found in the literature of GPU computing in discrete optimization are Ant Colony Optimization (ACO), Particle Swarm Optimization (PSO) and Flocking Birds (FB). ACO is the most widely studied Swarm intelligence metaheuristic (23 publications), followed by PSO (18) and FB (3). ACO is also the only swarm intelligence method applied to routing problems in our survey, which is why we will discuss it here. For an overview of GPU implementations of the other swarm intelligence methods, we refer to Section 3.1.

In ACO, there is a collection of ants where each ant builds a solution according to a combination of cost, randomness and a global memory, the so-called pheromone matrix. Applied to the TSP this means that each ant constructs its own solution. Afterwards, the pheromone matrix is updated by one or more ants placing pheromone on the edges of its tour according to solution quality. To avoid stagnation and infinite growth, there is a pheromone evaporation step added before the update, where all existing pheromone levels are reduced by some factor. There exist variants of ACO in addition to the basic Ant System (AS). In the Max-Min Ant System (MMAS), only the ant with the best solution is allowed to deposit pheromone and the pheromone levels for each edge are limited to a given range. Proposed by Stützle, the MMAS has proven to be one of the most efficient ACO metaheuristics. The most studied problem with ACO is the TSP. There are also several ACO papers on the Shortest Path Problem and variants of the VRP.

Parallel versions of ACO have been studied extensively in the literature, and several concepts have been developed. The two predominant, basic parallelization schemes are parallel ants, where one process/thread is allocated to each ant, and the multiple colonies approach. In [71], Pedemonte et al. introduce a new taxonomy for classifying parallel ACO algorithms and also present a systematic survey of the current state-of-the-art on parallel ACO implementations. As part of the new taxonomy they describe the master-slave category, where a master process manages global information and slave processes perform subordinate tasks. This concept can again be split into coarse-grained and fine-grained. In the former, the slaves compute whole solutions, as done in parallel ants. In the latter, the slaves only perform parts of the computation for one solution. Pedemonte et al. consider a wide variety of parallel computing platforms. However, out of the 69 publications surveyed, only 13 discuss multi-core CPU (9) and GPU platforms (4). Table 1 presents an overview of the routing related GPU papers implementing ACO that we found in the literature, showing which steps of ACO are performed on the GPU in what fashion by which paper.

ACO exhibits apparent parallelism in the tour construction phase, as each ant generates its tour independently. The inherent parallelism has led to early implementations of this phase on the GPU using the graphics pipeline. In Catala et al. [13] and Wang et al. [90] fragment shaders are used to compute the next city selection. In both papers, the necessary data is stored in textures and computational results are made available by render-to-texture, enabling later iterations to use earlier results. In [90], Wang et al. assign to each ant-city combination a unique (x, y) pixel coordinate and only generate one fragment per pixel. This leads to a

Author	Year	Problem	Algorithm	GPU(s)
Catala et al. [13]	2007	OP	ACO	GeForce 6600 GT
Bai et al. [5]	2009	TSP	multi colony MMAS	GeForce 8800 GTX
Li et al. [51]	2009	TSP	MMAS	GeForce 8600 GT
Wang et al. [90]	2009	TSP	MMAS	Quadro Fx 4500
You [96]	2009	TSP	ACO	Tesla C1060
Cecilia et al. [14]	2011	TSP	ACO	Tesla C2050
Delévacq et al. [25]	2012	TSP	MMAS & multi-colony	2 x Tesla C2050
Diego et al. [27]	2012	VRP	ACO	GeForce 460 GTX
Uchida et al. [89]	2012	TSP	AS	GeForce 580 GTX

Author	Tour construction			Ph. update		Max.	CPU
	GP	CUDA: one-ant-per		GP	CUDA	Speedup	code
		thread	block				
Catala et al. [13]	x						[67]
Bai et al. [5]		x			x	2.3	?
Li et al. [51]			x		x	11	?
Wang et al. [90]	x					1.1	?
You [96]		x				21	?
Cecilia et al. [14]		x	x		x	29	[28]
Delévacq et al. [25]		x	x		-/x	23.6	?
Diego et al. [27]		x			x	12	?
Uchida et al. [89]			x		x	43.5	own

Table 1: ACO implementations on the GPU related to routing. Legend: OP – orienteering problem; Ph. – pheromone; GP – graphics pipeline; -/x – in some settings; ? – unknown CPU code.

conceptually simple setup that needs multiple passes to compute the result. In [13] Catala et al. relate one pixel to an ant at a certain iteration and generate one fragment per city related to this pixel. The authors utilize depth testing to select the next city, and also provide an alternative implementation of tour construction using a vertex shader.

With the arrival of CUDA and OpenCL, programming the GPU became easier and consequently more papers studied ACO implementations on the GPU. In CUDA and OpenCL there is the basic concept of having a thread/workitem as basic computational element. Several of them are grouped together into blocks/workgroups. For convenience we will use the CUDA language of threads and blocks. From the parallel master-slave idea, one can derive two general approaches for the tour construction on the GPU. Either a thread is assigned to computing the full tour of one ant, or one thread computes only part of the tour and a whole thread block is assigned per ant. Thus we have the one-ant-per-thread and the one-ant-per-block schemes. Many papers implement either the former (Bai et al. [5], You [96], Diego et al. [27]) or the latter (Li et al. [51], Uchida et al. [89]). Only a few publications (Cecilia et al. [14], Delévacq et al. [25]) compare the two. Cecilia et al. argue that the one-thread-per-ant approach is a kind of task parallelization, and that the number of ants for the studied problem size is not enough to fully exploit the GPU hardware. Moreover, they argue that there is divergence within a warp and that each ant has an unpredictable memory access pattern. This motivated them to study the one-block-per-ant approach as well.

Most papers provide a single implementation of their selected approach, often reporting how they use certain GPU specifics such as shared and constant memory.

In contrast, the papers by Cecilia et al. [14], Delévacq et al. [25], and Uchida et al. [89] study different implementations of at least one of the approaches. For the one-ant-per-thread scheme, Cecilia et al. [14] examine the effects of separating the computation of the probability for each city from the tour construction. They also introduce a list of nearest neighbors that have to be visited first, in order to reduce the amount of random numbers. The effects of shared memory and texture memory usage are studied. Delévacq et al. also examine the effects of using shared memory or not. Moreover, they study the addition of a local search step to improve each ants solution. Uchida et al. [89] examine different approaches of city selection in the tour construction step in order to reduce the amount of probability summations.

As the pheromone update step is often less time consuming than the tour construction step, not all papers put it on the GPU. Most of the ones that do, investigate only a single pheromone update approach. In contrast, Cecilia et al. [14] propose different pheromone update schemes, and investigate different implementations of those schemes.

An additional parallelization concept developed already in the pre-GPU literature is multi-colony ACO. Here, several colonies independently explore the search space using their own pheromone matrices. The colonies can cooperate by periodically exchanging information [71]. On a single GPU this approach can be realized by assigning one colony per block, as done by Bai et al. in [5] and by Delévacq et al. in [25]. If several GPUs are available, one can of course use one GPU per colony as studied by Delévacq et al. in [25].

Both Catala et al. [13] and Cecilia et al. [14] provide information about the CPU implementation used for computing the achieved speedups, see Table 1. Catala et al. compare their implementations against the GRID-ACO-OP algorithm [67] running on a grid of up to 32 Pentium IV.

From the above description, we observe that for the ACO, the task most commonly executed on the GPU is tour construction. The papers of Cecilia et al. [14] and Delévacq et al. [25] indicate that the one-ant-per-block scheme seems to be superior to the one-ant-per-thread scheme.

2.3 Population Based Metaheuristics

By population based metaheuristics we understand methods that maintain and evolve a population of solutions, in contrast with trajectory (or single solution) based metaheuristics that are typically based on local search. In this subsection we will focus on evolutionary algorithms. For a discussion of swarm intelligence methods on the GPU we refer to the Section 2.2 above.

In Evolutionary Algorithms, a population of solutions evolves over time, yielding a sequence of generations. A new population is created from the old one by using a process of reproduction and selection, where the former is often done by crossover and/or mutation and the latter decides which individuals form the next generation. A crossover operator combines the features of two parent solutions in order to create children. Mutation operators simply change (mutate) one solution. The idea is that, analogous to natural evolution, the quality of the solutions in the population will increase over time. Evolutionary algorithms provide clear parallelism. The computation of offspring can be performed with at most two

Author	Year	Problem	Algorithm	Operators
Li et al. [53]	2009	TSP	IEA	PMX, mutation
Chen et al. [17]	2011	TSP	GA	crossover, 2-opt mutation
Fujimoto and Tsutsui [32]	2011	TSP	GA	OX, 2-opt local search
Zhao et al. [98]	2011	TSP	IEA	gene string move, multi bit exchange

Author	Selection	
	Immune	next Population
Li et al. [53]	Better	Tournament
Chen et al. [17]		Best
Fujimoto and Tsutsui [32]		Best
Zhao et al. [98]	Best position	Tournament

Author	GPU(s)	Max. Speedup	CPU code
Li et al. [53]	GeForce 9600 GT	11.5	?
Chen et al. [17]	Tesla C2050	1.7	?
Fujimoto and Tsutsui [32]	GeForce GTX 285	24.2	?
Zhao et al. [98]	GeForce GTS 250	7.5	?

Table 2: Overview of EA GPU implementations on the GPU for routing. Legend: IEA – immune EA; GA – genetic algorithm; PMX – partially mapped crossover; OX – order crossover; Better – best of vaccinated and not vaccinated tour; Best position – vaccine creates set of solutions, best is chosen; Best – best of parent and child; ? – unknown CPU code.

individuals (the parents). Moreover, the crossover operators might be parallelizable. Either way, enough individuals are needed to fully saturate the GPU, but at the same time all of them have to make a contribution to increasing the solution quality (see e.g. Fujimoto and Tsutsui [32]).

In our literature search, we found publications on Evolutionary Algorithms (EA) and Genetic Algorithms (GA) (25), Genetic Programming (12), and Differential Evolution (3) within this category. For combinations of EA/GA with LS, and memetic algorithms, see Section 3.4 below.

Although the literature is rich on GPU implementations of population based metaheuristics, only a few publications discuss routing problems. The ones we found are all presented in Table 2. They use either a genetic algorithm, or an immune evolutionary algorithm which combines concepts from immune systems[1] with evolutionary algorithms. All the papers we have found in this category use CUDA.

In some of the GPU implementations, the crossover operator is completely removed to avoid binary operations and yield totally independent individuals. In the routing related GPU literature the apparent parallelism has lead to the two parallelization schemes of assigning one individual to one thread (coarse grained parallelism) (Chen et al. [17]), and one individual to one block (fine grained parallelism) (Li et al. [53], Fujimoto and Tsutsui [32]), see also Table 3. In some papers, different parallelization schemes are used for different operators. We have seen no paper that directly compares both schemes for the same operation.

[1] Artificial Immune Systems (AIS) is a sub-field of Biologically-inspired computing. AIS is inspired by the principles and processes of the vertebrate immune system.

Author	Crossover		Mutation		Vaccination		Tour evaluation	
	T	B	T	B	T	B	T	B
Li et al. [53]								x
Chen et al. [17]	x		x					
Fujimoto and Tsutsui [32]		x		x				x
Zhao et al. [98]			x			U		

Table 3: Studied implementation approaches with respect to whether one individual is assigned to one thread or block. Legend: T – thread; B – block; U – uncertain.

The scheme chosen obviously influences the efficiency and quality of the GPU implementation. On the one hand a minimum number of individuals is needed to fully saturate all of the computational units of the GPU, especially with the one-individual-per-thread scheme. On the other hand, from an optimization point of view, it might not increase the quality of the algorithm to have a huge population size (Fujimoto and Tsutsui [32]). Analogously, the one-individual-per-block scheme only makes sense if the underlying operation can be distributed over the threads of a block.

Most of the papers describe their approach with details on the implementation. In [98], Zhao et al. also compare with the results of four other papers [2, 50, 51, 53]. They report that their own implementation has the shortest GPU running time, but interestingly the speedup compared to unknown CPU implementations is highest for [53].

2.4 Local Search and Trajectory-based Metaheuristics

Local Search (LS, neighborhood search), see for instance [1], is a basic algorithm in discrete optimization and trajectory-based metaheuristics. It is the computational bottleneck of single solution based metaheuristics such as Tabu Search, Guided Local Search, Variable Neighborhood Search, Iterated Local Search, and Large Neighborhood Search. Given a current solution, the idea in LS is to generate a set of solutions — the neighborhood — by applying an operator that modifies the current solution. The best (or, alternatively, an improving) solution is selected, and the procedure continues until there is no improving neighbor, i.e., the current solution is a local optimum. An LS example is described in Part I [9].

The evaluation of constraints and objective components for each solution in the neighborhood is an embarrassingly parallel task, see for instance [65] and [9] for an illustrating example. Given a large enough neighborhood, an almost linear speedup of neighborhood exploration in LS is attainable. The massive parallelism in modern accelerators such as the GPU seems well-suited for neighborhood exploration. This has naturally lead to several research papers implementing local search variations on the GPU, reporting speedups of one order of magnitude when compared to a CPU implementation of the same algorithm. Profiling and fine-tuning the GPU implementation may ensure good utilization of the GPU. Schulz [82] reports a speedup of up to one order of magnitude compared to a naive GPU implementation. To fully saturate the GPU, the neighborhood size is critical; it must be large enough [82]. The effort of evaluating all neighbors can be exploited

more efficiently than by just applying one move. In [12], a set of improving and independent moves is determined heuristically and applied simultaneously, reducing the number of neighborhood evaluations needed.

We would have liked to present clear guidelines for implementing LS on the GPU based on the observed literature. Due to the richness of applications, problems, and variations of LS, this is not possible. Instead, we shall discuss approaches taken in papers that study routing problems.

Although the term originates from Genetic Algorithms, we will use the term *fitness structure* for the collection of delta values (see Section 5 in [9]) and feasibility information for all neighbors of the current solution. Table 4 provides an overview of the routing related GPU papers using some kind of local search. The earliest by Janiak et al. [40] utilizes the graphics pipeline for tabu search by providing a fragment shader that evaluates the whole neighborhood in a one fragment per move fashion. The remaining steps of the search were performed on the CPU.

With the availability of CUDA, the number of papers studying LS and LS-based metaheuristics on the GPU increased. The technical report by Luong et al. [57] discusses a CUDA based GPU implementation of LS. To the authors' best knowledge, this is the first report of a GPU implementation of pure LS. Further research is discussed in two follow-up papers [62,61]. The authors apply LS to different instances of well-known DOPs such as the Quadratic Assignment Problem and the TSP. We will concentrate on their results for routing related problems, i.e., the TSP.

2.4.1 Local Search on the GPU

Thanks to the flexibility and ease of programming of CUDA, more steps of the LS process can be executed on the GPU. Table 5 provides an overview of what steps are done on the GPU in which routing related publication. Table 6 shows the CPU-GPU copy operations involved. Broadly speaking, LS consists of neighborhood generation, evaluation, neighbor/move selection, and solution update. The first task can be done in several ways. A simple solution is to generate the neighborhood on the CPU and copy it to the GPU on each iteration. Alternatively, one may create the neighborhoods directly on the GPU. The former approach, taken by Luong et al. in [62], involves copying a lot of information from the CPU to the GPU on each iteration. The neighborhood is normally represented as a set of moves, i.e., specific changes to the current solution. If one thread on the GPU is responsible for the evaluation of one or several moves, a mapping between moves and threads can be provided. This mapping can either be an explicit formula (Luong et al. [62], Burke and Riise [12], Coelho et al. [20], Rocki and Suda [77], Schulz [82]) or an algorithm (Luong et al. [62]). Alternatively, it can be a pre-generated explicit mapping that lies in the GPU memory as investigated by Janiak et al. [40], and Schulz [82]. The advantage of the mapping approach is that there is no need for copying any information to the GPU on each iteration. The pre-generated mapping only needs to be copied to the GPU once before the LS process starts.

The neighborhood evaluation is the most computationally intensive task in LS-based algorithms. Hence, all papers perform this task on the GPU. In contrast, selecting the best move is not always done on the GPU. A clear consequence of CPU based move selection is the necessity of copying the fitness structure to the CPU on each iteration. GPU based move selection eliminates this data transfer,

Author	Year	Problem	Algorithm	Neighborhood
Janiak et al. [40]	2008	TSP	TS	2-exchange (swap)
Luong et al. [62]	2011	TSP	LS	2-exchange (swap)
O'Neil et al. [70]	2011	TSP	MS-LS	2-opt
Burke and Riise [12]	2012	TSP	ILS	VND: 2-opt + relocate
Coelho et al. [20]	2012	SVRPDSP	VNS	swap + relocate
Rocki and Suda [77]	2012	TSP	(LS)	2-opt, 3-opt
Schulz [82]	2013	DCVRP	LS	2-opt, 3-opt

Author	Approach
Janiak et al. [40]	Graphics pipeline: move evaluation by fragment shader
Luong et al. [62]	CUDA
O'Neil et al. [70]	CUDA: multiple-ls-per-thread, load balancing
Burke and Riise [12]	CUDA: one-move-per-thread, applies several independent moves at once
Coelho et al. [20]	CUDA: one-move-per-thread
Rocki and Suda [77]	CUDA: several-moves-per-thread
Schulz [82]	CUDA: one-move-per-thread, asynchronous execution, very large nbhs

Author	GPU(s)	Max. Speedup	CPU code
Janiak et al. [40]	GeForce 8600 GT	1.12	C#
Luong et al. [62]	a.o. Tesla M2050	19.9	?
O'Neil et al. [70]	Tesla C2050	61.9	single core
Burke and Riise [12]	GeForce GTX 280	70×7.5	?
Coelho et al. [20]	Geforce GTX 560 Ti	17	own
Rocki and Suda [77]	a.o. Geforce GTX 680	27	32 cores
Schulz [82]	GeForce GTX 480		

Table 4: Overview of LS-based GPU literature on routing. Legend: TS - tabu search; LS - local search; MS-LS - multi start local search; ILS - Iterated Local Search; VND - variable neighborhood descent; VNS - variable neighborhood search; SVRPDSP - single vehicle routing problem with deliveries and selective pickups; a.o. - amongst others (only maximum speedup is mentioned); (LS) - Rocki and Suda consider only the neighborhood evaluation part.

Author	Nbh gen.	Nbh eval.	Neighbor sel.	Sol. update
Janiak et al. [40]	i	x		
Luong et al. [62]	-/i	x	-/x	
O'Neil et al. [70]	i	x	x	x
Burke and Riise [12]	i	x		
Coelho et al. [20]	i	x		x
Schulz [82]	i	x	x	x

Table 5: Tasks performed on the GPU during one iteration. Legend: Nbh – neighborhood; gen. – generation; eval. – evaluation; sel. – selection; Sol – solution; i – neighborhood generation is done implicitly (use of some nbh description); -/x – done in some settings; -/i – done in some settings.

but an efficient selection algorithm needs to be in place on the GPU. A clear example is simple steepest descent, where the best move can be computed by a standard reduction operation. A tabu search can also be implemented on the GPU by first checking for each move whether it is tabu, and then reducing to the best non-tabu move. In general, it may not be clear which approach will perform better; it depends on the situation at hand. In such cases, the alternative implementations

Author	Once		In each iteration			
	Prob. desc.	Nbh. desc.	Sol.	Nbh.	FS	Sel. move
Janiak et al. [40]	↑	↑	↑		↓	
Luong et al. [62]	↑		↑	-/↑	-/↓	-/↓
O'Neil et al. [70]	↑					
Burke and Riise [12]	↑		↑		s↓	
Coelho et al. [20]	↑				↓	↑
Schulz [82]	↑	↑				↓

Table 6: Data copied from and to GPU. Legend: Prob. – problem; Nbh. – neighborhood; desc. – description; Sol. – solution; FS – fitness structure; Sel. – selected; ↑ – upload to GPU from CPU; ↓ – download from GPU to CPU; s↓ – subset of fitness structure is downloaded from GPU; -/↑ or -/↓ – copied in some settings.

must be compared. All routing related papers we found use either one or the other approach for a given algorithm, see Table 5. Luong et al. [62] compare them for hill climbing on the permuted perceptron problem.

If move selection is performed on the GPU, the update of the current solution may also be performed on the device. This eliminates the otherwise necessary copying of the updated current solution from the CPU to the GPU. Alternatively, the chosen move can be copied to the GPU (Coelho et al. [20]).

2.4.2 Efficiency aspects and limitations

In CUDA it is not possible to synchronize between blocks inside a kernel. Since most papers employ a one-move-per-thread approach, the LS process needs to be implemented using several kernels. In combination with the different copy operations that might be needed, the question of asynchronous execution becomes important. By using streams in combination with asynchronous CPU - GPU coordination, it is possible to reduce the time where the GPU is idle, even to zero. Only the paper by Schulz [82] proposes and investigates an asynchronous execution pattern.

The efficiency of a kernel is obviously important for the overall speed of the computation. The papers (Luong et al. [62], O'Neil et al. [70], Coelho et al. [20], Rocki and Suda [77], Schulz [82]) all discuss some implementation details and CUDA specific optimizations. Only Schulz [82] provides a profiling analysis of the presented details.

So far we have assumed that the GPU memory is large enough to store all necessary information such as problem data, the current solution, and the fitness structure. For very large neighborhoods the fitness structure might not fit into GPU memory. Luong et al. mention this problem in [62]. They seem to solve it by assigning several moves to one thread. Schulz [82] provides an implementation for very large neighborhoods by splitting the neighborhood in parts.

When evaluating the whole neighborhood one naturally selects a single, best improving move. However, as observed by Burke and Riise in [12], one may waste a lot of computational effort. They suggest an alternative strategy where ones finds independent improving moves and applies them all. This reduces the amount of iterations needed to find a local optimum.

2.4.3 Multi-start Local Search

Pure local search is guaranteed to get stuck in a local optimum, given sufficient time. Amongst alternative remedies, multi-start LS is maybe the simplest. New initial solutions may be generated randomly, or with management of diversity. Multi-start LS thus provides another degree of parallelism, where one local search instance is independent of the other. In the GPU literature we have found two main approaches. Either, a GPU based parallel neighborhood evaluation of the different local searches is performed sequentially (Luong et al. [61]), or, the local searches run in parallel on the GPU (O'Neil et al. [70,99], Luong et al. [61]).

For approaches where there is no need for data transfer between the CPU and GPU during LS, the former scheme should be able to keep the GPU fully occupied with neighborhood evaluation. However, LS might use a complicated selection procedure that is more efficient to execute on the CPU, despite the necessary copy of fitness structure. In this case one could argue that using sequential parallel neighborhood evaluation will lead to too many CPU-GPU copy operations, slowing down the overall algorithm. However, this is not necessarily true. If the copying of data takes less time than neighborhood evaluation, asynchronous execution might be able to fully hide the data transfer. In one iteration, while the fitness structure of the i-th local search is copied to the CPU, the GPU can already evaluate the neighborhood for the next, j-th local search where $j = i + 1$. Once the copying is finished, the CPU can then perform move selection for the i-th local search, all while the GPU is still evaluating the neighborhood of the j-th local search.

The second idea of using one thread per LS instance also has its drawbacks. First, for the GPU to be fully utilized, thousands of threads are needed. This raises the question, whether, from a solution quality point of view, it makes sense to have that many local searches. On the GPU, all threads in a warp perform exactly the same operation at any time. Hence, all local searches in a warp must use the same type of neighborhood. Moreover, different local searches in a warp might have widely varying numbers of iterations until they reach a local optimum. If all threads in the same warp simply run their local search to the end, they have to 'wait' until the last of their local searches is finished before the warp can be destroyed.

There are ways to tackle these problems. In [70] O'Neil et al. use the same neighborhood for all local searches and employ a kind of load balancing to avoid threads within a warp waiting for the others to complete. Another idea, used e.g. in (Zhu et al. [99], Luong et al. [61]), is to let the LS in each thread run only for a given number of iterations and then perform restart or load balancing before continuing. Due to the many variables involved, it is impossible to state generally that the sequential parallel neighborhood evaluation is better or worse than the one thread per local search approach. Even for a given situation, such a statement needs to be based on implementations that have been thoroughly optimized, analyzed and profiled, so that the advantages and limitations of each approach become apparent. We have not found a paper that provides such a thorough comparison between the two approaches.

2.5 GPU Computing for Shortest Path Problems

Already in 2004, Micikevicius [66] describes his graphics pipeline GPU implementation of the Warshall-Floyd algorithm for the all-pairs shortest paths problem. He reports speedups of up to 3x over a CPU implementation. In 2007, Harish and Narayanan [39] utilize CUDA to implement breadth first search, single source shortest path, and all-pairs shortest path algorithms aimed at large graphs. They report speedups, but point out that the size of the device memory limits the size of the graphs handled on a single GPU. Also, the GPU at the time only supported single precision arithmetic. Katz and Kider [41] describe a shared memory cache efficient CUDA implementation to solve transitive closure and the all-pairs shortest-path problem on directed graphs for large datasets. They report good speedups both on synthetic and real data. In contrast with the implementation of Harish and Narayanan, the graph size is not limited by the device memory.

Buluç, Gilbert, and Budak [10] implemented (CUDA) a recursively partitioned all-pairs shortest-paths algorithm where almost all operations are cast as matrix-matrix multiplications on a semiring. They report that their implementation runs more than two orders of magnitude faster on an NVIDIA 8800 GPU than on an Opteron CPU. The number of vertices in the test graphs used vary between 512 and 8192. The all-pairs SPP was also studied by Tran [88], who utilized CUDA to implement two GPU-based algorithms and reports an incredible speedup factor of 2,500 relative to a single core implementation.

In a recent paper [26], Delling et al. present a novel algorithm called PHAST to solve the nonnegative single-source shortest path problem on road networks and other graphs with low highway dimension. PHAST takes advantage of features of modern CPU architectures, such as SSE and multi-core. According to the authors, the method needs fewer operations, has better locality, and is better able to exploit parallelism at multicore and instruction levels when compared to Dijkstras algorithm. They also implement a GPU version of PHAST (GPHAST) with CUDA, and report up to three orders of magnitude speedup relative to Dijkstras algorithm on a high-end CPU. They conclude that GPHAST enables practical all-pairs shortest-paths calculations for continental-sized road networks.

With robotics applications as main focus, Kider et al. [43] implement a GPU version of R*, a randomized, non-exact version of the A* algorithm called R*GPU. They report that R*GPU consistently produces lower cost solutions, scales better in terms of memory, and runs faster than R*.

3 Literature on Non-routing Problems

Although the specifics of a metaheuristic may change according to the problem at hand, its main idea stays the same. Therefore it is also interesting to study GPU implementations of metaheuristics in a non-routing setting. This is especially true for metaheuristics where so far no routing related GPU implementation exists. In the following, we present a short overview over existing GPU literature for metaheuristics applied to DOPs other than routing problems.

3.1 Swarm Intelligence Metaheuristics (Non-ACO)

Particle Swarm Optimization (PSO) is normally considered to belong to swarm intelligence methods, but may also be regarded as a population based method. Just as GA, PSO may be used both for continuous and discrete optimization problems. An early PSO on GPU paper is Li et al. (2007) [52]. They use the graphics pipeline to fine grained parallelization of PSO, and perform computational experiments on three unconstrained continuous optimization problems. Speedup factors up to 5.7 were observed. In 2011, Solomon et al. [83] report from an implementation of a collaborative multi-swarm PSO algorithm on the GPU for a real-life DOP application: the task matching problem in a heterogeneous distributed computing environment. They report speedups factors up to 37.

Emergent behavior in biology, e.g., flocking birds and schooling fish, was an inspiration for PSO. However, the flocking birds brand is still used for PSO-like swarm intelligence methods in optimization. Charles et al. [16] study flocking-based document clustering on the GPU and report a speedup of 3-5 relative to a CPU implementation. In a 2011 follow-up paper with partly the same authors [23], speedup factors of 30-60 were observed. In an undergraduate honors thesis [91], Weiss investigates GPU implementation of two special purpose swarm intelligence algorithms for data mining: an ant colony optimization algorithm for rule-based classification, and a bird-flocking algorithm for data clustering. He concludes that the GPU implementation provides significant benefits.

3.2 Population Based Metaheuristics

In [97], Yu et al. describe an early (2005) implementation of a fine-grained parallel genetic algorithm for continuous optimization, referring to the 1991 paper by Spiessens and Manderick [85] on massively parallel GA. They were probably the first to design and implement a GA on the GPU, using the graphics pipeline. Their approach stores chromosomes and their fitness values in the GPU texture memory. Using the Cg language for the graphics pipeline, fitness evaluation and genetic operations are implemented entirely with fragment programs (shaders) that are executed on the GPU in parallel. Performance of an NVidia GeForce 6800GT GPU implementation was measured and compared with a sequential AMD Athlon 2500+ CPU implementation. The Colville function in unconstrained global optimization was used as benchmark. For genetic operators, the authors report speedups between 1.4 (population size 32^2) and 20.1 (population size 512^2). Corresponding speedups for fitness evaluation are 0.3 and 17.1, respectively.

Also in 2005, Luo et al. [55] describe their use of the graphics pipeline and the Cg language for a parallel genetic algorithm solver for 3-SAT. They compare performance between two hardware platforms.

Wong et al. [95,94,30] investigate hybrid computing GAs where population evaluation and mutation are performed on the GPU, but the remainder is executed on the CPU. In [93], Wong extends the work to multi-objective GAs and uses CUDA for the implementation. For a recent comprehensive survey on GPU computing for EA and GA but not including Genetic Programming, see Section 1.3.2 of the PhD Thesis of Thé Van Luong [56].

Genetic Programming (GP) is a special application of GA where each individual is a computer program. The overall goal is automatic programming. Early GPU implementations (2007) are described by Chitty [18], who uses the graphics pipeline and Cg. Harding & Banzhaf [38] also use the graphics pipeline but with the Accelerator package, a .Net assembly that provides access to the GPU via DirectX. Several papers involving a subset of Banzhaf, Harding, Harrison, Langdon, and Wilson [36,47,6,48,37] reports from extensions of this initial work. Robilliard et al. have published three papers on GPU-based GP using CUDA [75,76,74], initially a fine-grained parallelization scheme on the G80 GPU, then with different parallelization schemes and better speedups. Maitre, Collet & Lachiche [64] reports from similar work. For details, we refer to the recent survey by Langdon [49] and the individual technical papers.

3.3 Local Search and Trajectory-based Metaheuristics

Luong et al. have published several follow-up papers to [57,62,61]. In [58], they discuss how to implement LS algorithms with large-size neighborhoods on the GPU[2], with focus on memory issues. Their general design is based on socalled iteration-level parallelization, where the CPU manages the sequential LS iterations, and the GPU is dedicated to parallel generation and evaluation of neighborhoods. Mappings between threads and neighbors are proposed for LS operators with Hamming distance 1, 2, and 3. From an experimental study on instances of the Permuted Perceptron Problem from cryptography the authors conclude that speedup increases with increasing neighborhood cardinality (Hamming distance of the operator), and that the GPU enables the use of neighborhood operators with higher cardinality in LS. Similar reports are found in [59] and [60]. The PhD thesis of Thé Van Luong from 2011 [56] contains a general discussion on GPU implementation of metaheuristics, including results from the papers mentioned above.

The paper by Janiak et al. [40] applies Tabu search also to the Permutation Flowshop Scheduling Problem (PFSP) with the Makespan criterion. Their work on the PFSP was continued by Czapinski & Barnes in [24]. They describe a tabu search metaheuristic based on swap moves. The GPU implementation was done with CUDA. Two implementations of move selection and tabu list management were considered. Performance was optimized through experiments and tuning of several implementation parameters. Good speedups were reported, both relative to the GPU implementation of Janiak et al. and relative to a serial CPU implementation, for randomly generated PFSP instances with 10–500 tasks and 5–30 machines. The authors mainly attribute the improved efficiency over Janiak et al. to better memory management.

The first of three publications we have found on GPU implementation of Simulated Annealing (SA) is a conference paper by Choong et al. [19]. SA is the preferred method for optimization of FPGA placement[3]. In [35] Han et al. study SA on the GPU for IC floorplanning by using CUDA. They work with multiple

[2] The title of the paper may suggest that it discusses the Large Neighborhood Search metaheuristic, but this is not the case.

[3] As discussed in Section 2.1 above, FPGAs were used in early works in heterogeneous discrete optimization.

solutions in parallel and evaluate several moves per solution in each iteration. As the GPU based algorithm works differently than the CPU method, Han et al. examine three different modifications to their first GPU implementation with respect to solution quality and speedup. They achieve a speedup of up to 160 for the best solution quality, where the computation times are compared to the CPU code from the UMpack suite of VLSI-CAD tools [3]. Stivala et al. use GPU based SA in [86] for the problem of searching a database for protein structures or occurrences of substructures. They develop a new SA based algorithm for the given problem and provide both a CPU and a GPU implementation[4]. Each thread block in the GPU version runs its own SA schedule, where the threads perform the database comparisons. The quality of the proposed method various with different problems, but good speedups of the GPU version versus the CPU one are obtained.

3.4 Hybrid Metaheuristics

The definition of *hybrid metaheuristics* may seem unclear. In the literature, it often refers to methods where metaheuristics collaborate or are integrated with exact optimization methods from mathematical programming, the latter also known as *matheuristics*. A restricted definition to combinations of different metaheuristics arguably has diminishing interest, as increasing emphasis in the design of modern metaheuristics is put on the combination and extension of relevant working mechanisms of different classical metaheuristics. As regards hybrid methods, the three relevant publications we have found all discuss GPU implementation of combinations of Genetic Algorithms with LS, a basic form of *memetic algorithms*.

In 2006, Luo & Liu [54] follow up on the 2005 graphics pipeline GA paper on the 3-SAT problem by Luo et al. [55] referred to in Section 3.2 above. They develop a modified version of the parallel CGWSAT hybrid of cellular GA and greedy local search due to Folino et al. [31] and implement it on a GPU using the graphics pipeline with Cg. They report good speedups over a CPU implementation with similar solution quality. GPU-based hybrids of GA and LS for Max-SAT was investigated in 2009 by Munawar et al. in [68].

In [44], Krüger et al. present the first implementation of a generic memetic algorithm for continuous optimization problems on a GTX295 gaming card using CUDA. Reportedly, experiments on the Rosenbrock function and a real world problem show speedup factors between 70 and 120.

Luong et al. propose in [63] a load balancing scheme to distribute multiple metaheuristics over both the GPU and the CPU cores simultaneously. They apply the scheme to the quadratic assignment problem using the fast ant metaheuristic, yielding a combined speedup (both multiple cores on CPU and GPU) of up to 15.8 compared to a single core on the CPU.

3.5 GPU Implementation of Linear Programming and Branch & Bound

Also relevant to discrete optimization we found five publications on GPU implementation of linear programming (LP) methods. Greeff [33] published a technical

[4] The CPU version is generated by compiling the kernels for the CPU.

report on a GPU graphics pipeline implementation of the revised simplex method in 2005. Reported speedups were large compared to a CPU implementation. The implementation could not solve problems with more than 200 variables, however.

In their 2008 paper, Jung & O'Leary [69] present a mixed-precision CPU-GPU interior point LP algorithm. By comparing GPU and CPU implementations, they demonstrated performance improvement for sufficiently large dense problems with up to some 700 variables and 500 constraints.

In 2009, Spampinato & Elster [84] published a continuation of the work by Greeff from 2005. Their CUDA implementation of the revised simplex method solves LPs with up to 2000 variables on a CPU/GPU system. They report speedups factors of 2.5 for large problem instances.

Early GPUs had only single precision arithmetic. In 2011, Lalami et al. [46] report a maximum speedup of 12.5 for their simplex method implementation with double precision arithmetic on a GTX 260 GPU. They use randomly generated non-sparse LP instances. Also in 2011, the same authors report from a CUDA implementation of the simplex method on a multi GPU architecture [45]. Computational tests on random, non-sparse instances show a maximum speedup of 24.5 with two Tesla C2050.

Branch & Bound is a widely used exact method for solving DOPs. In [15] the GPU is used for the bound operator in the algorithm applied to the flow shop scheduling problem. The paper discusses GPU specific details of the implementation and in experiments a speedup of up to 77.5 compared to a single core on a CPU is achieved.

4 Lessons for Future Research

In the previous section we presented a literature survey on GPU computing in discrete optimization and a more detailed discussion of selected papers on routing problems. In the following we will provide our views on future research on GPU computing in discrete optimization.

4.1 GPU Implementations in Discrete Optimization

The overwhelming majority of routing related papers on GPU usage in discrete optimization has focused on relatively simple, well-known optimization algorithms on the GPU. A main goal is to compare GPU implementations with equivalent single core CPU versions. The results predominantly show significant speedups and hence provide proofs of concept. The observations are consistent with GPU related research from other parts of scientific computing. Also in optimization, the GPU is a viable and powerful tool that can be used to increase performance. This is not uninteresting, particularly from a pragmatic stance. Also from a scientific point of view, proof of concept papers are important. More power for computational experiments will lead to better algorithms and better understanding of optimization problems.

Is this the final word? Far from it. Most of the relevant literature does not consider important aspects of GPU usage and the development of *novel* algorithms which fully utilize the combined advantages of the CPU and the GPU to provide

faster and more robust solutions. In our opinion, the subfield of GPU computing in discrete optimization is still in its infancy.

For a practitioner it may be of little interest whether the GPU or CPU is used to its full capacity. From a scientific perspective we would like to use scientific methods to develop algorithms which are able to yield better and more robust solutions than the algorithms of today by fully utilizing all available hardware efficiently. To achieve this goal, research that provides knowledge and ideas towards this end is welcome. What qualifies such research, and what is lacking so far?

Focusing on comparing CPU and GPU versions of an algorithm is an important step to provide proof of concept implementations showing the performance potential provided by the GPU. Nevertheless, towards the specified scientific goal of new and efficient algorithms, this approach has several potential drawbacks.

Solution quality: Many of the papers comparing a CPU and a GPU implementation do not discuss solution quality. On the one hand, if the algorithm is the same, it can be expected that the solution quality is too. However, the considered algorithm that is run on the GPU might not be a state-of-the-art CPU based algorithm and thus not be competitive in terms of latest solution quality.

CPU speed: Similar to the point above, the used algorithm might not be cutting edge for the CPU. Hence, even if the GPU implementation is faster than its CPU counterpart, the leading CPU algorithm might still be faster than the studied GPU implementation. In addition, the considered implementation of the algorithm on the CPU might not be state-of-the-art. An efficient GPU implementation requires effort in finding the right memory access patterns, the right distribution of data over the different memories, synchronization and cooperation strategies and much more. An equally optimized CPU implementation would amongst others utilize multiple cores, have caching strategies and use SSE or AVX instructions[5]. Such an effort is rarely seen in the literature.

GPU usage: Although the GPU implementation might perform faster than the CPU implementation, it does not mean it uses the GPU efficiently. There might be a better way to distribute the work over the GPU architecture, a faster memory access pattern, or other improving variants. It might be that the GPU implementation is using the GPU only a fraction of the time, leaving it idle for a substantial part of the time. This means that there could be a different implementation or algorithm for the problem which is able to use the GPU more efficiently, with resulting speed and/or quality improvement.

CPU usage: In most of the papers comparing CPU and GPU implementations, the CPU is basically idle the whole time. This is a waste of computational resources. A truly heterogeneous algorithm will typically have higher performance.

In our opinion, future research papers on GPU usage in discrete optimization should contain algorithm analysis and analysis of hardware utilization. Such analyses will identify areas of further improvement, spawn ideas for novel algorithms,

[5] Modern CPUs support vector operations, enabling simultaneous operations on all elements of those vectors [29]. These so-called SIMD extensions/operations started with MMX on 64byte registers and developed further with SSE (128byte registers) into AVX (256byte registers). For a coarse overview see [92], a more detailed discussion of the operations including examples can be found in [29].

and point to further research directions. Such analyses are time-consuming. Although the potential gain is high[6], one cannot expect that researchers in optimization will follow these steps of research in computational science to their end. We think that initial steps should be mandatory, however.

Analysis of algorithm(s)

This is obviously a wide area that covers mathematical analyses as well as computational experiments. Such analyses may show that a known algorithm, deemed too inefficient on the CPU, can now be used beneficially[7] with the help of the GPU. Another example is the development of new algorithms that use the intrinsic properties of the available hardware (CPU and GPU together) to provide better or more robust solutions. Clearly one focus here would be on the improvement of the solution quality. In general, when studying algorithms on the GPU, one has to make sure that the work done on the GPU is actually beneficial to the algorithm. In LS one could, for example, question the meaning of evaluating billions of moves if just one of them is applied afterwards. Does this really increase the solution quality compared to a simpler first improvement strategy? One could, as suggested by Burke and Riise in [12], utilize several of the improving moves found.

Hardware utilization

Hardware utilization should be analyzed, at least to a basic level, so major bottlenecks are identified and removed. This includes an examination of the CPU-GPU coordination and whether asynchronous execution patterns might be possible and beneficial. An example is found in the paper by Schulz [82], although in general it will not be possible to conduct such a detailed and time-consuming analysis and performance tuning. The analysis and conclusions should be based on solid scientific methods and fair comparison.

Even if it is not possible to perform the final steps of performance optimization, it is important to understand whether an algorithm or implementation is able to use the hardware efficiently. If not, it is equally interesting to discover why this is not the case and what the limiting factors are. This will provide valuable information for the development of other, more efficient algorithms or implementation approaches.

4.2 Heterogeneous Discrete Optimization in general

The lessons learnt from GPU-based algorithms in discrete optimization are in principle also true for heterogeneous discrete optimization. The goal should be algorithms that use all available hardware resources[8] efficiently towards finding

[6] The paper by Schulz [82] indicates an order of magnitude speedup by careful tuning of a basic GPU implementation.

[7] Beneficially here means to improve the overall solution quality, speed or robustness of the overall solution method.

[8] I.e., multiple CPU cores and one or more stream processing accelerators according to the scope of this paper.

high quality solutions. Ideally, such algorithms should be self-adapting and automatically configure themselves to the problem, the hardware, and even to the problem solving status while executing. We think that papers in heterogeneous discrete optimization and similar areas should give a reasonable contribution in the form of knowledge that can be used to create and develop such algorithms. This requires full specification of hardware platforms utilized as well as algorithmic and implementational details.

A promising and virtually unexplored research avenue is the development of collaborative methods in discrete optimization that fully utilize modern, heterogeneous PC architectures. In the next ten years we may see a general performance increase in discrete optimization that surpasses the historical increase pointed to by Bixby [7] for commercial LP solvers.

5 Summary and Conclusion

The sequence of two papers of which this paper is the second, has two primary goals. The first, addressed in Part I [9], is to provide a tutorial style introduction to modern PC architectures and the computational performance increase opportunities that they offer through a combination of parallel cores for task parallelization and one or more stream processing accelerators. The second goal, addressed in Part II here, is to present a survey of the literature relevant to discrete optimization and routing problems in particular.

Part I [9] starts with a short overview of the historical development of CPUs and stream processing accelerators such as the GPU, followed by a brief discussion of the development of more user-friendly GPU programming environments. To illustrate modern GPU programming with CUDA, we provided a concrete example: local search for the TSP. This was followed by the presentation of best practice and state-of-the-art strategies for developing efficient GPU code. We also discussed heterogeneous aspects involved in keeping both the CPU and the GPU busy. Here, in Part II, we provide a comprehensive survey of the existing literature on parallel discrete optimization for modern PC architectures with focus on routing problems. Virtually all related papers report on implementation of an existing optimization algorithm on a stream processing accelerator, mostly the GPU. We provide a critical, detailed review of the literature relevant to routing problems. Finally, we present lessons learnt and our subjective views on future research directions.

GPU computing in discrete optimization is still in its infancy. The bulk of the literature consists of reports from rather basic implementations of existing optimization methods on GPU, with measurement of speedup relative to a CPU implementation of unknown quality. It is our opinion that further research should be performed in a more scientific fashion: with stronger focus on the efficiency of the implementation, proper analyses of algorithms and hardware utilization, thorough and fair measurement of speedup, with efforts to utilize all of the available hardware, and with reports that better enable reproduction. The ultimate goal would be the development of novel, fast and robust high quality methods that exploit the full heterogeneity of modern PCs efficiently while at the same time being flexible by self-adapting to the hardware at hand. The potential gains are hard to over-estimate.

Acknowledgements The work presented in this paper has been partially funded by the Research Council of Norway as a part of the Collab project (contract number 192905/I40, SMARTRANS), the DOMinant II project (contract number 205298/V30, eVita), the Respons project (contract number 187293/I40, SMARTRANS), and the CloudViz project (contract number 201447, VERDIKT).

References

1. Aarts, E., Lenstra, J.K. (eds.): Local Search in Combinatorial Optimization. Princeton University Press (2003)
2. Acan, A.: GAACO: A GA + ACO Hybrid for Faster and Better Search Capability. In: M. Dorigo, G. Di Caro, M. Sampels (eds.) Ant Algorithms, *Lecture Notes in Computer Science*, vol. 2463, pp. 300–301. Springer Berlin / Heidelberg (2002). Proceedings of Third International Workshop, ANTS 2002
3. Adya, S., Markov, I.: Fixed-outline floorplanning: enabling hierarchical design. Very Large Scale Integration (VLSI) Systems, IEEE Transactions on **11**(6), 1120 –1135 (2003)
4. Alba, E.: Parallel Metaheuristics: A New Class of Algorithms. Wiley Series on Parallel and Distributed Computing. John Wiley & Sons (2005)
5. Bai, H., OuYang, D., Li, X., He, L., Yu, H.: MAX-MIN Ant System on GPU with CUDA. In: Fourth International Conference on Innovative Computing, Information and Control (ICICIC), pp. 801 –804 (2009)
6. Banzhaf, W., Harding, S., Langdon, W., Wilson, G.: Accelerating Genetic Programming through Graphics Processing Units. In: R.L. Riolo, T. Soule, B. Worzel (eds.) Genetic Programming Theory and Practice VI, pp. 229–249. Springer (2008)
7. Bixby, R.E.: Solving Real-World Linear Programs: A Decade and More of Progress. Operations Research **50**, 3–15 (2002)
8. Bleiweiss, A.: GPU accelerated pathfinding. In: Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware, GH '08, pp. 65–74. Eurographics Association, Aire-la-Ville, Switzerland, Switzerland (2008)
9. Brodtkorb, A.R., Hagen, T.R., Schulz, C., Hasle, G.: GPU Computing in Discrete Optimization – Part I: Introduction to the GPU. EURO Journal on Transportation and Logistics (2013)
10. Buluç, A., Gilbert, J.R., Budak, C.: Solving path problems on the GPU. Parallel Comput. **36**(5-6), 241–253 (2010)
11. Bura, W., Boryczka, M.: The Parallel Ant Vehicle Navigation System with CUDA Technology. In: P. Jedrzejowicz, N. Nguyen, K. Hoang (eds.) Computational Collective Intelligence. Technologies and Applications, *Lecture Notes in Computer Science*, vol. 6923, pp. 505–514. Springer Berlin / Heidelberg (2011)
12. Burke, E.K., Riise, A.: On Parallel Local Search for Permutations. Submitted
13. Catala, A., Jaen, J., Modioli, J.: Strategies for accelerating ant colony optimization algorithms on graphical processing units. In: 2007 IEEE Congress on Evolutionary Computation (CEC 2007), pp. 492 –500 (2007)
14. Cecilia, J., Garcia, J., Ujaldon, M., Nisbet, A., Amos, M.: Parallelization strategies for ant colony optimisation on GPUs. In: Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on, pp. 339 –346 (2011)
15. Chakroun, I., Mezmaz, M., Melab, N., Bendjoudi, A.: Reducing thread divergence in a GPU-accelerated branch-and-bound algorithm. Concurrency and Computation: Practice and Experience (2012)
16. Charles, J., Potok, T., Patton, R., Cui, X.: Flocking-based Document Clustering on the Graphics Processing Unit. In: N. Krasnogor, G. Nicosia, M. Pavone, D. Pelta (eds.) Nature Inspired Cooperative Strategies for Optimization (NICSO 2007), *Studies in Computational Intelligence*, vol. 129, pp. 27–37. Springer Berlin / Heidelberg (2008)
17. Chen, S., Davis, S., Jiang, H., Novobilski, A.: CUDA-Based Genetic Algorithm on Traveling Salesman Problem. In: R. Lee (ed.) Computer and Information Science 2011, *Studies in Computational Intelligence*, vol. 364, pp. 241–252. Springer Berlin / Heidelberg (2011)
18. Chitty, D.M.: A data parallel approach to genetic programming using programmable graphics hardware. In: D. Thierens, H.G. Beyer, J. Bongard, J. Branke, J.A. Clark, D. Cliff, C.B. Congdon, K. Deb, B. Doerr, T. Kovacs, S. Kumar, J.F. Miller, J. Moore, F. Neumann, M. Pelikan, R. Poli, K. Sastry, K.O. Stanley, T. Stutzle, R.A. Watson, I. Wegener

(eds.) GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation, vol. 2, pp. 1566–1573. ACM Press, London (2007)
19. Choong, A., Beidas, R., Zhu, J.: Parallelizing Simulated Annealing-Based Placement Using GPGPU. In: Field Programmable Logic and Applications (FPL), 2010 International Conference on, pp. 31 –34 (2010)
20. Coelho, I., Ochi, L., Munhoz, P., Souza, M., Farias, R., Bentes, C.: The Single Vehicle Routing Problem with Deliveries and Selective Pickups in a CPU-GPU Heterogeneous Environment. In: 2012 IEEE 14th International Conference on High Performance Computing and Communication, pp. 1606 –1611 (2012)
21. Crainic, T.G.: Parallel Solution Methods for Vehicle Routing Problems. In: B. Golden, S. Raghavan, E. Wasil, R. Sharda, S. Voss (eds.) The Vehicle Routing Problem: Latest Advances and New Challenges, *Operations Research/Computer Science Interfaces Series*, vol. 43, pp. 171–198. Springer US (2008)
22. Crainic, T.G., Le Cun, B., Roucairol, C.: Parallel Branch-and-Bound Algorithms, pp. 1–28. John Wiley & Sons, Inc. (2006)
23. Cui, X., St. Charles, J., Beaver, J., Potok, T.: The GPU Enhanced Parallel Computing for Large Scale Data Clustering. In: Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC), 2011 International Conference on, pp. 220–225 (2011)
24. Czapiński, M., Barnes, S.: Tabu Search with two approaches to parallel flowshop evaluation on CUDA platform. Journal of Parallel and Distributed Computing **71**, 802–811 (2011)
25. 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). Metaheuristics on GPUs
26. Delling, D., Goldberg, A.V., Nowatzyk, A., Werneck, R.F.: PHAST: Hardware-Accelerated Shortest Path Trees. In: Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium, IPDPS '11, pp. 921–931. IEEE Computer Society, Washington, DC, USA (2011)
27. Diego, F.J., Gómez, E.M., Ortega-Mier, M., García-Sánchez, A.: Parallel CUDA Architecture for Solving de VRP with ACO. In: S.P. Sethi, M. Bogataj, L. Ros-McDonnell (eds.) Industrial Engineering: Innovative Networks, pp. 385–393. Springer London (2012)
28. Dorigo, M., Stützle, T.: Ant Colony Optimization. Bradford Company, Scituate, MA, USA (2004)
29. Fog, A.: Optimizing software in C++ – An optimization guide for Windows, Linux and Mac platforms. Copenhagen University College of Engineering. http://www.agner.org/optimize
30. Fok, K.L., Wong, T.T., Wong, M.L.: Evolutionary Computing on Consumer Graphics Hardware. Intelligent Systems, IEEE **22**(2), 69 –78 (2007)
31. Folino, G., Pizzuti, C., Spezzano, G.: Solving the Satisfiability Problem by a Parallel Cellular Genetic Algorithm. In: Proc. of Euromicro workshop on computational intelligence, IEEE Computer, pp. 715–722. Society Press (1998)
32. Fujimoto, N., Tsutsui, S.: A Highly-Parallel TSP Solver for a GPU Computing Platform. In: I. Dimov, S. Dimova, N. Kolkovska (eds.) Numerical Methods and Applications, *Lecture Notes in Computer Science*, vol. 6046, pp. 264–271. Springer Berlin / Heidelberg (2011)
33. Greeff, G.: The revised simplex algorithm on a GPU. Tech. rep., University of Stellenbosch (2005)
34. Guntsch, M., Middendorf, M., Scheuermann, B., Diessel, O., ElGindy, H., Schmeck, H., So, K.: Population based ant colony optimization on FPGA. In: Field-Programmable Technology, 2002. (FPT). Proceedings. 2002 IEEE International Conference on, pp. 125 – 132 (2002)
35. Han, Y., Roy, S., Chakraborty, K.: Optimizing simulated annealing on GPU: A case study with IC floorplanning. In: Quality Electronic Design (ISQED), 2011 12th International Symposium on, pp. 1 –7 (2011)
36. Harding, S., Banzhaf, W.: Fast genetic programming on GPUs. In: M. Ebner, M. O'Neill, A. Ekárt, L. Vanneschi, A.I. Esparcia-Alcázar (eds.) Proceedings of the 10th European Conference on Genetic Programming, *Lecture Notes in Computer Science*, vol. 4445, pp. 90–101. Springer, Valencia, Spain (2007)
37. Harding, S., Banzhaf, W.: Implementing cartesian genetic programming classifiers on graphics processing units using GPU.NET. In: S. Harding, W.B. Langdon, M.L. Wong, G. Wilson, T. Lewis (eds.) GECCO 2011 Computational intelligence on consumer games and graphics hardware CIGPU, pp. 463–470. ACM (2011)
38. Harding, S.L., Banzhaf, W.: Fast Genetic Programming and Artificial Developmental Systems on GPUs. In: 21st International Symposium on High Performance Computing Systems and Applications (HPCS'07), p. 2. IEEE Computer Society, Canada (2007)

39. Harish, P., Narayanan, P.J.: Accelerating large graph algorithms on the GPU using CUDA. In: Proceedings of the 14th international conference on High performance computing, HiPC'07, pp. 197–208. Springer-Verlag, Berlin, Heidelberg (2007)
40. Janiak, A., Janiak, W., Lichtenstein, M.: Tabu Search on GPU. Journal of Universal Computer Science **14**(14), 2416–2427 (2008)
41. Katz, G.J., Kider Jr, J.T.: All-pairs shortest-paths for large graphs on the GPU. In: Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware, GH '08, pp. 47–55. Eurographics Association, Aire-la-Ville, Switzerland, Switzerland (2008)
42. Kennedy, J., Kennedy, J., Eberhart, R., Shi, Y.: Swarm intelligence. The Morgan Kaufmann series in evolutionary computation. Morgan Kaufmann Publishers (2001)
43. Kider, J., Henderson, M., Likhachev, M., Safonova, A.: High-dimensional planning on the GPU. In: Robotics and Automation (ICRA), 2010 IEEE International Conference on, pp. 2515–2522 (2010)
44. Krüger, F., Maitre, O., Jimenez, S., Baumes, L., Collet, P.: Speedups between x70 and x120 for a generic local search (memetic) algorithm on a single GPGPU chip. In: C. Di Chio, S. Cagnoni, C. Cotta, M. Ebner, A. Ekárt, A. Esparcia-Alcazar, C.K. Goh, J. Merelo, F. Neri, M. Preuß, J. Togelius, G. Yannakakis (eds.) EvoNum 2010, *LNCS*, vol. 6024, pp. 501–511. Springer Berlin / Heidelberg (2010)
45. Lalami, M., El-Baz, D., Boyer, V.: Multi GPU Implementation of the Simplex Algorithm. In: High Performance Computing and Communications (HPCC), 2011 IEEE 13th International Conference on, pp. 179 –186 (2011)
46. Lalami, M.E., Boyer, V., El-Baz, D.: Efficient Implementation of the Simplex Method on a CPU-GPU System. In: Proceedings of the 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum, IPDPSW '11, pp. 1999–2006. IEEE Computer Society, Washington, DC, USA (2011)
47. Langdon, W., Banzhaf, W.: A SIMD interpreter for Genetic Programming on GPU Graphics Cards. In: M. O'Neill, L. Vanneschi, S. Gustafson, A.I. Esparcia Alcazar, I. De Falco, A. Della Cioppa, E. Tarantino (eds.) Proceedings of the 11th European Conference on Genetic Programming, EuroGP 2008, pp. 73–85. Springer (2007)
48. Langdon, W., Harrison, A.: GP on SPMD parallel Graphics Hardware for mega Bioinformatics Data Mining. Soft Computing A Fusion of Foundations Methodologies and Applications **12**(12), 1169–1183 (2008)
49. Langdon, W.B.: Graphics Processing Units and Genetic Programming: An overview. Soft Computing **15**, 1657–1669 (2011)
50. Li, J., Chi, Z., Wan, D.: Parallel genetic algorithm based on fine-grained model with GPU-accelerated. Control and Decision **23**(6) (2008)
51. Li, J., Hu, X., Pang, Z., Qian, K.: A Parallel Ant Colony Optimization Algorithm Based on Fine-grained Model with GPU-acceleration. International Journal of Innovative Computing, Information and Control **5**(11(A)), 3707–3716 (2009)
52. Li, J., Wan, D., Chi, Z., Hu, X.: An Efficient Fine-Grained Parallel Particle Swarm Optimization Method Based On GPU-Acceleration. International Journal of Innovative Computing Information and Control **3**(6), 1707–1714 (2007)
53. Li, J., Zhang, L., Liu, L.: A Parallel Immune Algorithm Based on Fine-Grained Model with GPU-Acceleration. In: Proceedings of the 2009 Fourth International Conference on Innovative Computing, Information and Control, ICICIC '09, pp. 683–686. IEEE Computer Society, Washington, DC, USA (2009)
54. Luo, Z., Liu, H.: Cellular Genetic Algorithms and Local Search for 3-SAT problem on Graphic Hardware. In: Evolutionary Computation, 2006. CEC 2006. IEEE Congress on, pp. 2988 –2992 (2006)
55. Luo, Z., Yang, Z., Liu, H., Lv, W.: GA Computation of 3-SAT problem on Graphic Process Unit. Environment **1**, 7–11 (2005)
56. Luong, T.V.: Métaheuristiques parallèles sur GPU. Ph.D. thesis, Université des Sciences et Technologie de Lille - Lille I (2011). This thesis is written in English
57. Luong, T.V., Melab, N., Talbi, E.G.: Parallel Local Search on GPU. Rapport de recherche RR-6915, INRIA (2009)
58. Luong, T.V., Melab, N., Talbi, E.G.: Large neighborhood local search optimization on graphics processing units. In: Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on, pp. 1 –8 (2010)
59. Luong, T.V., Melab, N., Talbi, E.G.: Local Search Algorithms on Graphics Processing Units. A Case Study: The Permutation Perceptron Problem. In: EvoCOP, pp. 264–275 (2010)

60. Luong, T.V., Melab, N., Talbi, E.G.: Neighborhood Structures for GPU-Based Local Search Algorithms. Parallel Processing Letters **20**(4), 307–324 (2010)
61. Luong, T.V., Melab, N., Talbi, E.G.: GPU-Based Multi-start Local Search Algorithms. In: C. Coello (ed.) Learning and Intelligent Optimization, *Lecture Notes in Computer Science*, vol. 6683, pp. 321–335. Springer Berlin / Heidelberg (2011)
62. Luong, T.V., Melab, N., Talbi, E.G.: GPU Computing for Parallel Local Search Metaheuristic Algorithms. IEEE Transactions on Computers **99**(PrePrints) (2011). DOI http://doi.ieeecomputersociety.org/10.1109/TC.2011.206
63. Luong, T.V., Taillard, E., Melab, N., Talbi, E.G.: Parallelization Strategies for Hybrid Metaheuristics Using a Single GPU and Multi-core Resources. In: C. Coello, V. Cutello, K. Deb, S. Forrest, G. Nicosia, M. Pavone (eds.) Parallel Problem Solving from Nature - PPSN XII, *Lecture Notes in Computer Science*, vol. 7492, pp. 368–377. Springer Berlin Heidelberg (2012)
64. Maitre, O., Lachiche, N., Collet, P.: Fast evaluation of GP trees on GPGPU by optimizing hardware scheduling. In: Proceedings of the 13th European conference on Genetic Programming, EuroGP'10, pp. 301–312. Springer-Verlag, Berlin, Heidelberg (2010)
65. Melab, N., Talbi, E.G., Cahon, S., Alba, E., Luque, G.: Parallel Metaheuristics: Models and Frameworks. In: E.G. Talbi (ed.) Parallel Combinatorial Optimization, pp. 149–161. John Wiley & Sons, Inc. (2006)
66. Micikevicius, P.: General Parallel Computation on Commodity Graphics Hardware: Case Study with the All-Pairs Shortest Paths Problem. In: PDPTA, pp. 1359–1365 (2004)
67. Mocholí, J., Jaén, J., Canós, J.: A grid ant colony algorithm for the orienteering problem. In: The 2005 IEEE Congress on Evolutionary Computation, vol. 1, pp. 942 –949 (2005)
68. Munawar, A., Wahib, M., Munetomo, M., Akama, K.: Hybrid of genetic algorithm and local search to solve MAX-SAT problem using nVidia CUDA framework. Genetic Programming and Evolvable Machines pp. 391–415 (2009)
69. O'Leary, D.P., Jung, J.H.: Implementing an interior point method for linear programs on a CPU-GPU system. Electronic Transactions on Numerical Analysis **28**, 174–189 (2008)
70. O'Neil, M.A., Tamir, D., Burtscher, M.: A Parallel GPU Version of the Traveling Salesman Problem. URL http://www.gpucomputing.net/?q=node/12874. Presentation at 'PDPTA'11 - The 2011 International Conference on Parallel and Distributed Processing Techniques and Applications'
71. Pedemonte, M., Nesmachnow, S., Cancela, H.: A survey on parallel ant colony optimization. Applied Soft Computing **11**(8), 5181 – 5197 (2011)
72. Ralphs, T.K.: Parallel Branch and Cut. In: E. Talbi (ed.) Parallel Combinatorial Optimization, pp. 53–101. Wiley, New York (2006)
73. Ralphs, T.K., Ladányi, L., Saltzman, M.J.: Parallel Branch, Cut, and Price for Large-Scale Discrete Optimization. Mathematical Programming **98**, 253–280 (2003)
74. Robilliard, D., Marion, V., Fonlupt, C.: High performance genetic programming on GPU. In: Proceedings of the 2009 workshop on Bio-inspired algorithms for distributed systems, BADS '09, pp. 85–94. ACM, New York, NY, USA (2009)
75. Robilliard, D., Marion-Poty, V., Fonlupt, C.: Population parallel GP on the G80 GPU. In: Proceedings of the 11th European conference on Genetic programming, EuroGP'08, pp. 98–109. Springer-Verlag, Berlin, Heidelberg (2008)
76. Robilliard, D., Marion-Poty, V., Fonlupt, C.: Genetic programming on graphics processing units. Genetic Programming and Evolvable Machines **10**(4), 447–471 (2009)
77. Rocki, K., Suda, R.: Accelerating 2-opt and 3-opt local search using GPU in the travelling salesman problem. In: 2012 International Conference on High Performance Computing and Simulation (HPCS), pp. 489 –495 (2012)
78. Rostrup, S., Srivastava, S., Singhal, K.: Fast and Memory-Efficient Minimum Spanning Tree on the GPU. In: 2nd International Workshop on GPUs and Scientific Applications (GPUScA 2011). Inderscience (2011)
79. Sanci, S., Isler, V.: A Parallel Algorithm for UAV Flight Route Planning on GPU. International Journal of Parallel Programming **39**(6), 809–837 (2011)
80. Scheuermann, B., Janson, S., Middendorf, M.: Hardware-oriented ant colony optimization. Journal of Systems Architecture **53**(7), 386 – 402 (2007)
81. Scheuermann, B., So, K., Guntsch, M., Middendorf, M., Diessel, O., ElGindy, H., Schmeck, H.: FPGA implementation of population-based ant colony optimization. Applied Soft Computing **4**(3), 303 – 322 (2004). Special Issue on Hardware Implementations of Soft Computing Techniques

82. Schulz, C.: Efficient local search on the GPU – Investigations on the vehicle routing problem. Journal of Parallel and Distributed Computing **73**(1), 14 – 31 (2013). Metaheuristics on GPUs
83. Solomon, S., Thulasiraman, P., Thulasiram, R.: Collaborative multi-swarm PSO for task matching using graphics processing units. In: Proceedings of the 13th annual conference on Genetic and evolutionary computation, GECCO '11, pp. 1563–1570. ACM, New York, NY, USA (2011)
84. Spampinato, D., Elster, A.: Linear optimization on modern GPUs. In: Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on, pp. 1 –8 (2009)
85. Spiessens, P., Manderick, B.: A Massively Parallel Genetic Algorithm Implementation and First Analysis. In: Proceedings of 4th International Conference on Genetic Algorithms (1991)
86. Stivala, A., Stuckey, P., Wirth, A.: Fast and accurate protein substructure searching with simulated annealing and GPUs. BMC Bioinformatics **11**(1), 446 (2010)
87. Talbi, E.: Parallel combinatorial optimization. Wiley series on parallel and distributed computing. Wiley-Interscience (2006)
88. Tran, Q.N.: Designing Efficient Many-Core Parallel Algorithms for All-Pairs Shortest-Paths Using CUDA. In: Proceedings of the 2010 Seventh International Conference on Information Technology: New Generations, ITNG '10, pp. 7–12. IEEE Computer Society, Washington, DC, USA (2010)
89. Uchida, A., Ito, Y., Nakano, K.: An Efficient GPU Implementation of Ant Colony Optimization for the Traveling Salesman Problem. Third International Conference on Networking and Computing pp. 94–102 (2012)
90. Wang, J., Dong, J., Zhang, C.: Implementation of Ant Colony Algorithm Based on GPU. In: Computer Graphics, Imaging and Visualization, 2009. CGIV '09. Sixth International Conference on, pp. 50 –53 (2009)
91. Weiss, R.M.: GPU-Accelerated Data Mining with Swarm Intelligence. Honors Thesis. Department of Computer Science. Macalester College http://metislogic.net/thesis.pdf (2010)
92. Wikipedia: Streaming SIMD Extensions. http://en.wikipedia.org/wiki/Streaming_SIMD_Extensions
93. Wong, M.L.: Parallel multi-objective evolutionary algorithms on graphics processing units. In: Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference: Late Breaking Papers, GECCO '09, pp. 2515–2522. ACM, New York, NY, USA (2009)
94. Wong, M.L., Wong, T.T.: Parallel Hybrid Genetic Algorithms on Consumer-Level Graphics Hardware. In: Evolutionary Computation, 2006. CEC 2006. IEEE Congress on, pp. 2973 –2980 (2006)
95. Wong, M.L., Wong, T.T., Fok, K.L.: Parallel evolutionary algorithms on graphics processing unit. In: Evolutionary Computation, 2005. The 2005 IEEE Congress on, vol. 3, pp. 2286 – 2293 Vol. 3 (2005)
96. You, Y.S.: Parallel Ant System for Traveling Salesman Problem on GPUs. URL http://www.gpgpgpu.com/gecco2009. Entry in 'GPUs for Genetic and Evolutionary Computation' competition, GECCO 2009, 2 pages
97. Yu, Q., Chen, C., Pan, Z.: Parallel Genetic Algorithms on Programmable Graphics Hardware. In: L. Wang, K. Chen, Y. Ong (eds.) ICNC 2005, LNCS 3612, pp. 1051–1059 (2005)
98. Zhao, J., Liu, Q., Wang, W., Wei, Z., Shi, P.: A parallel immune algorithm for traveling salesman problem and its application on cold rolling scheduling. Information Sciences **181**(7), 1212 – 1223 (2011)
99. Zhu, W., Curry, J., Marquez, A.: SIMD tabu search for the quadratic assignment problem with graphics hardware acceleration. International Journal of Production Research **48**(4), 1035–1047 (2010)