Accelerated Combinatorial Optimization using Graphics Processing Units and C++ AMP

Alexandru Voicu
Beyond3D
45 Cambridge Place,
Cambridge
CB2 1NS, United Kingdom

ABSTRACT
In the course of less than a decade, Graphics Processing Units (GPUs) have evolved from narrowly scoped application specific accelerators to general-purpose parallel machines capable of accommodating an ever-growing set of algorithms. At the same time, programming GPUs appears to have become trapped around an attractor characterised by ad-hoc practices, non-portable implementations and inexact, uninformative performance reporting. The purpose of this paper is two-fold, on one hand pursuing an in-depth look at GPU hardware and its characteristics, and on the other demonstrating that portable, generic, mathematically grounded programming of these machines is possible and desirable. An agent-based meta-heuristic, the Max-Min Ant System (MMAS), provides the context. The major contributions brought about by this article are the following: (1) an optimal, portable, generic-algorithm based MMAS implementation is derived; (2) an in-depth analysis of AMD's Graphics Core Next (GCN) GPU and the C++ AMP programming model is supplied; (3) a more robust approach to performance reporting is presented; (4) novel techniques for raising the abstraction level without sacrificing performance are employed. This represents the first implementation of an algorithm from the Ant Colony Optimisation (ACO) family using C++ AMP, whilst at the same time being one of the first uses of the latter programming environment.

General Terms
Algorithms, Optimization, Parallel Programming, GPGPU

Keywords
C++ AMP, Generic Programming, GPU Programming, MAX-MIN Ant System, Parallel Meta-Heuristics

1. INTRODUCTION
Combinatorial optimisation [1] is one of the key pillars of modern quantitative research, being present in some form or another numerous fields and having become one of the staples of robust scientific analysis in spite of its youth [2]. It is therefore unfortunate, that the computational demands typically encountered in the field are extensive and expansive, with the limit being represented by NP-completeness [3]. Two developments have sprung forth from this reality. First, researchers have moved from exact algorithms to approximating ones, with meta-heuristics being a noteworthy example in recent years. Second, the need for additional computational performance is never exhausted, with ever more intricate problems arising and requiring ever more processor cycles. Given the above, we can establish the broad context for our work, which focuses on accelerating a particular combinatorial, optimisation-centric meta-heuristic, by levering the GPU.

The Max-Min Ant System (MMAS) [4]–[7], is part of the Ant Colony Optimisation (ACO) [8] family of agent-based meta-heuristics. This class of algorithms uses simple agents, which individually traverse some search space and then indirectly exchange information through a shared fabric symbolically associated with the process of pheromone deposition included in the foraging behaviour of real-world ants. The latter steps act as a means of guiding the search towards potentially interesting optima, since the stochastic process that drives the choice of vertex traversal order is biased by the accumulation of pheromones. A recently published survey [9] shows that ACO algorithms have wide applicability in combinatorial optimisation. Out of all developments of ACO, MMAS appears to be the most robust and successful. Furthermore, literature about Graphics Processing Unit (GPU) based acceleration of MMAS is rather recent, thus providing us with insight into the state-of-the-art. The GPU has drawn notable attention in the scientific community, bringing the promise of extremely high performance at an accessible price. Frequently, the literature quotes orders of magnitude improvements. We, and by extension this paper, take a more cautious look, drawing from the school of thought represented by [10], [11], and therefore elect to focus on a challenging test scenario and on accurate and extensive reporting of performance.

We initially detail the MMAS algorithm and focus on its application to the Travelling Salesman Problem (TSP). We then proceed to conduct an in-depth analysis of an advanced GPU architecture, the AMD Graphics Core Next (GCN) chip [12]. The C++ AMP [13] augmentation of the C++ language, which we have chosen as GPU programming environment, is described. Thus, it becomes possible to proceed with conducting an in-depth performance analysis in tandem with a description of our software architecture choices and their motivation. We demonstrate performance that is both much higher than the one presented in previous works and portable across GPUs.

2. THE MAX-MIN ANT SYSTEM AND ITS APPLICATION TO THE TSP
For self-containment of this work, we will detail the MMAS algorithm, as described in [6]. Whilst we will return to this point later on, we clarify *ab initio* that our implementation follows the canonical form laid out in the aforementioned reference, to ensure comparability with reference works in the field. MMAS came as an improvement upon the first ACO algorithm, the Ant System (AS) [8]. AS was introduced as a solver for the Travelling Salesman Problem (TSP) [14], one of

the most widely studied and accepted test-beds in combinatorial optimisation; results for larger instances were unsatisfactory, whilst performance requirements were high. MMAS proved significantly more robust, augmenting the exploitation of the search history phase to ensure more resilience in the face of early convergence to local optima. An ACO algorithm, and by extension MMAS, is an agent-based approach which relies on simplistic agents (ants) for conducting an oriented search through the solution space associated with a particular combinatorial optimisation problem. Ants are myopic in that they have no look ahead possibility, choosing the order of traversal based on realizations of a stochastic process. Once solutions are constructed, the option exists to rely on a separate optimisation algorithm to tune the resulting tours further. Irrespective of whether or not the "daemon" option is acted upon, an iteration concludes with an indirectly cooperative process that sees the ants updating a globally visible pheromone "map". Pheromone values are adjusted based on solution quality, and are subsequently factored in the agents' choices in following iterations. This ensures the stochastic choice is biased towards particular components which are encountered in "better" solutions, where better is a problem dependent metric. Figure 1 schematically reproduces the algorithmic flow:

Initialize: set up invariants, seed pheromone "map";
Repeat: construct_solutions;
daemon_action; // Optional step.
update_pheromones;
Until: convergence achieved or termination criterion met.

Figure 1. Algorithmic flow for the ACO algorithms

A TSP is fully defined by a complete, directed weighted graph $G = (V, A, d)$ with $V = \{1, \dots, n\}$ the set of vertices, $A = \{(i,j) | (i,j) \in V \times V\}$ the set of edges and $d: A \to U^+$ a weighing function that maps from the set of edges into a set of positive values U. It is usually the case that $U = \mathbb{N}^+$, however we find this restriction unnecessary, as, in general, it suffices if U has a pseudo additive semi-group structure and allows for weak ordering of its elements. The optimisation problem, in this particular context, is to find the minimum weight Hamiltonian cycle attached to G, or, more intuitively stated, the path that uniquely includes all vertices in the graph and minimizes the sum of attached weights. There are two typical cases for the TSP, the symmetrical one (STSP), where $d_{ij} = d_{ji}\ \forall\ i,j \in V$, and the asymmetrical one (ATSP), where $d_{ij} \neq d_{ji}$ for some $i,j \in V$. We shall focus on the STSP case. The solution to an STSP is expressible as a vector $S^n \in \mathbb{N}$, for which any two subsequent components form an edge in a complete cycle through the graph. Then, in an iteration t, the pheromone "map" contains a quantity $\tau_{ij}(t)$ which represents the intensity of the pheromone associated with a particular edge. This value is adjusted at the end of each iteration. The algorithm proceeds as follows:

1. **construct_tour phase:** a number $m \leq n$ of ants are randomly assigned a starting node from V; in $t \leq (n-1)$ construction steps, each ant builds a cycle through the graph, by choosing the next vertex to move to, on the basis of a discrete probability distribution:

$$p_{ij}^k(t) = \begin{cases} \dfrac{[\tau_{ij}(t)]^\alpha \times [\eta_{ij}(t)]^\beta}{\sum_{l \in N_i^k} [\tau_{il}(t)]^\alpha \times [\eta_{il}(t)]^\beta}, & j \in N_i^k \\ 0, & j \notin N_i^k \end{cases} \quad (1)$$

$\eta_{ij} = \frac{1}{d_{ij}}, \alpha, \beta \in \mathbb{R}$ elasticities, N_i^k feasible set for ant_k

The feasible set is the set of unvisited nodes by rapport with any ant_k i.e. there are m distinct sets at any time t. Interestingly, it can be observed that the probability mass function (PMF) is in effect a Cobb-Douglas [15] form. Furthermore, the η_{ij} quantity ("heuristic" value in literature) is constant, thus the PMF is actually uni-variate, and α-homogeneous. We are not aware, at the time of writing, of any analysis that has placed emphasis on these traits, and will seek to present one in a subsequent publication.

2. **daemon_action phase (optional):** at this stage it is possible to for a number $p \leq m$ ants to apply some form of local optimisation to the tours constructed in the prior phase, with common choices being the 2-opt, 3-opt or Lin-Kernighan algorithms [16].

3. **update_pheromones phase:** as a last step in an iteration t, the pheromone values are updated, under the influence of a constant decay ("evaporation") process and, respectively, an accumulation process:

$$\tau_{ij}(t+1) = \rho \times \tau_{ij}(t) + \sum_{k=1}^{m} \Delta\tau_{ij}^k(t) \quad (2)$$

$\rho \in [0,1)$ persistence, $\Delta\tau_{ij}^k(t) = \uparrow (\tau_{ij})\ \forall (i,j) \in cycle_k$

The obvious exponential decay affecting edges that are not part of the cycle reduces the probability of the latter being chosen in future cycles. In MMAS, only edges that are contained in the iteration best or globally best tour are incremented, with the schedule for the choice between these two latter categories being detailed in [6], therefore in (2) the final sum simplifies to $\Delta\tau_{ij}^{\{I,G\}_{best}}(t)$, where:

$$\Delta\tau_{ij}^{\{I,G\}}(t) = \begin{cases} \dfrac{1}{L_{\{I,G\}_{best}}}, & (i,j) \in cycle_k \\ 0, & (i,j) \notin cycle_k \end{cases} \quad (3)$$

$L: \mathbb{N}^2 \to U^+$ a cost function e.g. $\sum_{i=1}^{n} d(i,j), \forall (i,j) \in cycle_k$

Furthermore, unlike in the case of AS, the quantity $\tau_{ij}(t)$ has a strict upper and lower bound:

$$\limsup \tau_{ij} \triangleq \tau_{ij}^{MAX}(t) = \sum_{i=1}^{t} \rho^{t-i} \times \frac{1}{L_{OPT}} + \rho^t \times \tau_{ij}(0) \quad (4)$$

$\lim_{i \to \infty} \tau_{ij}^{MAX} = \frac{1}{1-\rho} \times \frac{1}{L_{OPT}}, L_{OPT} \approx$ cost of optimal tour

$$\liminf \tau_{ij} \triangleq \tau_{ij}^{MIN}(t) = \frac{\tau_{ij}^{MAX}(t) \times \left(1 - \sqrt[n]{p_{best}}\right)}{\left(\frac{n}{2} - 1\right) \times \sqrt[n]{p_{best}}} \quad (5)$$

p_{best} = the probability of making the optimal choice.

This modification ensures that no edge is removed from the candidate set. Moreover, due to the upper bound the most frequently chosen edges never end up strongly dominating selection. The two effects' interaction leads to an avoidance of early convergence to suboptimal minima.

The above, abridged, representation is sufficient for the following conjecture: MMAS does not naturally lend itself across the board to one of the classical models for parallel

computation, be it task-parallelism [17] or data-parallelism [18]. We contend that an optimal mapping probably adheres to the braided parallelism paradigm [19], in which data parallel bits (e.g. the construct_tour phase) are interwoven with serial or weakly parallel ones (i.e. the selection of the best tour in a particular iteration), with tasks acting as software units of scheduling. As is often the case with meta-heuristics, there are no tight, proven, bounds on MMAS' convergence behaviour. [20] provides a rather generic proof of convergence in value. [21] shows notably tighter bounds on expected runtime for certain MMAS classes: $\Theta(n \log n)$ and $\Theta(n^2)$ - this makes it clear that the algorithm is subject to the curse of dimensionality [5]. We now briefly sample the literature in the area of parallel MMAS implementations. We find investigations being conducted on multi-core and many-core Central Processing Unit (CPU) [22]–[28], which are congruent in their observation that the primary bound on achievable parallel speed-ups is the data-movement / exchange. Consequently, the best results in these contexts, which tend to be characterized by slow and latency-affected inter-CPU exchanges, are achieved when multiple concurrent executions are undertaken or, at worse, minimal data is exchanged (e.g. only broadcasting the globally best tour). In ACO specific terminology, this maps into running either multiple independent algorithm executions in parallel or multiple colonies in parallel.

In recent years, under the influence of increasing interest in GPUs as a research vehicle, a number of authors have published ACO / MMAS on GPU works [29]–[35]. In terms of exploited parallelism, we encounter varying degrees; some works only move tour-construction to the GPU, whereas by contrast [30] does only ancillary work on-CPU, claiming remarkable speed-ups versus the ACOTSP code provided with [36]. [37] provides a comprehensive overview of the field and is also the first text that uses the OpenCL [38] API, including a proposed implementation, although no performance evaluation is provided. We believe that, whilst these forays into the realm of GPU ACO / MMAS are very valuable data-points, there is ample room for improvement. With this clarification in place, we proceed to a discussion about GPU architecture in general, and AMD's Graphics Core Next (GCN) [12] in particular.

3. AMD GCN ARCHITECTURE

We first formulate the following general observations, given their relevance in context:

1. in spite of the notable obfuscation affecting the field, GPUs are not new classes of processors, but rather multi-core processors that employ Single Instruction Multiple Data (SIMD) execution to increase computational density within a core;
2. the frequently encountered Single Instruction Multiple Thread (SIMT) [39] or Single Program Multiple Data (SPMD) [40] descriptions associated with GPUs are primarily properties of the GPU programming model;
3. from (1) and (2) it follows that it is adequate to describe modern GPUs as moderately many-core machines – highest end configurations enable at 32 SIMD Cores for AMD's GCN [12] and 15 SIMD Cores for NVIDIA's Kepler [41]; the "thousands of cores" assessment is inaccurate, as those reflect the count of Arithmetic Logic Units (ALUs) / SIMD lanes available on a particular chip;
4. the GPU memory hierarchy is different from that of CPUs in that register files are frequently larger than caches or scratch-pads, in order to support the concurrent execution of multiple batches, coupled with fast context switching used to hide high-latency operations without stalling the processor.

We shall now focus our analysis on the GCN architecture. A GCN processor includes a variable count of Compute Units (CUs). Each CU subsumes a Vector Unit (VU) and a Scalar Unit (SU). The former is implemented as four 16-wide SIMD units, whereas the latter is a general-purpose integer-only processor. The vector ALUs support a full complement of integer and IEEE-754 compliant floating-point operations, with throughput varying based on data-type [12]. Transcendental operations can be efficiently evaluated using microcode, as long as a sacrifice in precision is acceptable. The hardware unit of scheduling (batch or "wavefront" in AMD terminology) is 64-wide, and all elements being operated on within a single cycle are guaranteed to be processed in lockstep. A 32 KB, four-way, L1 Instruction Cache services groups of four CUs, using 64 Byte lines and a Least Recently Used (LRU) replacement policy. Unless explicitly stated, assume these characteristics for all other caches. Instructions are fetched at a rate of 32 Bytes per cycle.

The SU handles control flow at the CU level, de-centralizing its management across the chip. It incorporates two pipelines, one dedicated to handling conditional branches, as well as interrupts and CU-level synchronization. The other pipeline has the full integer processing capability, handling address generation and non-linear control flow. Hardware assisted predication is also handled at this level. GPUs hide their SIMD nature and allow the programmer the illusion of having independent control flow per SIMD lane by using multiple execution, masking and predication. In general, if lanes within the same unit of scheduling "diverge", all paths are executed and, upon reaching the first nearest dominator in the program, lanes are re-converged by predicated writes. GCN uses an 8 KB register file divided into 512 32-bit entries for each 16-wide SIMD, where the SU can write, for example, the results of a comparison for each element in a batch. The SU is serviced by a 16 KB read-only, 4-way associative, 4-banked, L1 scalar data cache. Up to 4 CUs can share a single scalar cache, each reading up to 16 Bytes per cycle.

The VU is the primary locus of processing on a GCN GPU. It accesses a 256 KB register file, partitioned into four equal-sized slices, each owned by one of the 16-wide SIMDs. Batches are scheduled to the SIMDs in one cycle, but take 4 cycles to complete. Scheduling is done in a round robin fashion, the SU issuing one instruction to one SIMD per cycle. Each of the SIMDs has enough resources to track up to 10 batches, as long as register and Local Data Share (LDS) requirements are met. Across four SIMDs, this leads to an upper bound of 40 batches resident per CU. The above translate into the ability of transparently hiding a dependency chain that is up to 40 cycles.

The next fastest memory structure accessible to the VU is the LDS. The LDS is a heavily multi-banked array of SRAM (16 or 32 banks); each bank is arranged as a 512 deep stack of un-typed 32-bit entries. An all-to-all crossbar interface lies between the LDS and the VU, making it possible to access one entry from each bank per cycle, with conflicts automatically handled through serialization. Atomic operations are available at the same rate. LDS accesses do not occupy ALU resources, and there is efficient broadcast capability embedded in the hardware, making it possible for all SIMD lanes to access a single entry within a single cycle. If, for example, the data access patterns are difficult to predict

and thus hard to exploit via a software-managed cache, the VU is backed by a hardware-controlled 16 KB Read / Write L1 Data cache, that implements a write-through / write-allocate policy. Another innovation for GCN is that this cache can be made coherent with the L2 cache and, by extension, with the other L1s, following a relaxed consistency model aligned with the C++11 one, for which details can be found in Chapter 5 of [42]. For liner access patterns, up to 16 32 -bit values can be read from the cache per cycle. In terms of pure throughput, this is inferior to the LDS, albeit it is obviously more programmer friendly given that no explicit cache management is needed.

The general chip-wide caching structure comes in the form of a distributed Read / Write L2 cache. The cache is physically split into S slices, where $S = \boldsymbol{count}(\text{memory\_controllers})$, $\boldsymbol{sizeof}(s_i) \in \{64, 128\}, \forall\, i \in (0, S)$ and $\boldsymbol{sizeof}$ returns the size of its argument in KBytes. Accessible through a crossbar fabric, the L2 is 16-way associative. Up to 16, conflict-free, atomic ops to the same cache-line can be resolved per cycle, thus allowing for the implementation of efficient global synchronization primitives. Acquire / Release semantics are used for maintaining coherence, in accordance with a relaxed consistency model.

At the base of the memory hierarchy pyramid lies the main GPU RAM, accessed through C 64-bit wide memory controllers, $C \in \mathbb{N}^+, S = C$. Each controller is wired into two independent 32-bit GDDR5 memory channels, albeit DDR3 configurations are possible [12]. For top end configurations, much greater bandwidth than that available to a top-end CPU is provided. Optimal throughput is obtained when accesses use a stride that sees separate batches address different controllers / channels. It is very important to note that access to this memory level is a high latency, very low throughput operation when compared to all other levels. It is therefore important to exploit caching both explicitly (through register and LDS use) and implicitly (through cache-friendly access patterns).

In closing, let us consider the chip-wide scheduling hardware, which encompasses the command processor (CP) and A Asynchronous Compute Engines (ACE), $A \in \mathbb{N}^+$. The CP is a micro-controller using a RISC-like ISA, tasked with interfacing with the host machine. It issues DMA requests for fetching command buffers from host RAM through the PCI-Express Bus. Furthermore, it maps the command stream to the hardware for execution. The ACE manages resource allocation and task scheduling for all GPU compute tasks, fetching commands through the memory hierarchy and establishing priority based work-queues. If resources are available and dependency chains allow it, it is possible for different tasks to execute concurrently. Execution is, potentially, out-of-order, and retirement is in-order, ordering being maintained by the ACE. Inter-ACE synchronization is possible through the memory hierarchy. Based on this analysis we can derive a series of observations:

1. opportunities for data re-use across SIMD lanes must be identified and exploited by caching the LDS;
2. optimal performance requires sizing work-units as integral multiples of 64, the size of the hardware unit of scheduling;
3. accesses of unit stride equal to 32-bits yield efficient accesses in all the levels of the memory hierarchy;
4. the frequently suggested strategy of dispatching as much work as possible is potentially sub-optimal.

Progressing beyond hardware aspects, the next section focuses on introducing C++ AMP [13].

4. C++ AMP

C++ AMP is a combination between a library and a minimal language extension, proposed as a productivity focused alternative to lower-level parallel compute APIs [38] or proprietary ones [39]. Its specification is open and publicly available. AMP can be layered on top of various lower-level APIs [43], which decouples it from any particular Operating System or tool-set and makes it portable. An aspect that is generally applicable to all GPU programming interfaces currently available is that they operate in an indirect model. This entails that command buffers generated on the CPU side and subsequently dispatched, with availability of work results being query-able in an asynchronous fashion, and no support for pre-empting or other, more advanced interfacing with the GPU. AMP is designed as a C++11 library, drawing inspiration from the Standard Template Library (STL) [44]. An ample subset of the language is supported, enabling, for example, the construction of Abstract Data Types (ADTs) or inheritance hierarchies, the use of lambda functions or templates [45]. The construct `restrict(amp)` acts as a decorator for function signatures, and is employed to enforce adherence to the supported set of language features and signal the programmer's intent of executing on an accelerator such as a GPU.

AMP code is in-line C++, directly integrated with application code and compiled as such. The entry-point for accelerator execution is an overload of the function template `parallel_for_each` taking in:

- a parameter of type `accelerator_view`, which abstracts the locus of execution;
- a parameter of type `extent<int>`, which defines the rank and size of the space of execution;
- a parameter that models the concept of a function which describes the computation to be performed for each element in the domain, and which receives as an argument the position within the space of execution.

The execution domain is iterated across either in a linear, ungrouped fashion, case in which coordinate data is passed in the form of an `index<int>` object, or in a tiled, structured mode, case in which coordinate data is a `tiled_index<…>` object. The latter case adds additional dimensionality / structuring to the execution space in the form of tiles that offer finer grained control over per-element execution (e.g. synchronization) and can access the hardware scratchpad. Tiles are the software concept exposing the underlying SIMD nature of execution, being associated with the SIMD threads executing on the machine. An additional storage class, `tile_static`, has been added in order to denote data being cached in the scratchpad. The interested reader is directed to the more extended exposés in [13]. Let CM be the set of CPU visible / accessible memory addresses and GM the equivalent structure defined for the GPU. For current C++ AMP versions, $CM \cap GM = \emptyset$, and thus a conduit for marshalling data from one domain to the other is necessary. In our work, we employ the mechanism exposed by the class template `array_view<typename T, int N>` to pass data seamlessly across domains. In practice, `array_view` is a lightweight wrapper over a dense range of data, which offers implicit, runtime-controlled data-movement. The AMP programming model allows for explicit synchronization within any particular tile, exposing `barrier` objects, as well

as fences for cases in which merely ensuring weak ordering without requiring consistency suffices. Synchronization between tiles is guaranteed only at a `parallel_for_each` invocation boundary. Overall, this yields an interface to the GPU hardware that does not inhibit optimal exploitation, whilst being considerably less verbose and, in our experience, more productive. The ability to define ADTs, coupled with the generality brought by having access to the C++ template mechanism, has been extremely useful in practice.

5. A HIGH-PERFORMANCE C++ AMP IMPLEMENTATION OF MMAS

The hardware structure and nature suggest some generally desirable features. Let TS be the set of all tiles. Then we can formulate the following general points about about optimal structuring of execution on GCN hardware:

1. for maximising latency-hiding, a sufficient number of batches has to be in flight at any time, where sufficiency is defined as follows:

$$\frac{|TS| \times tile_{size}}{batch_{size}} \geq \Gamma \times \boldsymbol{count}(\text{CU}) \tag{7}$$

$$\Gamma \in \mathbb{N}^{+} \setminus [1,3], tile_{size} \in \mathbb{N}^{+} \setminus (1024, \infty), batch_{size} \triangleq 64$$

2. for optimally exploiting CU resources:
   a. $tile_{size} = k \times batch_{size},\ k \in \mathbb{N}^{+}$;
   b. $\frac{scratchpad_{size}}{|TS|} \leq 32$ KBytes;

For the implementation, we focus on type properties and supported operations, and rely on self-designed ADTs, adhering to guidelines laid out in [46]. Three, progressively refined, approaches are investigated, each representing a refinement of the prior. We establish the following invariants:

1. let AN be the set of ants, $|AN| = m$, TH the set of SIMD threads, $|TH| = |TS|$; we choose to implement a non-injective mapping $\Xi: AN \rightarrow TH$, which associates ants with the lanes within a SIMD thread – this choice is not only supported by our analysis, but also by the literature [30], [31], [35]; this bounds the value of $m = |TS|$;
2. we hold $tile_{size} = batch_{size} = 64$ fixed, which allows us to implement size-aware barriers [47] and by way of consequence obtain barrier elision and thus lightweight synchronization;
3. the canonical selection rule from Eq. (1) is substituted with I-roulette selection [30] in the construct_solutions phase;
4. the prior points lead to developing an ADT that models the ant queen concept i.e. a class template `Amp_max_min_as_queen<typename T, unsigned int colony_size, unsigned int tour_sz>`, which coordinates the search, where:
   a. T is the type of the quantities used in I-roulette selection, i.e. real numbers;
   b. colony_size = $tile\_size$;
   c. tour_size = n – we use template meta-programming to feed n, a runtime value, into a compile time constant;
   d. each `Ant_queen_max_min_as` object references two containers allocated in `tile_static` storage, an array that acts as scratch-space for intermediate computations and a bit set (similar to `std::bitset` [44]) which holds the list of visited nodes;
   e. PRN generation is done per lane, by way of a Ranlim32 generator [48] which we implemented in C++ AMP.
5. for the daemon_action phase we chose to apply 2.5-opt [49] by way of an optimized C++ AMP implementation that we derive starting from [16], [49]–[51].

For implementing the MMAS solver concept, we develop a `Max_min_as` ADT, a class that hooks into our performance measurement harness and our problem specification loading one. Data representation is separated into a class `template<typename T> Max_min_as_rep`, which contains the physical buffers – we use `std::vector` – wrapped in `array_view`s, which are accessed on the GPU. Our MMAS implementation uses the alternation between using the best tour in an iteration and the best tour across the entire execution in accordance with the schedule proposed in [6]. Given that our execution set up implies that $|AN| = |TS| \leq n$, we have to account for the situation in which there are less ant queens than potential starting nodes. In this case, starting nodes are assigned pseudo-randomly to ants, ensuring that no starting node is assigned more than once within a run. We also exploit the observation that an optimal tour can be found within a low-order sub-graph of G [4], [52]. Let $NNL \subset V$ be a set of nodes, and $a \in V$ an arbitrary node. By imposing a weak, strictly non-increasing, ordering on NNL based on $d(a, nn)$, $\forall\, nn \in NNL$, we obtain the set of a's $|NNL|$ nearest neighbours. Based on the initial observation we assert that optimality or near optimality, i.e. the shortest tour, can be achieved whilst limiting the search for the next node to NNL. We set $|NNL| = tile_{size}$, which simplifies processing, and generate the n sets upon initial data set-up, storing them in a matrix. I-roulette selection is used only when choosing from this set, after which we fall back to making a simple greedy choice amongst the non-visited nodes which are not in NNL.

Table 1. Data containers and their characteristics

Name	Data-type	Size	Contents
choice_info	`float`	n^2	$\tau_{ij}^{\alpha} \times \eta_{ij}^{\beta}$
choice_info_nn	`float`	$n \times \lvert NNL \rvert$	$\tau_{ij}^{\alpha} \times \eta_{ij}^{\beta}$ for $\lvert NNL \rvert$
costs	`unsigned int`	n^2	$d(i,j)$
glob_best _tour	`unsigned int`	n	$L_{\{I,G\}_{best}}$
heuristics	`float`	n^2	η_{ij}
nnlist	`unsigned int`	$n \times \lvert NNL \rvert$	$\lvert NNL \rvert_a$
pheromones	`float`	n^2	τ_{ij}
temp_tours	`unsigned int`	$\lvert AN \rvert \times n$	Per iteration tours.

The following approaches have been developed:

A. in the construction phase, the full tour is stored in `tile_static` memory; the complete undergoes local optimisation and no candidate lists are used; the best tour is selected on the CPU, by identifying the minimum cost across all the tours generated within an iteration;
B. nodes are cached in registers as they are added to the tour, and once $tile_{size}$ have been cached they are written out to temp_tours – this is repeated until tour completion; 64-node sub-tours undergo local optimisation, using a

64-element candidate list; selection is remains on the CPU;

C. same as B, with selection being moved to the GPU.

Some features are common to all the solutions we have developed. We extract / provide as much information as possible at compile time, and then feed in through non-type template parameters. All implementations are flexible in what regards their configuration: MMAS parameters, $tile_{size}$, $|AN|$, $|NNL|$, data-types etc. We ensure coalesced accessing and conflict minimisation by using strides aligned that are integral multiples of $batch_{size}$. In closing, let us consider the concrete, systematic execution of an iteration:

1. the choice_info matrix is updated through a non-tiled `parallel_for_each` across n^2 elements;
2. the choice_info_nn matrix is updated through a non-tiled `parallel_for_each` across $n \times |NNL|$ elements, reading the value found at nnlist $(i,j)\ \forall\ i \in V, j < |NNL|$ and copying it into choice_info_nn (i,j);
3. construct_solutions phase:
   a. through a tiled `parallel_for_each` across $|AN|$ tiles, $|AN|$ tours are constructed and written to temp_tours:
      i. first nnlist $(node_{current}, j) \forall\ j < |NNL|$ is checked for non-visited candidates, and apply I-Roulette selection;
      ii. if no non-visited candidates are found in nnlist, greedily choose the node with $\max \tau_{ij}^{\alpha} \times \eta_{ij}^{\beta}$;
      iii. mark node as visited.
   b. through a tiled `parallel_for_each` with the same dispatch structure as (a), apply 2.5-opt;
   c. through a tiled `parallel_for_each` with the same dispatch structure as (a) compute and store costs for the new tours;
4. identify the best tour for the iteration:
   i. for (A) and (B), read back costs to the CPU, extract the row in temp_tours that is associated with the minimum cost;
   ii. (C), tiled `parallel_for_each` invocation across one tile;
   iii. if the new minimum is a global minimum, update parameters and store in glob_best_tour;
5. update pheromones matrix:
   a. through a non-tiled `parallel_for_each` across n^2 elements apply Eq. (2);
   b. through a non-tiled `parallel_for_each` across n elements apply Eq. (3) to the n edges of the best tour (iteration or global, in accordance with the schedule).

6. PERFORMANCE EVALUATION

6.1 Experimental Methodology

We used the following platform:

- CPU: 2 modules / 4 thread AMD A10-4600M running at 2300MHz (Turbo mode was disabled);
- RAM: 8 GB 1600 MHz DDR3, 128-bit interface;
- GPU: AMD Radeon 7730M, 8 CUs running at 575 MHz, 2 GB 1800 MHz DDR3 RAM, 128-bit memory interface;
- Environment: Microsoft Windows 8 Professional, Microsoft Visual Studio 2012 Update 2 + Visual Studio 2012 November 2012 CTP (v120) and AMD Catalyst 13.2 drivers for the GPU; for compilation, we use full optimization (/Ox), favour speed (/Ot), generate AVX code (/arch: AVX), enable whole program optimization (/GL).

MMAS parameters are set in accordance with [52]: $|AN| = |TS| = 128, tile_{size} = 64, |NNL| = 64$. We time execution and initialization separately, using the high-resolution timer exposed by the C++ standard library. We run 11 full solves, discarding the first and recording the subsequent 10, each solve iterating 2500 times. We focus our investigation on the TSPs shown in Table 2, available in the TSPLIB library [53]:

Table 2. Candidate problems

Name	pcb1173	d1291	d2103	pr2392	pcb3038
Size	1173	1291	2103	2392	3038

We report exact timings as opposed to reporting speed-ups. Proper performance characterisations are crucial in allowing comparability between separate works.

6.2 Performance measurements

6.2.1 Analysis of proposed solutions

Table 3 synthesizes the performance characteristics of the three approaches we have developed A, B and C, expressed as the median time per iteration in milliseconds:

Table 3. Performance evaluation of variants

*	pcb1173	d1291	d2103	pr2392	pcb3038
A* (ms)	31	25	73	90	259
B (ms)	16	16	27	32	43
C (ms)	14	15	26	30	41
*For A, $|AN| = 48$ to avoid triggering TDR events [63].					

The on-GPU execution model of is the fastest, due to lowering the pressure exerted on `tile_static` memory, moving to register caching and using bounded neighbourhoods during local optimisation. The primary performance gain is attributable to the candidate list optimization, caeteris paribus by rapport with A. The benefits of reduced coupling between execution set up and problem size is apparent in the weakening of the relationship between the latter and iteration time. We define quality as the percentile deviation from the known optimum tour: $\%\Delta_{quality} = \frac{L_{\{G_{best}, median\}} - L_{optimum}}{L_{optimum}}$. Alongside the best solution, we consider the sample's median value and its standard deviation (expressed as percentage of the best-known solution). The quality yielded by the three approaches when dealing with the largest TSPs is presented in Table 4:

Table 4. Solution quality evaluation

*	pr2392	pcb3038
$L_{optimum}$	378032	137694

$\%\Delta_{quality}$	$L_{G_{best}}$	L_{median}	σ^2	$L_{G_{best}}$	L_{median}	σ^2
A (%)	1.60	1.87	0.29	1.63	2.25	0.30
B, C (%)	2.71	3.31	0.42	2.94	3.31	0.20

Obviously, the optimisations added to the local search phase have an unfavourable impact on solution quality. This is expected, as moving to a bounded neighbourhood means that potentially beneficial swaps are never considered. However, the maximum deviation registered with C is under 4%. The difficulty associated with partitioning a tour into sub-tours and optimising only to the latter is a known issue [54]. We are confident that we can diminish the adverse effects to the point of nullification by overhauling our simple partitioning scheme to match, for example, [55], and we shall investigate this.

6.2.2 Comparative analysis versus the literature

We compare C against other implementations from the literature. This is a difficult endeavour, because not all write-ups provide absolute measurements, or a way of deriving them. The works that allow for comparability are listed in Table 5:

Table 5. ACO / MMAS on GPU in the literature

Paper	Algorithm	GPU	Prog. Lang.
A1: J. M. Cecilia, J. M. García, A. Nisbet, M. Amos, and M. Ujaldón, "Enhancing data parallelism for Ant Colony Optimization on GPUs" [30]	AS	NVIDIA Tesla C2050	CUDA
A2: A. Uchida, Y. Ito, and K. Nakano, "An Efficient GPU Implementation of Ant Colony Optimization for the Traveling Salesman Problem" [35]	AS	NVIDIA GeForce GTX 580	CUDA
A3: K. Tantawy, "Ant Colony Optimization Parallel Algorithm for GPU" [34]	MMAS	NVIDIA GeForce GTX 480	CUDA

All the above works use hardware that is significantly more powerful than our experimental setup, as well as a proprietary API that is optimised for the underlying GPU. This is made apparent in Table 6, which briefly compares some significant characteristics:

Table 6. Metrics for GPUs used in the literature

GPU	Bandwidth		`int` (ADD)		`float` (MAD)	
A1: Tesla C2050	~144	GB/s	~0.51	TIOPS/s	~1.03	TFLOPS/s
A2: GeForce GTX 580	~192		~0.79		~1.58	
A3: GeForce GTX 480	~177		~0.67		~1.35	

Since A1 and A2 do not implement MMAS but rather AS, we compare only tour-construction mean times, disabling local optimisation and setting $|TS| = n$ (C-AS in the tables). We exploit this opportunity to demonstrate portability, by including performance data from the following two configurations:

1. AMD FX-8120 4 Modules / 8 Threads running at 3.1 GHz, 8 GB of 1333MHz DDR3 RAM, AMD Radeon HD 7970 GPU, 32 CUs running at 925MHz, 3GB of 6000 MHz GDDR5, 384-bit bus, all else being equal to the baseline; we use it to illustrate vertical scaling, i.e. from a low-end part to a high-end on the same architecture;
2. as above, but the GPU is swapped for an NVIDIA GTX 480, using the latest public drivers; we use it to illustrate horizontal scaling, i.e. scaling across different architectures.

Table 7. Performance evaluation versus the literature

*	lin318	pcb442	rat783	pr1002	d1291
C - AS (ms)	3	7	20	34	*
C (ms)	3	5	9	11	15
C – AS 7970 (ms)	1	1	4	6	10
C 7970 (ms)	1	2	4	5	7
C – AS GTX480 (ms)	3	4	8	10	15
C GTX480 (ms)	3	4	8	10	13
A1 (ms)	15	38	207	391	*
A2 (ms)	8	11	56	86	*
A3 (ms)	30	60	310	*	1420

We had to use other, smaller, TSPs in order to obtain alignment between our research and the literature. Improvements can range up to two orders of magnitude better, when comparing C with A3 and, respectively, one order of magnitude better when comparing C to A1. We posit that this is an important development, as it opens up ACO / MMAS experimentation to researchers who do not have access to expensive hardware, and does so in a portable way – our code runs anywhere where there is a DirectX 11 accelerator available. A caveat lies in the fact that vertical scaling from the 7730M to the 7970 is sub-linear when compared to the hardware differential. We shall investigate and correct this in the future.

Assuming that the increases are due to some intrinsic benefit of GCN hardware is incorrect, as we can observe that C yields vastly superior performance when run on the GTX 480, which is close to the hardware used in the other works. We can only conclude that a mix of algorithmic and programming interface benefits is the cause. Since C++ AMP is unlikely to be better than CUDA in terms of extracting performance from NVIDIA hardware, we conjecture that our data structures and algorithms are superior. Extending the analysis beyond this conjecture is difficult, as we have no access to the source code of the other solutions. In closing, we will compare against ACOTSP 1.02 software package [36]. We employ the same parameterisation that we have used up to now for the ACOTSP runs. To ensure optimal execution on a modern CPU, we change compilation parameters to match those used for our code, and perform a series of transforms on the source-code, which enable auto-vectorisation.

Table 8. Performance evaluation versus ACOTSP 1.02

*	pcb1173	d1291	d2103	pr2392	Pcb3038
C (ms)	14	15	26	30	41

ACOTSP	64	76	138	169	243
ACOTSP optimised	56	71	118	132	205

For the purpose of fairness and clarity we emphasize that the CPU we are using is not a high-end one, and that that ACOTSP, whilst properly auto-vectorised after our modifications, is neither fine-tuned nor parallel. Moreover, it uses double-precision floating-point whereas we use single-precision floating-point. At the same time, the GPU we are using is itself a low-end itself, with multiple optimisation opportunities still available. We do not expect the performance gap to be closed, albeit both solutions have room for growth. Before moving to conclusions, we must analyse the quality of the tours yielded by our MMAS implementation. We measure against the best tours generated by ACOTSP by way of the $\%\Delta_{quality}$ quantity computed against the best tour:

Table 9. Tour quality versus ACOTSP

*	d1291	d2103	pr2392	Pcb3038
A (%)	1.13	1.15	1.60	1.63
B, C (%)	2.1	2.34	2.71	2.94
ACOTSP (%)	0.33	0.98	1.56	1.94

Taking into account the penalty associated with using bounded neighbourhoods for local optimisation (B, C), it is clear that in the future we must strive to diminish the unfavourable effects. However, even with the current, sub-optimal, 2.5-opt implementation, C provides results comparable with those of ACOTSP, whilst being much faster. Finally, note that the maximum deviation is under 3%, which is generally acceptable in most scenarios.

7. CONCLUSIONS

We have introduced a high-performance MMAS implementation that exploits C++ AMP to access GPUs. The code is written at a high-level of abstraction, focusing on re-usability. Whilst it is not production ready, we believe that our implementation will prove useful to scientists interested in MMAS and ACO in particular and, more generally, on taking advantage of GPUs in their work. We demonstrate a performance level that establishes a new upper bound in the field of parallel ACO, supplanting the former state-of-the-art. We also provide in-depth analyses of the AMD GCN architecture and the C++ AMP, which hold general relevance. Our aims for the future are the following:

1. exploiting the symmetry embedded in the TSP, which would allow us to lower both the computational complexity and the storage complexity by a constant factor of two;
2. investigating improvements to the local optimisation tour partitioning scheme;
3. in the local optimisation pass, after a swap, only the costs associated with the edges that get deleted and re-wired change, whereas the other costs remain invariant – we could cache in the scratchpad, and only updated to reflect the newly added edges;
4. studying the behaviour of our solution when it is moved to higher-end hardware;
5. studying how our solution generalizes to other problems that can be solved through MMAS.

In a broader scope, we are primarily interested in opening up GPU computation to a wider sample of scientists. This goal can be reached by a lucid understanding of the hardware coupled with an emphasis on writing generic, portable code as opposed to problem specific patchwork. Another important question has to do with identifying the areas where GPUs can actually act as accelerators, as this is currently an incompletely explored topic.

8. ACKNOWLEDGMENTS

The author would like to thank the following individuals for their kindness and insight, without which this work would likely have been far worse: Amit Agarwal, Alex Goh, Cristina Galalae, Boby George, Michael Houston, Lee Howes, Konstantinos Kaparis, Maria Sabrina Preda, James Prior, William Stumpf, Ryszard Sommefeldt and Lingli Zhang. Whilst they have gone above and beyond the call of duty in pointing out errors and failures with the text, it is possible that some remain, all of them the sole responsibility of the author

9. REFERENCES

[1] B. Simeone, Ed., Combinatorial optimization: lectures given at the 3rd session of the Centro internazionale matematico estivo (C.I.M.E.) held at Como, Italy, August 25-September 2, 1986. Berlin ; New York: Springer-Verlag, 1989.

[2] A. Schrijver, “On the history of combinatorial optimization (till 1960),” in Handbooks in Operations Research and Management Science Discrete Optimization., K. Aardal, G. L. Nemhauser, and R. Weismantel, Eds. Burlington: Elsevier, 2005, pp. 1–68.

[3] M. R. Garey and D. S. Johnson, Computers and intractability: a guide to the theory of NP-completeness. San Francisco: W.H. Freeman, 1979.

[4] T. Stützle and H. Hoos, “MAX-MIN Ant System and local search for the traveling salesman problem,” 1997, pp. 309–314.

[5] T. Stützle and H. Hoos, “Improving the Ant System: A detailed report on the MAX-MIN Ant System,” 1996.

[6] [6] T. Stützle and H. H. Hoos, “MAX-MIN Ant System,” Future Gener Comput Syst, vol. 16, no. 9, pp. 889–914, Jun. 2000.

[7] T. Stützle and H. H. Hoos, “Improvements on the Ant-System: Introducing the MAX-MIN Ant System,” in Artificial Neural Nets and Genetic Algorithms, Vienna: Springer Vienna, 1998, pp. 245–249.

[8] M. Dorigo, V. Maniezzo, and A. Colorni, “Ant system: optimization by a colony of cooperating agents,” IEEE Trans. Syst. Man Cybern. Part B Cybern., vol. 26, no. 1, pp. 29–41, Feb. 1996.

[9] T. Stützle, M. López-Ibáñez, and M. Dorigo, “A Concise Overview of Applications of Ant Colony Optimization,” Wiley Encyclopedia of Operations Research and Management Science. John Wiley & Sons, Inc., Hoboken, NJ, USA, 15-Jun-2010.

[10] R. Vuduc, A. Chandramowlishwaran, J. Choi, M. Guney, and A. Shringarpure, “On the Limits of GPU Acceleration,” in Proceedings of the 2Nd USENIX Conference on Hot Topics in Parallelism, Berkeley, CA, USA, 2010, pp. 13–13.

[11] V. W. Lee, P. Hammarlund, R. Singhal, P. Dubey, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, and S. Chennupaty, “Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU,” 2010, p. 451.

[12] M. Mantor and M. Houston, "AMD Graphic Core Next: Low Power High Performance Graphics & Parallel Compute," presented at the High-Performance Graphics 2011, Vancouver, Canada, 07-Aug-2011.

[13] K. Gregory and A. Miller, C++ AMP: accelerated massive parallelism with Microsoft Visual C++. Sebastopol, California: O'Reilly Media, 2012.

[14] D. B. Shmoys, J. K. Lenstra, A. H. G. R. Kan, and E. L. Lawler, Eds., The Traveling salesman problem: a guided tour of combinatorial optimization. Chichester [West Sussex] ; New York: Wiley, 1985.

[15] P. H. Douglas, "The Cobb-Douglas Production Function Once Again: Its History, Its Testing, and Some New Empirical Values," J. Polit. Econ., vol. 84, no. 5, pp. 903–15, 1976.

[16] D. S. Johnson and L. A. McGeoch, "Experimental Analysis of Heuristics for the STSP," in The Traveling Salesman Problem and Its Variations, vol. 12, G. Gutin and A. P. Punnen, Eds. Boston, MA: Springer US, 2007.

[17] J. . Dongarra and D. . Sorensen, "A portable environment for developing parallel FORTRAN programs," Parallel Comput., vol. 5, no. 1–2, pp. 175–186, Jul. 1987.

[18] W. D. Hillis and G. L. Steele, "Data parallel algorithms," Commun. ACM, vol. 29, no. 12, pp. 1170–1183, Dec. 1986.

[19] B. R. Gaster and L. Howes, "Can GPGPU Programming Be Liberated from the Data-Parallel Bottleneck?," Computer, vol. 45, no. 8, pp. 42–52, Aug. 2012.

[20] T. Stützle and M. Dorigo, "A short convergence proof for a class of ant colony optimization algorithms," IEEE Trans. Evol. Comput., vol. 6, no. 4, pp. 358–365, Aug. 2002.

[21] W. J. Gutjahr, "Mathematical runtime analysis of ACO algorithms: survey on an emerging issue," Swarm Intell., vol. 1, no. 1, pp. 59–79, Oct. 2007.

[22] B. Bullnheimer, G. Kotsis, and C. Strauß, "Parallelization Strategies for the Ant System," in High Performance Algorithms and Software in Nonlinear Optimization, vol. 24, R. Leone, A. Murli, P. M. Pardalos, and G. Toraldo, Eds. Boston, MA: Springer US, 1999.

[23] P. Delisle, M. Gravel, M. Krajecki, C. Gagné, and W. L. Price, "Comparing parallelization of an ACO: message passing vs. shared memory," in Hybrid Metaheuristics, vol. 3636, M. J. Blesa, C. Blum, A. Roli, and M. Sampels, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 1–11.

[24] I. Ellabib, P. Calamai, and O. Basir, "Exchange strategies for multiple Ant Colony System," Inf. Sci., vol. 177, no. 5, pp. 1248–1264, Mar. 2007.

[25] [25] M. Manfrin, M. Birattari, T. Stützle, and M. Dorigo, "Parallel Ant Colony Optimization for the Traveling Salesman Problem," in Proceedings of the 5th International Conference on Ant Colony Optimization and Swarm Intelligence, Berlin, Heidelberg, 2006, pp. 224–234.

[26] J. Jun Ouyang and G.-R. Gui-Rong Yan, "A multi-group ant colony system algorithm for TSP," in Machine Learning and Cybernetics, 2004. Proceedings of 2004 International Conference on, 2004, vol. 1, pp. 117–121.

[27] M. Pedemonte, S. Nesmachnow, and H. Cancela, "A survey on parallel ant colony optimization," Appl. Soft Comput., vol. 11, no. 8, pp. 5181–5197, Dec. 2011.

[28] T. Stützle, "Parallelization strategies for Ant Colony Optimization," in Parallel Problem Solving from Nature — PPSN V, vol. 1498, A. E. Eiben, T. Bäck, M. Schoenauer, and H.-P. Schwefel, Eds. Berlin/Heidelberg: Springer-Verlag, 1998, pp. 722–731.

[29] H. Bai, D. Ouyang, X. Li, L. He, and H. Yu, "MAX-MIN Ant System on GPU with CUDA," in 2009 Fourth International Conference on Innovative Computing, Information and Control (ICICIC), 2009, pp. 801–804.

[30] J. M. Cecilia, J. M. García, A. Nisbet, M. Amos, and M. Ujaldón, "Enhancing data parallelism for Ant Colony Optimization on GPUs," J. Parallel Distrib. Comput., vol. 73, no. 1, pp. 42–51, Jan. 2013.

[31] A. Delévacq, P. Delisle, M. Gravel, and M. Krajecki, "Parallel Ant Colony Optimization on Graphics Processing Units," J. Parallel Distrib. Comput., vol. 73, no. 1, pp. 52–61, Jan. 2013.

[32] K. Kobashi, A. Fujii, T. Tanaka, and K. Miyoshi, "Acceleration of Ant Colony Optimization for the Traveling Salesman Problem on a GPU," 2011.

[33] O. Nitica, "A Parallel Ant Colony Optimization Algorithm for the Travelling Salesman Problem: Improving Performance Using CUDA," Bachelor thesis, University of Delaware, 2011.

[34] K. Tantawy, "Ant Colony Optimization Parallel Algorithm for GPU," Carleton University, Honours Project COMP 4905, Apr. 2011.

[35] A. Uchida, Y. Ito, and K. Nakano, "An Efficient GPU Implementation of Ant Colony Optimization for the Traveling Salesman Problem," in 2012 Third International Conference on Networking and Computing (ICNC), 2012, pp. 94–102.

[36] M. Dorigo, Ant colony optimization. Cambridge, Mass: MIT Press, 2004.

[37] J. S. Angelo, D. A. Augusto, and H. J. C. Barbosa, "Strategies for Parallel Ant Colony Optimization on Graphics Processing Units," in Ant Colony Optimization - Techniques and Applications, H. J. C. Barbosa, Ed. InTech, 2013.

[38] A. Munshi, Ed., OpenCL programming guide. Upper Saddle River, NJ: Addison-Wesley, 2012.

[39] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with CUDA," Queue, vol. 6, no. 2, p. 40, Mar. 2008.

[40] F. Darema, D. A. George, V. A. Norton, and G. F. Pfister, "A single-program-multiple-data computational model for EPEX/FORTRAN," Parallel Comput., vol. 7, no. 1, pp. 11–24, Apr. 1988.

[41] "NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110." NVIDIA, 2012.

[42] A. Williams, C++ concurrency in action. Shelter Island, N.Y: Manning, 2012.

[43] "multicoreware / cppamp-driver-ng / wiki / Home — Bitbucket." [Online]. Available: https://bitbucket.org/multicoreware/cppamp-driver-ng/wiki/Home. [Accessed: 29-Jun-2014].

[44] N. M. Josuttis, The C++ standard library: a tutorial and reference, 2nd ed. Upper Saddle River, NJ: Addison-Wesley, 2012.

[45] D. Vandevoorde, C++ templates: the complete guide. Boston, MA: Addison-Wesley, 2003.

[46] A. A. Stepanov, Elements of programming. Upper Saddle River, NJ: Addison-Wesley, 2009.

[47] B. R. Gaster and L. Howes, "OpenCL C++," in Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units, New York, NY, USA, 2013, pp. 86–95.

[48] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, Numerical recipes: the art of scientific computing, 3rd ed. Cambridge, UK ; New York: Cambridge University Press, 2007.

[49] J. J. Bentley, "Fast Algorithms for Geometric Traveling Salesman Problems," ORSA J. Comput., vol. 4, no. 4, pp. 387–411, Nov. 1992.

[50] A. Blazinskas and A. Misevicius, "Combining 2-opt, 3-opt and 4-opt with K-swap-kick Perturbations for the Traveling Salesman Problem," in Proceedings of the 17th International Conference on Information and Software Technologies, IT 2011, Kaunas, Lithuania, 2011.

[51] M. L. Fredman, D. S. Johnson, L. A. Mcgeoch, and G. Ostheimer, "Data Structures for Traveling Salesmen," J. Algorithms, vol. 18, no. 3, pp. 432–479, May 1995.

[52] G. Reinelt, The traveling salesman: computational solutions for TSP applications. Berlin ; New York: Springer-Verlag, 1994.

[53] "TSPLIB." [Online]. Available: http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/. [Accessed: 07-Mar-2013].

[54] D. S. Johnson and L. A. McGeoch, "The traveling salesman problem: a case study," in Local search in combinatorial optimization, E. H. L. Aarts and J. K. Lenstra, Eds. Princeton: Princeton University Press, 2003, pp. 215–310.

[55] M. G. A. Verhoeven, E. H. L. Aarts, and P. C. J. Swinkels, "A parallel 2-opt algorithm for the Traveling Salesman Problem," Future Gener. Comput. Syst., vol. 11, no. 2, pp. 175–182, 1995.