2014 IEEE Congress on Evolutionary Computation (CEC)
July 6-11, 2014, Beijing, China

Accelerating Ant Colony Optimization-based Edge Detection on the GPU using CUDA

Laurence Dawson and Iain A. Stewart

***Abstract*— Ant Colony Optimization (ACO) is a nature-inspired metaheuristic that can be applied to a wide range of optimization problems. In this paper we present the first parallel implementation of an ACO-based (image processing) edge detection algorithm on the Graphics Processing Unit (GPU) using NVIDIA CUDA. We extend recent work so that we are able to implement a novel data-parallel approach that maps individual ants to thread warps. By exploiting the massively parallel nature of the GPU, we are able to execute significantly more ants per ACO-iteration allowing us to reduce the total number of iterations required to create an edge map. We hope that reducing the execution time of an ACO-based implementation of edge detection will increase its viability in image processing and computer vision.**

I. Introduction

Ant Colony Optimization (ACO) is a nature-inspired population-based metaheuristic that models the behaviour of a colony of ants to solve a variety of optimization problems [1]. *Ant system* (AS) is the simplest ACO algorithm and consists of two stages: construction of solutions; and pheromone update. In the first stage, individual ants iteratively build up solutions to a given problem using heuristic information and via indirect communication with other ants using pheromone trails (also known as *stigmergy*). In the second stage, after solutions are constructed each ant deposits an amount of pheromone proportional to the quality of the solution constructed. The pheromone trails are built up gradually and shape the solutions constructed by ants in future iterations of the algorithm. Dorigo and Stützle note [1] that good solutions are an emergent property of this feedback mechanism and cooperation between ants. There have also been many other alternative ACO algorithms proposed including *elitist* AS, $\mathcal{MAX}-\mathcal{MIN}$ AS (MMAS), *ant colony system* and the *hyper-cube framework for ACO* [1].

ACO has been successfully applied to many different problems including; the Travelling Salesman Problem (TSP) [1], the Quadratic Assignment Problem [2] and, in recent years, to a range of image processing problems including edge detection [3]. Dorigo and Stützle remark [1] that for many applications ACO solutions often rival the best in class.

Edge detection is the process of producing a map of edges from a given input image. The resulting edge maps are an essential component of many computer vision algorithms as they drastically reduce the input data size whilst preserving vital information concerning the edge boundaries [4]. Baterina and Oppus [5] note that traditional approaches to edge detection can be computationally expensive as an exhaustive search is performed via per-pixel convolution to determine the position of edge boundaries from neighbouring pixels.

Nezamabadi-pour et al. [3] moved away from per-pixel convolution and proposed the first mapping of ACO for edge detection by implementing the AS algorithm. In their approach, ants are placed randomly on a given image and move around the image following variations in intensity. The ants then deposit pheromone so as to communicate which edges to follow. This process gradually results in ants grouping around the edges of the image which, in turn, is used to produce an edge map (to which a morphological thinning algorithm may be applied). The process produces high quality edge maps but, as Lu and Chen [6] note, overall it can be slow due to redundancy in the search.

NVIDIA CUDA is a parallel programming architecture for developing general purpose applications for direct execution on the GPU [7]. CUDA exposes the GPU's massively parallel architecture so that parallel code can be written to execute much faster than its optimized sequential counterpart. CUDA compatible GPUs can now be found in a wide range of devices from desktop computers to laptops and more recently mobile devices such as tablet computers and phones [8]. CUDA applications will automatically scale to utilize the number of available *CUDA cores*. Although CUDA abstracts the underlying architecture of the GPU, utilising and scheduling the GPU for maximum potential speedups is non-trivial. Due to the intrinsically parallel nature of ACO, there has been substantial research efforts ([9], [10], [11], [12], [13]) to develop parallel ACO implementations for execution on the GPU so as to exploit the architecture for significant performance gains.

In this paper we present the first parallel ACO edge detection implementation for execution on the GPU using NVIDIA CUDA. Our aim is to improve the runtime, and thus viability of the algorithm, rather than introduce fundamental changes to the design of the algorithm (however, the nuances of CUDA usually mean that algorithm amendments occur so as to secure an efficient implementation). Our implementation is able to match the quality of the edge maps produced by the implementation by Nezamabadi-pour et al [3]. By extending previous research [9], [10], we utilise a data-parallel approach: we are able to show that a previously proposed data-parallel ant mapping is still applicable for image processing despite conventional GPU image processing algorithms using the pixel-to-thread mapping. The primary contribution of this paper is the novel mapping of individual ants to CUDA thread warps so as to pack multiple ants into a single thread block whilst maintaining a data-parallel approach. This approach yields the best results and speedups of around 150x against an optimized sequential counterpart.

978-1-4799-1488-3/14/$31.00 ©2014 IEEE

II. Background

In this section we define ACO, AS and how AS can be applied to edge detection. For additional details regarding ACO we direct the reader to the extensive original works of Dorigo and Stützle [1], [14], [15], [16], [17].

As was previously mentioned ACO is a nature-inspired population-based meta-heuristic that models the behaviour of a colony of ants to solve various optimization problems. The AS algorithm arose after three initially-proposed ant algorithms [1] and consists of two main stages (see Fig. 1): ant solution construction; and pheromone update. These two stages are repeated until a termination condition is met, e.g., until after a set number of iterations have been undertaken or or until solutions of a predetermined quality have been constructed. The colony of ants contains m artificial ants. In the tour construction phase, these m ants construct solutions independently of each other.

```
procedure ACOMetaheuristic
  set parameters, initialize pheromone levels
  while (termination condition not met) do
    construct ants' solutions
    update pheromones
  end
end
```

Fig. 1: Overview of the AS algorithm

A. Algorithm setup

Nezamabadi-pour et al. [3] describe the first mapping of ACO to edge detection utilizing the AS algorithm. An image can be easily held in memory using a 2D array representation of a graph. This 'grid-based' graph serves as the artificial landscape for each ant to explore and find edges. To initialize the algorithm, the input image is read, converted to graph and each ant is randomly placed on a node. The pheromone trails (also known as the pheromone matrix) are seeded with the value 0.0001. Each ant has a limited memory of nodes it is not allowed to visit again so as to ensure that ants do not get stuck following the same trail repeatedly. A move is valid if the node is not currently in the ant's memory.

B. Solution construction

Once initialized each ant independently moves to a neighbouring node (horizontally, vertically or diagonally). Nezamabadi-pour et al. [3] consider one complete iteration of the algorithm to consist of each ant performing one step (a move from one node to another) and updating the pheromone matrix. In ACO ants decide how to construct solutions using the random proportional rule where each available move is assigned a probability and where the selection of a move is proporitionate to the probability [1]. Nezamabadi-pour et al. [3] adapted the random proportional rule for edge detection by considering two cases. In the first case, we consider that the ant has visited all of the neighbouring nodes and no valid move is currently available: in this case we simply randomly move the ant to another position on the graph. In the second case (where there are valid moves available), each ant surveys the eight neighbouring nodes (see Fig. 2) so as to determine where next to move. The probability of visiting a neighbouring node (i, j) from (r, s) so as to perform a valid move is (or each ant) defined as:

$$p^k_{(r,s),(i,j)} = \frac{[\tau_{ij}]^\alpha [\eta_{ij}]^\beta}{\sum_u \sum_v [\tau_{uv}]^\alpha [\eta_{uv}]^\beta} \tag{1}$$

where: τ_{ij} is the amount of pheromone currently deposited on the pixel (i, j); η_{ij} is is the visibility of the node (i, j); and α and β are user-defined parameters to control the influence of τ_{ij} and η_{ij}. The visibility of the pixel (i, j) is defined as:

$$\eta_{ij} = \frac{1}{I_{Max}} \times Max \begin{bmatrix} |I(i-1, j-1) - I(i+1, j+1)|, \\ |I(i-1, j+1) - I(i+1, j-1)|, \\ |I(i, j-1) - I(i, j+1)|, \\ |I(i-1, j) - I(i+1, j)| \end{bmatrix} \tag{2}$$

where I is the intensity of a pixel. By applying the simple pixel mask (see Fig. 2) to each pixel we can determine the variation in intensities between pixels. Nezamabadi-pour et al. [3] note that edge pixels should have the highest visibility. This in turn directly increases the chance of the pixel being selected by the random proportional rule.

i-1,j-1	i-1,j	i-1,j+1
i,j-1	i,j	i,j+1
i+1,j-1	i+1,j	i+1,j+1

Fig. 2: Surrounding neighbour pixels and valid moves from position (i, j) (providing none of the surrounding pixels have recently been visited)

C. Pheromone update

Once each ant has completed its move (or has been randomly relocated if there were no moves available), the pheromone matrix must be updated. To avoid stagnation of the colony, the pheromone level of every node is first evaporated according to the user-defined *evaporation rate* ρ. So, each pheromone level τ_{ij} becomes:

$$\tau_{ij} \leftarrow (1 - \rho)\tau_{ij}. \tag{3}$$

Over time this allows nodes that are seldom visited to be forgotten and potentially excluded from the final edge map which is generated from the pheromone matrix. After

evaporation the pheromone matrix must be updated with the last move of each ant so as to influence the subsequent iterations of the algorithm. Each ant deposits an amount of pheromone on the last node visited so that the pheromone level τ_{ij} becomes:

$$\tau_{ij} \leftarrow \tau_{ij} + \sum_{k=1}^{m} \Delta\tau_{ij}^{k}, \tag{4}$$

D. Termination conditions

The algorithm is executed for a set number of iterations after which an edge map is generated from the pheromone matrix [3]. At this stage it is important to note that the number of iterations is influenced by the number of ants. If we allow more ants to explore the graph simultaneously then we can reduce the total number of iterations required.

E. CUDA and the GPU

In 2007 NVIDIA introduced CUDA, a parallel architecture designed for executing applications on both the CPU and GPU, along with a new generation of GPUs (G80) with dedicated silicon to facilitate future parallel programming [18]. CUDA allows developers to run blocks of code, known as kernels, directly on the GPU using a parallel programming interface and using familiar programming languages (such as C). This broke away from the traditional approach of using complex graphics interfaces such as Cg to harness the power of the GPU for general purpose computation.

1) Blocks and threads: The typical architecture of a CUDA-compatible GPU consists of an array of streaming multiprocessors (SM), each containing a subset of streaming processors (SP). When a kernel method is executed, the execution is distributed over a grid of blocks each with their own subset of parallel threads. Each thread within a block is able to communicate with other threads in that block via *shared memory*. Within a block, threads execute in parallel in smaller sub-blocks known as warps, each containing 16-32 threads. There is no guarantee as to the order the warps will execute in; however, within a warp threads can communicate directly with each other using warp level primitives.

2) Memory types: CUDA exposes a set of different memory types to developers, each with unique properties that must be exploited to maximize performance. The first type is *register memory*. Registers are the fastest form of storage and each thread within a block has access to a set of fast local registers that exist on-chip. However, each thread can only access its own registers and as the number of registers per block is limited, blocks with many threads will have fewer registers per thread. For inter-thread communication within a block, shared memory must be used. Shared memory also exists on-chip and is accessible to all threads within the block but is slower than register memory. Newer *Kepler* CUDA GPUs can also communicate directly with other threads within their warp using the new *shfl* method [19]. For inter-block communication and larger data sets, threads have access to *global (DRAM)*, *constant* and *texture memory*. Ever since Fermi, access to global memory is now cached using L1 and L2 caches. Texture and constant memory also benefit from caching, but the initial load will be significantly slower than accessing shared or register memory. When designing a kernel of code for parallel CUDA execution, it is important to fully and properly use the three main memory types (register, shared and global). As Kirk and Hwu note [18], global memory is slow but often large, whereas shared memory is fast but extremely limited (up to 64kB). A common optimization is to load subsets of global memory into shared memory; this approach is known as *tiling*.

III. Related work

In this section we will review existing contributions to edge detection using ACO. As we cannot include a complete review of all ACO edge detection papers we will give a brief overview of the most significant contributions to date and of the most significant relevance to our research. At the time of writing we were unable to find any other existing research into parallel edge detection algorithms using ACO with NVIDIA CUDA or using other GPU frameworks. For completeness we will also briefly cover alternative GPU accelerated edge detection implementations and GPU accelerated ACO algorithms for the TSP.

A. Edge detection using Ant Colony Optimization

As was previously mentioned, Nezamabadi-pour et al. [3] were the first to propose edge detection via ACO. Their main contribution was the novel mapping of standard ACO components (random proportional rule and pheromone update) to edge detection using the simple AS algorithm. The edge maps produced were of high quality and the input parameters for the algorithm did not require modification for different images. Implementation details and execution times for their algorithm were not included however Lu and Chen [6] have subsequently remarked that this initial offering was slow.

Lu and Chen [6] provide an alternative method and focus their efforts on repairing broken edges and reducing the work done by the algorithm. When compared against the implementation by Nezamabadi-pour et al. [3], their results show that they were able to produce higher quality edge maps in around half the time. However, the time required to produce a single edge map was around 1 minute.

Tian et al. [20] detail an improved ACO edge detection algorithm based on the works of Nezamabadi-pour et al. [3]. Their approach differs by allowing ants to make multiple moves per each iteration and to update the pheromone levels after each of these moves and again after all ants have moved. They detail that the execution time of their implementation was also around 1 minute. They conclude that a parallel ACO algorithm could be effectively utilized to decrease the computational load and thus reduce the total execution time.

Many additional papers have subsequently been presented however mainly focussing on improving the viability of the algorithm by increasing the quality of the edge maps produced over reducing the execution time of the algorithm.

B. Other edge detection methods on the GPU

The Canny edge detection algorithm [4] produces extremely high quality edge maps and results in more complete edges than alternative algorithms such as Prewitt or Sobel due to the inclusion of the hysteresis step. Luo and Duraiswami [21] were the first to present a GPU implementation of the Canny algorithm using NVIDIA CUDA. Their implementation moved the entire execution of the algorithm to the GPU which yielded significant speedups over the optimized sequential counterpart. Luo and Duraiswami [21] detail their GPU implementation uses a pixel-to-thread mapping. Ogawa et al. [22] build upon the work of Luo and Duraiswami [21] and also present a GPU implementation of the Canny algorithm using a simple pixel-to-thread mapping.

Simpler edge detection algorithms such as using the Prewitt and Sobel operators have also been implemented in parallel by NVIDIA [23]; the process of applying one of the operators via convolution in parallel is simple and uses pixel-to-thread mapping.

C. ACO on the GPU

In our previous work [9] we presented a highly parallel GPU implementation of ACO for solving the TSP using CUDA. By extending the work of Cecilia et al. [11] and Delvàcq et al. [12] we adopted a data-parallel approach that maps individual ants to thread blocks. Our solution executed up to 82x faster than the sequential counterpart and up to 8.5x faster than the best existing parallel GPU implementation. We were able to improve these results by use of a candidate set [10] to restrict the number of cities available during tour construction. For a complete review of all ACO GPU literature to date we direct readers to [9]. We extend our previous techniques in what follows.

IV. Implementation

In this section we present a parallel implementation a ACO edge detection algorithm for execution on the GPU using NVIDIA CUDA. We execute each stage of the algorithm on the GPU to avoid unnecessary memory transfers.

Building upon our previous contributions [9], [10], we present a new parallel approach mapping multiple ants to each thread block on a warp level. Cecilia et al. [11] have previously shown that mapping individual ants to each CUDA thread is not effective and following a data-parallel mapping of one thread per thread block yields improved execution times. This speedup is due to reducing warp serialization as a result of eliminating thread divergence caused by each thread following its own path. When a warp is serialized the speedup is dramatically reduced and NVIDIA recommends, as best practice, that this should generally be avoided [7].

Edge detection differs from the TSP as each ant can potentially move to any city in the graph of size k; however, with ACO based edge detection each ant can only ever move to any of the 8 neighbouring pixels (see Fig. 2). The number of valid moves will also decrease as an ant is not permitted to revisit a pixel for a set number of iterations (this is to avoid the ant becoming stuck and not fully exploring the image). To accommodate the difference in the number of potentially valid moves, we map each ant to a warp of threads and execute multiple ants per thread block. This allows us to ensure all threads within the warp still follow the same execution path (avoiding warp serialization) but also to execute more ants per thread block thus reducing the total number of blocks required. Whilst we could have simply utilized the previous data-parallel mapping, this would have left most of the threads in each block idle during execution and increased execution time due to using more thread blocks. As was previously shown by Cecilia et al. [11], mapping a single thread to an individual results in slower execution times and our initial experiments also found this to be true for our edge detection implementation. As a result the following section will only document our warp-level ant mapping which consistently resulted in the best speedups and lowest execution times for all of the standard test images.

A. Algorithm setup

As Nezamabadi-pour et al. [3] note, an input image can be loaded into a 2D array. The color image is then converted to greyscale using the algorithm shown in Fig. 3. Novak and Shafer [24] note, around 90% of edges in an image can be found just using the greyscale values and this approach produces high quality edge maps.

The image array is later used by the random proportional rule for determining the visibility of a pixel which in turn alters the probability of an ant deciding to move to the pixel. However, as this input image remains static throughout each iteration of the algorithm we can significantly reduce the computational load of the algorithm by pre-processing the pixel visibilities. The visibility of a pixel is calculated using the variation of intensity around a given point. With existing implementations each ant must calculate the visibility of all pixels in the neighbourhood of a pixel for each iteration. This is a costly operation and unnecessary as the image data does not change. After loading the image data into an array, we calculate all pixel visibilities and save the results to a second array in global memory on the GPU. This array is then used by each ant when calculating the probability of visiting a neighbouring pixel thus replacing eight slow global memory lookups with a single lookup for each pixel.

```
procedure ColorToGreyscale (red, green, blue)
    return (red >>2) + (green >>1) + (blue >>2);
end
```

Fig. 3: Calculating the greyscale value for a color pixel

A pheromone matrix is allocated in global memory on the GPU and artificially seeded with the value 0.0001. A second pheromone matrix is also allocated in global memory. As each ant is executed in parallel, it can potentially deposit pheromone to the pheromone matrix before all ants have

made their next move. To accommodate this, after evaporation the new values are written to the second matrix. Each ant then deposits an amount of pheromone on the second matrix after constructing their solution using an atomic add operation. In the next iteration of the algorithm the two pheromone matrices are swapped thus allowing ants to deposit without impacting the current iteration.

Finally we allocate an array containing the ants. For each ant we maintain the current position on the image, the current iteration and a small array for the ant memory. As we are operating on the warp level we set the length of the memory array to 32 previous locations matching the size of the warp. By maintaining the current iteration, we are able to treat the memory array as a circular array using basic modulo arithmetic. For example on iteration 48 we would index position 48 % 32. This allows us to maintain a fast FIFO queue of previous locations on the image without the need for additional data structures. Before we enter the solution construction phase, the ants are randomly placed around the image. Lu and Chen [6] suggest an alternative to randomly placing the ants is to place the ants on the end points of edges extracted using alternative algorithms. However we will use the simpler random placement for our implementation.

B. Solution construction

In Section II we gave an overview of solution construction phase of the AS algorithm outlined by Nezamabadi-pour et al. [3]. In this section we will describe our parallel mapping of the algorithm to the GPU using CUDA.

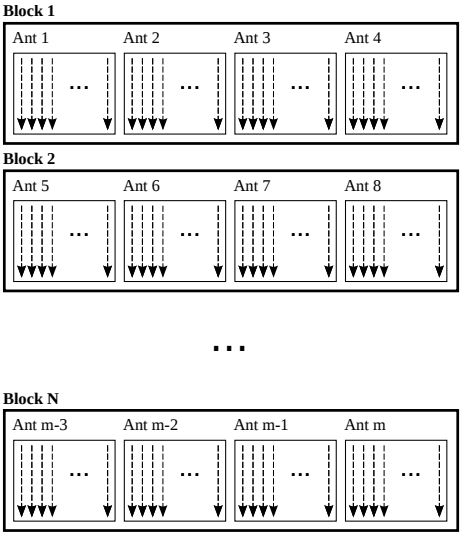


Fig. 4: Mapping individual ants to each warp of threads

Our primary contribution in this paper is the novel ant-to-warp mapping for the tour construction phase. As was previously mentioned, mapping a single ant to a thread is ineffective and leads to warp serialization. Mapping a single ant to each thread block is a wasteful use of the GPU as most threads will be idle for each iteration. We found experimentally that packing four ants into a thread block (using a total of 128 threads) and operating on a warp level yielded the best results. In Fig. 4 we illustrate our ant warp mapping for solution construction that is key to the speedup attained. In Fig. 5 we give an overview of the entire parallel solution construction phase performed by each ant within a thread block before detailing each step individually.

procedure SolutionConstruction
  Cache the surrounding pixel visibility data
  Calculate the probability of visiting local pixels
  Check if the local pixels have already been visited
  Perform reduction on the pixel probabilities
  Perform roulette wheel selection on the probabilities
  Update the ants current position
end

Fig. 5: An overview of solution construction

First the pre-calculated visibility values (as previously described) of the 8 neighbouring pixels are cached into shared memory. As we are operating on a warp level, the remaining 24 threads load in the same data to avoid warp divergence. As the values are cached in the L1 cache, this operation is faster than branching the warp.

After the visibility data is cached to shared memory, each thread loads the previously visited position from the ant's memory into a local register. As the size of the memory is limited to 32 previous positions each thread will load one value from the array. For example, thread 9 will load position 9 from the array and so on. This will later allow the warp to quickly check if a position has recently been visited.

Each of the first 8 threads in the warp then calculate the probability of visiting one of the the neighbouring pixels using the random proportional rule described in Section II. The probabilities calculated are saved to an array in shared memory. Each thread in the warp then checks if the previously visited pixel cached in its register is a valid move. If a thread finds that the move has recently been made it replaces the probability of visiting that pixel with 0. A warp-level reduction is then performed on the probability array returning the total of all the probability values. At this point the first thread checks if the total value returned is greater than 0 and if not the thread picks a new random location for the ant to move to (also saving this to the ants memory).

TABLE I: Roulette wheel selection

input	reduced	normalized	range
0.1	0.1	0.1	> 0.0 & ≤ 0.1
0.3	0.4	0.25	> 0.1 & ≤ 0.25
0.2	0.6	0.375	> 0.25 & ≤ 0.375
0.8	1.4	0.875	> 0.375 & ≤ 0.875
0.2	1.6	1.00	> 0.875 & ≤ 1.0

If the total is greater than 0 the first thread then calculates a random number in the range of 0 to 1 saving this to shared memory. The first 8 threads then apply roulette wheel selection (proportionate selection) to select the next pixel to move to. Using the previously reduced probability array in shared memory (and total value previously returned) each

thread normalizes the probability of its respective pixel bringing the probability into the range of 0 to 1 (see Table I). After the values are normalized each of the 8 threads then checks if the previously generated random number lies within the range for that pixel. If the thread determines that its pixel has been selected then the ant's current position is updated. This parallel implementation thus handles both cases for solution construction outlined by Nezamabadi-pour et al. [3].

C. Pheromone update

The second stage of the AS algorithm is pheromone update which consists of two stages: pheromone evaporation; and pheromone deposit. The pheromone update stage represents a very small proportion of the total execution time and thus is not the main focus of this paper. As we previously noted [9], the pheromone evaporation stage (see equation 3) is trivial to parallelize as all points on the matrix are evaporated by a constant factor ρ. We adopt the same efficient parallel strategy previously demonstrated [9] where a single thread block is launched which maps each thread to position on the pheromone matrix and decreases the value using ρ. A tiling strategy is used to ensure full coverage of all positions on the matrix. Each ant then deposits an amount of pheromone proportional to the quality of its move onto the pheromone matrix. As previously noted, we utilize two pheromone matrices (alternating between primary and secondary) to ensure that ants deposit to a matrix not currently being read by other ants. Each ant deposits an amount of pheromone to the second pheromone matrix using the operation *atomicAdd()* (for thread safety) which takes the previous value on the matrix and adds the new value.

V. Results

In this section we will discuss our experimental setup, algorithm parameters chosen, quality of the edge maps obtained via our parallel implementation and finally the execution times observed.

A. Experimental setup

For testing our implementation we use an NVIDIA GTX 580 GPU (Fermi) and an Intel i7 950 CPU (Bloomfield). The GPU contains 580 CUDA cores and has a processor speed of 1544 MHz. As the card is from the Fermi generation, it uses 32 threads per warp and up to 1024 threads per thread block with a maximum shared memory size of 64 Kb. The CPU has 4 cores which support up to 8 threads with a clock speed of 3.06 GHz. Our implementation was written and compiled using the latest CUDA tooklkit (v5.0) for C and executed under the latest stable Ubuntu release (v13.10).

B. Algorithm parameters

To ensure a fair comparison of the edge maps produced via our implementation, we use the same parameters as defined by Nezamabadi-pour et al. [3] with the exception of modifying the number of ants and iterations. Nezamabadi-pour et al. define the number of ants and iterations to be proportionate to the root of the size of the image. For example, an image of size 512×512 would have a total of 512 ants performing 512 iterations before producing an edge map (a total of 262144 moves). We found experimentally that executing more ants per iteration for fewer iterations yielded edge maps of comparable quality and was a better fit for the parallel architecture. There is an implicit overhead when scheduling a kernel for execution on the GPU and by increasing the number of ants (thus increasing the number of thread blocks) we can reduce the number of kernel executions. This approach will also scale automatically to CUDA devices with more CUDA cores which will further decrease the execution time. For a 512×512 image we use 3000 ants for a maximum of 50 iterations (a total of 150000 moves). The remaining algorithm parameters are as follows:

- α = 2.5
- β = 2
- ρ = 0.04
- Ant memory length = 32
- Edge threshold = mean image intensity

C. Solution quality

To ensure our implementation provided edge maps comparable to the original algorithm by Nezamabadi-pour et al. [3], we tested against the standard test images (*Lena, Peppers etc.*). Our parallel solution was able to match and often improve the quality of edge maps produced. In Fig. 6 we show the edge maps produced by the Sobel operator, the Canny edge detector and our parallel ACO implementation. The edge maps produced for Canny and Sobel were generated via the Image Processing toolbox in MATLAB [25].

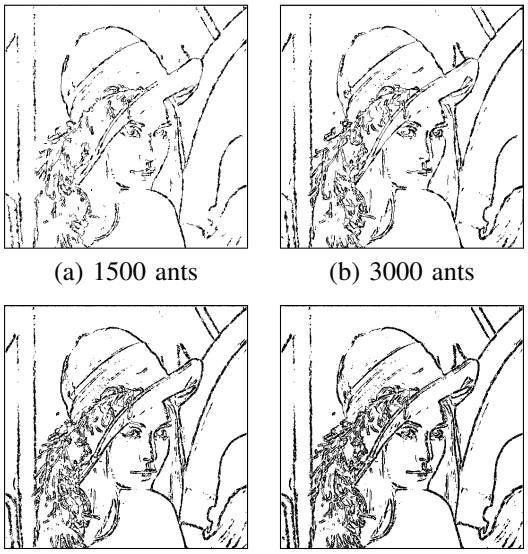
(a) Original (b) Sobel

(c) Canny (d) Our solution

Fig. 6: A comparison of edge maps produced by the Sobel, Canny and our parallel ACO edge detection algorithms

We found experimentally that by increasing the number of ants (whilst keeping the iteration count static) we were able to increase the thickness of the edges produced. Over time

ants settle on the major edges in the image due to deposits on the pheromone matrix. As more ants are added to the image this effect is amplified creating a stylized effect. In Fig. 7 we show the edge maps produced when using 1500 ants, 3000 ants (standard), 4500 and 6000 ants.



(a) 1500 ants (b) 3000 ants

(c) 4500 ants (d) 6000 ants

Fig. 7: The effect on edge thickness when alternating the number of ants in a 512x512 image

D. Benchmarks

In Section IV we detailed the various mappings for ACO on the GPU. The simplest of the mappings uses a single CUDA thread for each ant, the second mapping uses a whole CUDA thread block for each ant and the third mapping uses a single thread warp per ant (with multiple ants per CUDA block). Our results (shown in Table II) show that the different approaches yield significantly different results. The execution times detailed are for both the solution construction and pheromone update stages of the AS algorithm.

TABLE II: Average execution times (ms) when varying the number of threads per block

Threads per block	Ant-to-thread	Ant-to-block	Ant-to-warp
64	62.679	21.435	10.693
128	63.025	18.233	6.531
256	62.858	18.433	8.884
512	62.601	18.431	9.308
768	64.845	18.973	9.597

The first mapping (although the simplest to implement) produced the worst results and varying the number of threads per block did little to change this. As Cecilia et al. [11] note, this simple mapping is not suited to ACO as the solution construction phase results in warp divergence which increases the overall execution time. As expected the second data-parallel mapping produced significantly better results and executed in around a third of the time of the first mapping. The third approach which utilized our ant-to-warp mapping consistently produced the best results. The mapping performed best when using 128 threads (4 ants using 4 warps of 32 threads per CUDA thread block).

In our ant-to-warp implementation each thread in the warp caches a value of a recently visited city to shared memory. This later allows the warp to quickly check if a neighbouring pixel is still valid or has been visited recently. As Fermi generation GPUs automatically cache values to the L1 cache we observed the execution times of manually caching values to the shared memory versus relying on the L1 cache (see Fig. 8). The results show that manually caching the values is still considerably faster and necessary to obtain the best possible speedups.

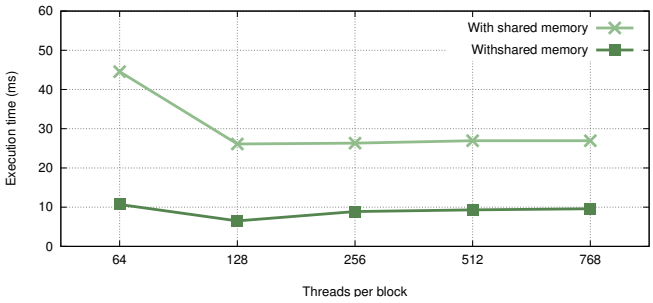


Fig. 8: Observed execution speeds for the ant-to-warp mapping with/without caching visited positions to shared memory

An ant placed on an input image has no perceivable concept of difficulty and will explore an image irrespective of the number of edges on the image. As a result we found that unlike with other edge detection algorithms (such as the Canny edge detector) varying the input image had little effect on the overall execution time of the algorithm.

Finally, we compared the results of executing our ant-to-warp mapping implementation against an optimized sequential counterpart. We found that the sequential implementation took just under 1000ms to execute compared against 6.531ms for the best thread configuration for our parallel CUDA version. This represents around a 150x speedup for our GPU implementation against the optimized CPU implementation.

VI. Conclusions

In this paper we present the first parallel ACO edge detection implementation for execution on the GPU. By extending our previous contributions we are able to adapt the data-parallel GPU ACO mapping for edge detection. By harnessing the massively parallel nature of the GPU we reduced the number of iterations required to produce the edge map and increased the number of ants per iteration. Our implementation is able to match the quality of edge maps produced by the sequential implementation and executed up to 150x faster. Our future work will aim to implement parallel versions of other ACO algorithms so as to increase the quality of edge maps produced.

ACKNOWLEDGEMENTS

The authors would like to thank Hossein Nezamabadi-pour and Fateme Darake for providing the MATLAB source code of the implementation in [3]. This allowed us to verify the validity of our GPU implementation and ensure we were able to match the quality of edge maps generated.

REFERENCES

[1] M. Dorigo and T. Stützle, *Ant Colony Optimization*. MIT Press, 2004.

[2] V. Maniezzo and A. Colorni, "The Ant System applied to the Quadratic Assignment Problem," *IEEE Transactions on Knowledge and Data Engineering*, vol. 11, no. 5, pp. 769–778, 1999.

[3] H. Nezamabadi-pour, S. Saryazdi, and E. Rashedi, "Edge Detection using Ant Algorithms," *Soft Computing*, vol. 10, no. 7, pp. 623–628, 2006.

[4] J. Canny, "A Computational Approach to Edge Detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-8, no. 6, pp. 679–698, 1986.

[5] A. V. Baterina and C. Oppus, "Image Edge Detection Using Ant Colony Optimization," *WSEAS Transactions on Signal Processing*, vol. 6, no. 2, pp. 58–67, 2010.

[6] D.-S. Lu and C.-C. Chen, "Edge detection improvement by ant colony optimization," *Pattern Recognition Letters*, vol. 29, no. 4, pp. 416–425, 2008.

[7] NVIDIA, "CUDA C Programming Guide," http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (last accessed 10/01/2014).

[8] ——, "CUDA for ARM Platforms is Now Available," https://developer.nvidia.com/content/cuda-arm-platforms-now-available (last accessed: 10/01/2014).

[9] L. Dawson and I. Stewart, "Improving Ant Colony Optimization performance on the GPU using CUDA," in *IEEE Congress on Evolutionary Computation (IEEE-CEC'13)*. IEEE, 2013, pp. 1901–1908.

[10] L. Dawson and I. A. Stewart, "Candidate Set Parallelization Strategies for Ant Colony Optimization on the GPU," in *13th Int. Conf. on Algorithms and Architectures for Parallel Processing (ICA3PP'13)*, ser. Lecture Notes in Computer Science, vol. 8285. Springer, 2013, pp. 216–225.

[11] J. M. Cecilia, J. M. García, A. Nisbet, M. Amos, and M. Ujaldon, "Enhancing data parallelism for ant colony optimization on GPUs," *Journal of Parallel and Distributed Computing*, vol. 73, no. 1, pp. 42–51, 2013.

[12] A. Delèvacq, P. Delisle, M. Gravel, and M. Krajecki, "Parallel ant colony optimization on graphics processing units," *Journal of Parallel and Distributed Computing*, vol. 73, no. 1, pp. 52–61, 2013.

[13] J. Fu, L. Lei, and G. Zhou, "A parallel Ant Colony Optimization algorithm with GPU-acceleration based on All-In-Roulette selection," in *Third Int. Workshop on Advanced Computational Intelligence (IWACI)*. IEEE, Aug. 2010, pp. 260–264.

[14] M. Dorigo, "Optimization, learning and natural algorithms," Ph.D. dissertation, Dipartimento di Elettronica, Politecnico di Milano, Milan, Italy, 1992.

[15] M. Manfrin, M. Birattari, T. Stützle, and M. Dorigo, "Parallel ant colony optimization for the traveling salesman problem," in *Fifth Int. Workshop on Ant Colony Optimization and Swarm Intelligence (ANTS)*, ser. Lecture Notes in Computer Science, vol. 4150. Springer, 2006, pp. 224–234.

[16] T. Stützle, "Parallelization Strategies for Ant Colony Optimization," in *Fifth Int. Conf. on Parallel Problem Solving from Nature (PPSN-V)*, ser. Lecture Notes in Computer Science, vol. 1498. Springer-Verlag, 1998, pp. 722–731.

[17] T. Stützle and H. H. Hoos, "MAX-MIN ant system," *Future Generation Computer Systems*, vol. 16, no. 9, pp. 889–914, 2000.

[18] D. Kirk and W.-M. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., 2010.

[19] NVIDIA, "Inside Kepler," http://developer.download.nvidia.com/GTC/PDF/GTC2012/PresentationPDF/S0642-GTC2012-Inside-Kepler.pdf (last accessed 18/01/2014).

[20] J. Tian, W. Yu, and S. Xie, "An ant colony optimization algorithm for image edge detection," in *IEEE Congress on Evolutionary Computation (IEEE-CEC'08)*. IEEE, 2008, pp. 751–756.

[21] Y. Luo and R. Duraiswami, "Canny Edge Detection on NVIDIA CUDA," in *Computer Vision and Pattern Recognition Workshops (CVPRW'08)*. IEEE, 2008, pp. 1–8.

[22] K. Ogawa, Y. Ito, and K. Nakano, "Efficient Canny Edge Detection Using a GPU," in *First Int. Conf. Networking and Computing (ICNC*. IEEE, 2010, pp. 279–280.

[23] NVIDIA, "Image Convolution with CUDA," http://goo.gl/ybdGbQ (last accessed 10/01/2014).

[24] C. Novak and S. Shafer, "Color edge detection," in *Proc. DARPA Image Understanding Workshop*, vol. 1. Morgan Kaufmann, 1987, pp. 35–37.

[25] MathWorks, "Image Processing Toolbox," http://www.mathworks.co.uk/products/image/ (last accessed 10/01/2014).