A Parallel Strategy for a Genetic Algorithm in Routing Wavelength Assignment Problem Using GPU with CUDA

Esdras La-Roque[1], **Cassio T. Batista**[1], **Josivaldo de S. Araújo**[1]

[1]Institute of Exact and Natural Sciences
Federal University of Pará – Belém – Brazil

{esdras,cassiotb,josivaldo}@ufpa.br

Abstract. *This paper presents a parallel strategy with a heuristic approach to reduce the execution time bottleneck of a routing and wavelength assignment problem in wavelength-division multiplexing networks of a previous work that uses a sequential genetic algorithm. As the parallelization solution, the GPU hardware processing on CUDA architecture and CUDA C programming language were adopted. The results achieved were between 35 and 40 times faster than the sequential version of the genetic algorithm.*

1. Introduction

Genetic algorithms [Silva and Gabriel 2019] have increasingly gained prominence to solve Routing and Wavelength Assignment problems (RWA) in WDM networks (Wavelength-Division Multiplexing), demonstrating satisfactory results regarding request blocking probability when compared to classical algorithms as Dijkstra and Bellman-Ford for the routing subproblem, as well as first-fit, random-fit, for example, for the wavelength assignment subproblem [Teixeira et al. 2017]. However, genetic algorithms take too much time to process [Cantú-Paz 1998], which makes it unfeasible for real world applications. In addition, this limitation makes advances in research difficult for huge WDM networks due to their huge number of nodes and connections.

This article explores the possibility of adopting a parallel execution strategy for a RWA-GA simulation environment by using a GPGPU (General Purpose Graphic Processing Unit) approach with NVIDIA cards which are able to run CUDA programs. Further, this paper discusses the process and strategy used to turn the simulation code originally written in Python language [Teixeira et al. 2017] into C-99 for sequential execution, but still running at the CPU side (no GPU involved), then turn the C-99 code into CUDA C language for running the simulation at the GPU side in a CUDA parallel architecture. A performance evaluation was carried out focusing on processing time of the three implementation codes (Python, C-99, and CUDA C) as well as on the GA correctness for the three code versions, demonstrating that CUDA GPU environment is a feasible parallelization solution for the context of the RWA-GA problems. The simulation experiments were run using NSFNET, the U.S. National Science Foundation (NSF) network topology [Frazer et al. 1996], which contains 14 routing nodes.

2. RWA Problem in WDM Networks

In WDM networks, a number of optical carriers is multiplexed into a single optical fiber using various different wavelengths. For transparent WDM networks, also called all-optical networks, two intrinsic constraints arise: the wavelength-continuity constraint, which states that the same wavelength must be used on each and every physical fiber link of a traversed path [Zang and Jue 2000]; and the distinct wavelength constraint, which says that two connections must not use the same wavelength on the same fiber link [Randhawa and Sohal 2010].

Both constraints occur because of the absence of opto-electro-opto converters on the nodes of all-optical networks, which means the signal remains in the optical domain throughout all communications. Analogously, those restrictions are not considered on opaque networks because, unlike transparent networks, its intermediate nodes do perform optoelectronic conversion before forwarding packets [Ellinas et al. 2004].

The routing and the wavelength assignment subproblems, together, compose the RWA problem in WDM networks. Given a fixed number of wavelengths per link and a set of connection requests, an RWA algorithm for static traffic must maximize the number of optical connections established on the network [Zang and Jue 2000] while attempting to minimize the total number of wavelengths used [Varela and Sinclair 1999].

2.1. Routing

The routing subproblem itself can be classified into three types: fixed routing, fixed-alternate routing, and adaptive routing [Wason and Kaler 2007].

- Fixed routing: given a source-destination pair of nodes, the same predetermined route, typically the shortest one, is always chosen. Dijkstra's algorithm is the main example of the fixed routing approach.
- Fixed-alternate routing: a set of routes is kept in a routing table instead of a single route for each SD pair. Yen's algorithm, also known as K-shortest path algorithm, is an example of fixed-alternate routing.
- Adaptive routing: the route for a SD pair is calculated dynamically based on the current configuration of all connections in progress on the network. An example of this technique is the least-congested routing algorithm, which picks the path with more wavelengths available per link [Chatterjee et al. 2015].

2.2. Wavelength Assignment

The wavelength assignment subproblem can be solved through many approaches, such as greedy coloring, first-fit, random-fit, most-used, least-used, etc. [Zang and Jue 2000]. This section addresses only the first two algorithms aforesaid, since both were used on the simulations.

- Greedy coloring: the same wavelength (color) is assigned to as many lightpaths as possible before moving to the next wavelength. This can be done by a sequential graph-coloring algorithm, which is normally preceded by a vertex ordering strategy, such as largest-first or smallest-last.
- First-fit: the wavelengths are sorted by weight in ascending order and then the first available one is picked. First-fit requires no global information and has a low computational overhead since it does not need to search the entire space if a low-weighted wavelength is ready to be used.

3. Genetic Algorithm

Genetic algorithms [Goldberg 1989] are generalized methods of search and optimization inspired by Darwin's Evolution Theory. Within this context, a population of individuals, also called chromosomes, is created as potential solutions to the problem. Each chromosome is submitted to genetic operators, such as selection, crossover and mutation, which either modify the current individuals or generate new ones into the population. The main idea behind GAs is to apply a fitness function to each chromosome in order to evaluate them for finally choosing, after some number of generations, among the fittest individuals — which have higher chances to be the best ones, since they have survived along the process —, the most suitable solution to the problem.

The GA in use for this work was proposed on [Teixeira et al. 2017]. The chromosome encoding and genetic operators such as selection, mutation and crossover are explained below briefly. The evaluation step itself, exclusively, is mostly based on [Cardoso et al. 2010].

3.1. Chromosome

Each chromosome is created randomly based on a greedy algorithm [Teixeira et al. 2017], and represents a valid route in the network, as can be seen in Figure 1, where 3 routes from node 0 to node 8 are illustrated. Thus, the routing subproblem, which is to find alternative routes between two points, is solved. Each gene stores an index that represents a router in the network. The alleles are positive integers ranging from 0 up to the maximum number of routers in the network topology. Three chromosomes C_1, C_2 and C_3 would then belong to the population if the respective routes R_1, R_2 and R_3 are valid paths in the network. They are coded as lists, as shown below:

$C_1 = R_1 =$ `<0 2 8>`

$C_2 = R_2 =$ `<0 5 6 7 8>`

$C_3 = R_3 =$ `<0 1 3 4 9 8>`.

3.2. Evaluation

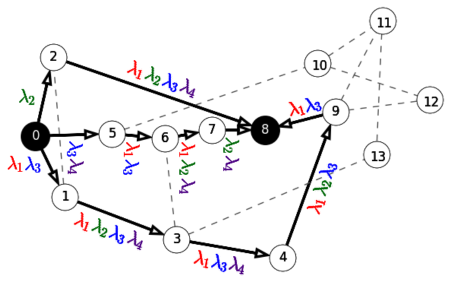
The first step of the evaluation routine is about the wavelength assignment problem under the wavelength-continuity and the distinct wavelength constraints (as discussed on Section 2). Both conditions are met by using a general objective function (GOF) [Cardoso et al. 2010], which was developed considering only static traffic. This function, shown in Equation 1, ensures a communication with minimal restrictions in static-traffic, wavelength-multiplexed networks by not taking power, physical distance and optoelectronic conversion constraints into consideration.

$$L(R_j, \lambda_x) = \frac{\sum_{i=1}^{n} (w_{\lambda_x})_i}{w_{\lambda_x} \sum_{i=1}^{n} l_i} \tag{1}$$

According to [Cardoso et al. 2010], GOF defines a label L for the light-path defined by a route R_j when a wavelength λ_x is used. When $L = 1$, the wavelength is available over all links l of the route, which means that no wavelength converters must be used to establish a connection from a source node to a destination node. Any value

$L < 1$, however, means at least one λ converter must be used on that light-path in order to communicate the SD pair. Notice that lower wavelength indexes have lower weights w_{λ_x} as well, i.e., $w_{\lambda_{x-1}} < w_{\lambda_x}$. The number of iterations to calculate links and wavelength weights in links is defined by n and i identifies the number of links at a route.

A simulation of the computation of the labels for hypothetical, previously shown routes $\texttt{R}_1$, $\texttt{R}_2$ and $\texttt{R}_3$ is shown in Figure 1. The NSFNET topology is assumed to have four channels or wavelengths on each link, represented by λ_1, λ_2, λ_3 and λ_4. A missing wavelength on a specific link means that it is being used in another light-path.



0
1
2
3
4
5
6
7
8
9
10
11
12
13
λ_2
$\lambda_1 \lambda_3$
$\lambda_1 \lambda_2 \lambda_3 \lambda_4$
$\lambda_3 \lambda_4$
$\lambda_1 \lambda_3$
$\lambda_1 \lambda_2 \lambda_4$
$\lambda_2 \lambda_4$
$\lambda_1 \lambda_3$
$\lambda_1 \lambda_2 \lambda_3 \lambda_4$
$\lambda_1 \lambda_3 \lambda_4$
$\lambda_1 \lambda_2 \lambda_3$

Figure 1. Hypothetical routes $\mathbf{R}_1$, $\mathbf{R}_2$ and $\mathbf{R}_3$ over NSFNET.

By analyzing the labels computed by the general objective function for the route $\texttt{R}_1$ = <0 2 8>, we can notice that λ_2 is the only wavelength available on all links of the path, which agrees with the label $L(R_1, \lambda_2) = 1$. On the other hand, it can be seen at the second route $\texttt{R}_2$ = <0 5 6 7 8> that none of the wavelengths is available on all links of the path, which is confirmed by the labels $L(R_2, \lambda_x) < 1$, $\forall x$. All the calculations for routes $\texttt{R}_1$ and $\texttt{R}_2$ are provided as follows.

$$L(R_1, \lambda_1) = \frac{0+1}{1 \times 2} = 0.50 \qquad L(R_2, \lambda_1) = \frac{0+1+1+0}{1 \times 4} = 0.50$$

$$L(R_1, \lambda_2) = \frac{2+2}{2 \times 2} = 1.00 \checkmark \qquad L(R_2, \lambda_2) = \frac{0+0+2+2}{2 \times 4} = 0.50$$

$$L(R_1, \lambda_3) = \frac{0+3}{3 \times 2} = 0.50 \qquad L(R_2, \lambda_3) = \frac{3+3+0+0}{3 \times 4} = 0.50$$

$$L(R_1, \lambda_4) = \frac{0+4}{4 \times 2} = 0.50 \qquad L(R_2, \lambda_4) = \frac{4+0+4+4}{4 \times 4} = 0.75$$

Conversely, the route R_3 = <0 1 3 4 9 8> has two wavelengths available on all links along the path: λ_1 and λ_3, as confirmed by GOF calculations below. In cases when more than one wavelength is available per link, the proposed genetic algorithm chooses the one with lower weight, which is based on the first-fit procedure. In other words, λ_1 would be chosen over λ_3 in this case because $w_{\lambda_1} < w_{\lambda_3}$.

$$L(R_3, \lambda_1) = \frac{1+1+1+1+1}{1 \times 5} = 1.00 \checkmark$$

$$L(R_3, \lambda_2) = \frac{0+2+0+2+0}{2 \times 5} = 0.40$$

$$L(R_3, \lambda_3) = \frac{3+3+3+3+3}{3 \times 5} = 1.00 \checkmark$$

$$L(R_3, \lambda_4) = \frac{0+4+4+0+0}{4 \times 5} = 0.40$$

In addition to the label defined by the general objective function, two more parameters are used for choosing the best chromosome. Those criteria will be detailed at Section 3.3.

3.3. Selection

The selection operator defines which individuals must reproduce. The tournament selection technique was used on the simulation, where k individuals are randomly chosen from the population and the one with higher fitness value is picked as the first parent chromosome. The same process is repeated for choosing the second parent. The crossover rate P_c defines the percentage of individuals that are selected to reproduce.

In the simulations of Section 3.2, both routes $\mathtt{R}_1$ and $\mathtt{R}_3$ have available wavelengths over all links of their respective paths. The genetic algorithm would consider, however, $\mathtt{R}_3$ as being better than $\mathtt{R}_1$, since $\mathtt{R}_3$ has two wavelengths available, while $\mathtt{R}_1$ has just one. In other words, even though $\mathtt{R}_3$ is longer than $\mathtt{R}_1$, the former is still least congested than the latter, which agrees with the adaptive routing concept. In case of a tie in the congestion criteria (let's say $\mathtt{R}_1$ had two wavelengths available on each link, as well as $\mathtt{R}_3$), the selection step would choose the route with lowest wavelength-available weight w_λ. An insertion sort algorithm was configured to sort the chromosomes according to the criteria. Given that the routes/chromosomes have at least one channel free, which is verified by the GOF algorithm: (1) Sort them on descending order by the number of wavelengths available on the entire path. In other words, the least congested path comes first; (2) If there is a tie on the number of λ available, the selection follows the first-fit procedure by choosing the route whose channel has lower weight w_λ; (3) In case of a second tie, now on the weight of the λ available, the route with the least number of hops is chosen, which means the GA gives priority to the shortest path. If a third tie happens here, the GA then chooses any route randomly.

3.4. Crossover

Crossover is the process of exchanging genes between individuals. It is expected that, at each new generation, the offspring adapts to the environment as well as or even better than its parents. A one-point crossover was used, in which a common router (gene) between two parent routes $\mathtt{C}_{\mathtt{parent}_1}$ and $\mathtt{C}_{\mathtt{parent}_2}$ is chosen as crossover point. Two offsprings $\mathtt{C}_{\mathtt{off}_1}$ and $\mathtt{C}_{\mathtt{off}_2}$ are then generated, each one receiving parts from both parents. The following example illustrates the crossover operation between two routes from source 0 to destination 12. Given that $\underline{8}$ is the parents' common router chosen as crossover point:

$\mathtt{C}_{\mathtt{parent}_1}$ = <**0 5 6 7** $\underline{8}$ 9* 12>

$\mathtt{C}_{\mathtt{parent}_2}$ = <0 2 $\underline{8}$ 9* 11 10 12>

$\mathtt{C}_{\mathtt{off}_1}$ = <**0 5 6 7** $\underline{8}$ 9 11 10 12>

$\mathtt{C}_{\mathtt{off}_2}$ = <0 2 $\underline{8}$ **9 12**>.

If there is more than one common router (such as the index 9* in the example above), the crossover point is randomly chosen, with equal probability, among all the common routers. On the other hand, if there is no common router between the parents, the chromosomes do not cross and no offspring is generated. An offspring is also discarded if it has the same router index on both parts received from each of their parents, since it would produce a loop in the path.

3.5. Mutation

The mutation operator is applied in order to keep the diversity of the population and to prevent premature convergence. Based on a mutation probability value P_m, a random gene g is chosen from a chromosome $\mathtt{C_{original}}$ as a single mutation point, and a new fresh route is created from g to the destination node, producing a new chromosome $\mathtt{C_{mutated}}$ as well. In the following example, $g = \underline{7}$ is the router index chosen as mutation point, and a new route from 7 to the destination 12 is created, which is composed by the router indexes r_1, r_2, ..., r_n.

$$\mathtt{C_{original}} = \mathtt{<0\ \ 5\ \ 6\ \ \underline{7}\ \ 8\ \ 9\ \ 12>}$$
$$\mathtt{C_{mutated}} = \mathtt{<0\ \ 5\ \ 6\ \ \underline{7}}\ \ r_1\ \ r_2\ \ \ldots\ \ r_n\ \ \mathtt{12>}$$

If there is no way to build a new fresh route, the mutation is not performed and the chromosome $\mathtt{C_{original}}$ returns to the population.

4. Proposed Parallel Strategy

This section presents a parallelization strategy for the problem of slowness in the sequential version of the GA simulation. This is done by means of parallelizing the whole simulation by offloading the GA execution to the CUDA device, which greatly reduces the processing time. The work proposed here is an evolution of the previous one [Teixeira et al. 2017] and the simulation presented here assumes the same execution flux of that work.

In general, genetic algorithms demand too much processing time [Lim et al. 2017] to return the fittest solution because it requires the computation of the generations one by one when a sequential processing is in use, which makes their utilization practically unfeasible for huge topologies, like mesh topology, for example. In [Cantú-Paz 1998] some techniques were studied in order to make GA applications run in parallel architectures, where three main types of parallel GAs are highlighted: (1) global single-population master-slave GAs, (2) single-population fine-grained, and (3) multiple-population coarse grained GAs. However, this parallel strategy maintains the sequential GA but turns the simulation environment parallel in order to make future GA researches as to routing problems for huge networks feasible.

4.1. Simulation Description

As the NSFNET are not very large, the minimum number of generations of the GA was set to 35 and the maximum to 80, two values that ensure a good convergence and also allow GA's execution to finish quicker. The same reason can be used to explain the population size set to 30: the diversity of the population does not increase for greater values since the number of different paths from source to destination is relatively small. Mutation and crossover rates were set to $P_m = 0.02$ and $P_c = 0.50$, respectively.

The load L_E on the network was increased from 1 up to 64 Erlangs. The amount of traffic was modeled by a Poisson distribution. The source and destination nodes were kept fixed for all connection requests across the simulation. Based on the number of hops, two distant nodes were chosen to compose the source-destination pairs (SD-pair). The inter-arrival times between connections are exponentially distributed, as well as the time a connection occupies the network resources (holding time), which inspite of being rather simplistic, are good from the theoretical point of view. Given a load L_E, in Erlangs, the average blocking probability $\overline{B}_p$ can be computed as the arithmetic mean of the ratio between the number of blocked calls N_{bc} and the number of calls arrived N_{ca}, as shown in (2):

$$\overline{B}_p(L_E) = \frac{1}{N_s} \times \sum_{i=1}^{N_s} \frac{N_{bc}(L_E)_i}{N_{ca}(L_E)_i}. \quad (2)$$

A total of $N_s = 30$ simulations were executed, each one with the number of iterations set to 150. The NSFNET topology, a 14-node network with bidirectional links, previously shown in Figure 1 was adopted in simulation. The SD-pair is 0 and 12, respectively. The number of λ channels on each fiber link was set to 4 and 8 for analysis with both in results section. Since only wavelength-continuity and distinct wavelength constraints were considered, the distance between optical nodes was not taken into consideration.

4.2. Parallelization

In order to turn a sequential simulation into parallel, firstly it is necessary to understand the main idea behind the sequential version shown in the pseudocode: Algorithm 1. Basically, there are three nested main loops that control: i) the network load (in Erlangs), which requires update times and channels availability on each iteration, located at line 2 in the pseudocode by `erlang` variable; ii) the number of iterations per connection request as required for λ, which hides a queue subproblem, as shown at line 3 by the `ConnectionRequest` variable; iii) and the GA generation loop (line 5), where the GOF function is executed at line 6 as well as the NSFNET and the GA population structures (`NSFNET` and `P`) are passed as arguments. Only NSFNET data (network load and holding time) are maintained after each outer loop iteration. Now, it is just a matter of adapting the simulation program to the chosen parallel architecture which, in this case, is CUDA.

Algorithm 1 Sequential implementation

```
1:  $NSFNET \leftarrow$ InitializeNetwork ()
2:  for $erlang \leftarrow 0$ to $63$ do
3:      for $ConnectionRequest \leftarrow 0$ to $149$ do
4:          $P \leftarrow$ GeneratePopulation ()
5:          for $generation \leftarrow 0$ to $34$ do
6:              GeneticAlgorithm ($NSFNET$, $P$)
7:          end for
8:      end for
9:      UpdateNetwork ($NSFNET$)
10: end for
```

CUDA is a GPU architecture designed by NVIDIA Corporation [Corporation 2019] as a model of GPGPU (General Purpose Graphics Processing Unit) to execute a lightweight function [Sanders and Kandrot 2011] (called kernel function) in terms of threads through a grid configuration for one, two or three dimensions defined in a easy way by the programmer via CUDA API [Kirk and mei W. Hwu 2010] when the kernel function is launched at the CPU side. The grid is basically represented by the total number of threads organized in blocks and, depending on the problem dimension, how these threads are identified in their specific blocks in a global execution scope [Cheng et al. 2014]. All threads are launched at the same time theoretically speaking, but in reality, this depends on GPU device used for that purpose, because of the CUDA concept called warp [Corporation 2019]. Warp has a fixed size equals to 32 and that is a maximum number of threads that can be staggered per streaming multiprocessors (SM) [Cheng et al. 2014] in CUDA Fermi architecture, and that architecture was used for the simulation experiments. Table 1 shows the CUDA device for that purpose. So, as seen in the table, the used device was capable of staggering 64 threads at the same time, which is also called the number of active threads.

Table 1. CUDA Experiment Environment

Description	**Value**
Device	1x GeForce GT 630
CUDA Capability	2.1
Global Memory	4026 MBytes
CUDA Cores	96 (48/SM)
Multiprocessors	2

All those information is crucial to define what is the best strategy in order to parallelize the proposed solution, because parallel applications are very dependent on the architecture features. Having said that, the simulation presented in this paper was modeled in just one dimension, as shown in Figure 2. The grid was set to 150 blocks which is the total number of blocks where each block has 64 threads. The pseudocode in Algorithm 2 shows these settings according to the lines 2, 3, and 4, where the implementation calls the CUDA kernel at the CPU side.

The main idea behind these settings is to convert two of the three nested main loops from sequential simulation into blocks and threads which will run in parallel, thus improving the simulation performance. The blocks will manage connection requests per λ and the threads will manage the network load (in Erlangs), keeping the GA's generation loop in the kernel function which will be run for each thread. Blocks and threads are identified by `blockIdx.x` and `threadIdx.x`, respectively, as shown in Figure 2. With these settings, CUDA kernel will prepare 9600 CUDA threads.

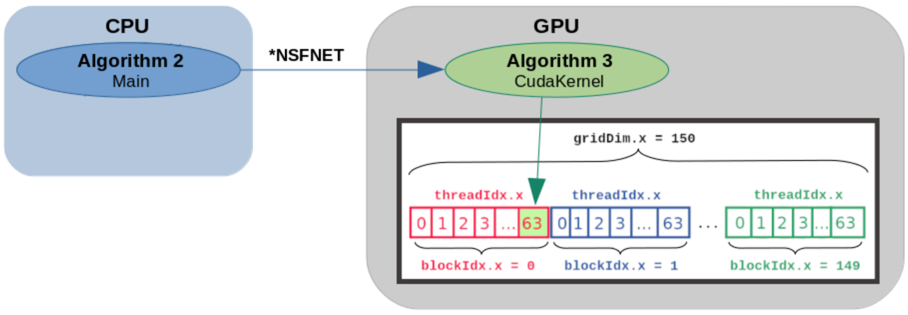


Figure 2. Grid configuration for parallelized simulation in one dimension.

In this parallel execution model (Figure 2), the simulation can handle the whole experiment using the same logic as the sequential version by implementing a queue of connection requests for blocks, that is, block with 0-index (`blockIdx.x = 0`) processing the GOF function in all 64 threads with the load value equals to the thread's index plus one, and the same request index, which is the block index. Thus, all link information that were updated in this block will be preserved to be read at the next block. CUDA architecture doesn't provide thread coordination and doesn't exchange data among threads in the same block [Cheng et al. 2014], but synchronized characteristic around λ availability evaluation is very important to make analysis in terms of block probability likewise that studied in the sequential versions. Because of that, the number of Erlangs was set to 64, keeping the two available SMs occupied with all threads in a block, in terms of warp, thus ensuring the sequential behavior among all CUDA blocks.

For a more complete understanding of the parallel execution, please refer to Figure 2, where the pseudocode Algorithm 2 shows the CPU side execution as well as Algorithm 3 shows the GPU side execution. The following 5 steps describe the parallel execution flux:

1. Initialize NSFNET structures and parameters at CPU side, as shown Algorithm 2, line 1;
2. Initialize network load (in Erlangs) at CPU side so that the first block can be processed at GPU side;
3. Prepare GPU memory and pointers for receiving NSFNET data from CPU side;
4. Call CUDA kernel function (line 4 in Algorithm 2), passing NSFNET data pointers obtained in step 3;
5. CUDA device executes GA for all threads block by block, returning the fittest solution (Algorithm 3);

Algorithm 2 Parallel implementation (Main)

```
1: $NSFNET \leftarrow$ InitializeNetwork ()
2: $nBlocks \leftarrow 64$
3: $nThreads \leftarrow 150$
4: CudaKernel $< nThreads, nBlocks >(NSFNET)$
```

Algorithm 3 Parallel implementation (CudaKernel)

```
1: $P \leftarrow$ GeneratePopulation $(NSFNET)$
2: for $generation = 0$ to $34$ do
3:     GeneticAlgorithm $(NSFNET, P)$
4: end for
5: UpdateNetwork $(NSFNET)$
```

5. Results

The results show that the simulation execution time had a significant improvement due to the adoption of the CUDA parallel architecture (CUDA C) when compared to the Python and C-99 sequential codes. Also, it must to be highlighted that the expected overlapping trend in the curves of Figure 3 (Fitness per implementation) and Figure 4 (Blocking Probability per implementation) was not verified due to the adoption of three different versions of the random number generation libraries as well as three different programming languages. The Python version uses NumPy module, C-99 version uses the `rand()` function from the standard libraries, and CUDA C version uses cuRAND library [Cheng et al. 2014].

5.1. Fitness

As shown in Figure 3, the behavior of the parallel curve (GA_RWA_CU) follows the same trend of the sequential curves (GA_RWA_C and GA_RWA_PY) showing that the parallel algorithm runs correctly at the GPU. Note that Python language has shown to be unstable at the GA convergence when compared to C-99 and CUDA C languages. The average behavior of the three programming languages shows a linear trend. Also, from generation 25 on (out of 35 generations), CUDA C presented the best fitness convergence peak. It is worth noting that after the 24th generation, the number of individuals with available λ reached the average behavior of convergence due to saturation of the possible free network paths.

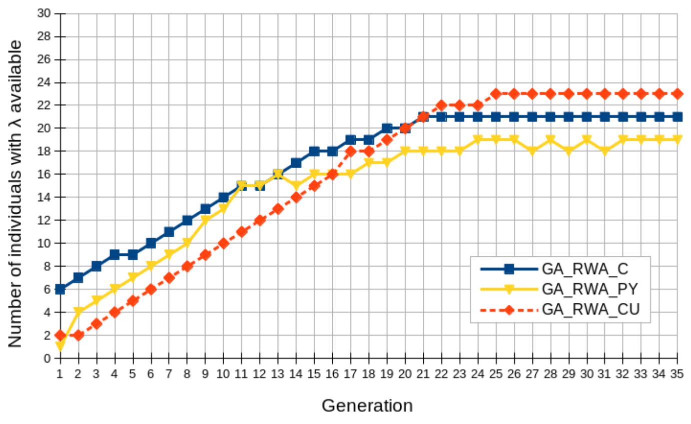


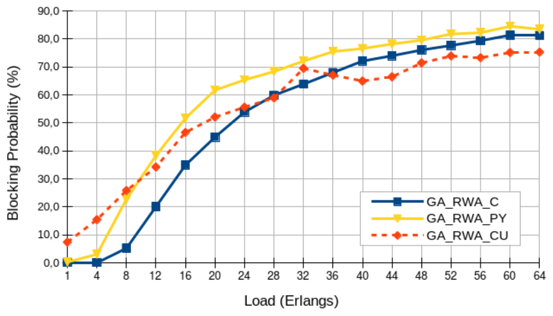
Figure 3. Fitness per implementation.

5.2. Blocking Probability

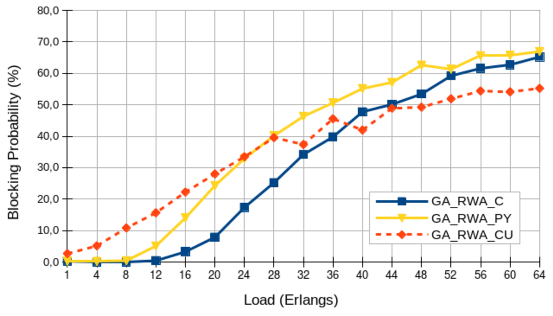
Figures 4a and 4b show that the blocking probability increases as the network load (in Erlangs) also increases, whose curves follow a trend of exponential average behavior, which allows to be stated that the parallel version (GA_RWA_CU) follows the same trend as the sequential versions (GA_RWA_C and GA_RWA_PY), thus validating the effectiveness of the adopted parallelization strategy. In addition, from 40 erlangs of network load on, the CUDA C version presented the least blocking probability for both 4 and 8 channels. It is important to note that the Python version presented the worst blocking probability and the parallel version presented instabilities. The blocking probability results demonstrates that the parallel version fulfilled its main objective, which is to work similarly as the sequential versions, but with better processing time.

5.3. Processing Time

The main result of this work, which is the improvement of the simulation processing time, is shown in Figure 5, which shows the performance of the three languages studied (GA_RWA_C, GA_RWA_PY, and GA_RWA_CU) both for 4 and 8 channels. The results shows that the proposed parallel version of the CUDA simulation obtained the best performance when compared to the sequential versions. The parallelization for the 4-channel



(a) Blocking probability with 64 Erlangs over a 4-channel network.



(b) Blocking probability with 64 Erlangs over a 8-channel network.

Figure 4. Blocking probability per implementation.

network had its execution time 40 times faster than the Python sequential version, as well as, 4 times faster than the C-99 sequential version, approximately. Further, in experiments with the 8-channel network (contrasting with the 4-channel network), the CUDA C processing time reached 35 times faster than the Python version and 4.5 times faster than C-99 version, approximately.

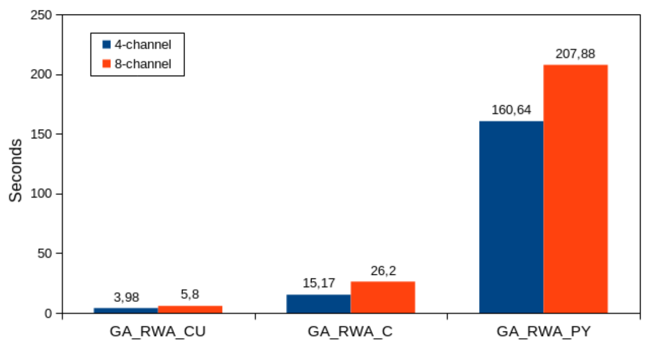


Figure 5. Processing time for 4-channel and 8-channel networks.

6. Conclusion

This paper presented a parallel strategy at GPU level on the CUDA architecture, aiming to reduce the execution time bottleneck found in routing and wavelength assignment (RWA) problems in wavelenght-routed (WDM) networks when genetic algorithm is used. The proposed solution was the creation of a heuristic program to parallelize the problem via hardware, using CUDA architecture and CUDA C programming language in order to improve the execution time of the sequential version of the GA. The proposed heuristic solution presented a performance improvement which was between 35 and 40 times faster than the sequential version in experiments carried out using a NSFNET topology, based on the execution time metric. The results were validated by discrete simulation, using programs developed by the authors in Python and C-99 programming language for the sequential algorithms and CUDA C programming language for the parallel algorithm.

References

Cantú-Paz, E. (1998). A survey of parallel genetic algorithms. *CALCULATEURS PARALLELES*, 10.

Cardoso, A. J. F., Costa, J. C. W. A., and Francês, C. R. L. (2010). A new proposal of an efficient algorithm for routing and wavelength assignment in optical networks. *Journal of Communication and Information Systems*, 25(1):11–18.

Chatterjee, B. C., Sarma, N., and Oki, E. (2015). Routing and spectrum allocation in elastic optical networks: A tutorial. *IEEE Communications Surveys Tutorials*, 17(3):1776–1800.

Cheng, J., Grossman, M., and McKercher, T. (2014). *Professional CUDA C Programming*. John Wiley & Sons, Inc.

Corporation, N. (2019). Cuda c programming guide.

Ellinas, G., Labourdette, J. F., Walker, J. A., Chaudhuri, S., Lin, L., Goldstein, E., and Bala, K. (2004). Network control and management challenges in opaque networks utilizing transparent optical switches. *IEEE Communications Magazine*, 42(2):S16–S24.

Frazer, K., Inc., M. N., and (U.S.), N. S. F. (1996). *NSFNET: A Partnership for High-speed Networking : Final Report, 1987-1995*. Merit Network.

Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition.

Kirk, D. B. and mei W. Hwu, W. (2010). *Progamming Massively Parallel Processors*. Elsevier.

Lim, S. M., Sultan, A. B. M., Sulaiman, M. N., Mustapha, A., and Leong, K. (2017). Crossover and mutation operators of genetic algorithms. *International Journal of Machine Learning and Computing*, 7(1):9–12.

Randhawa, R. and Sohal, J. S. (2010). Static and dynamic routing and wavelength assignment algorithms for future transport networks. *Optik - International Journal for Light and Electron Optics*, 121(8):702 – 710.

Sanders, J. and Kandrot, E. (2011). *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley.

Silva, E. and Gabriel, P. (2019). Genetic algorithms and multiprocessor task scheduling: A systematic literature review. In *Anais do XVI Encontro Nacional de Inteligência Artificial e Computacional*, pages 250–261, Porto Alegre, RS, Brasil. SBC.

Teixeira, D. B. A., Batista, C. T., Cardoso, A. J. F., and Araújo, J. d. S. (2017). A genetic algorithm approach for static routing and wavelength assignment in all-optical wdm networks. In Oliveira, E., Gama, J., Vale, Z., and Lopes Cardoso, H., editors, *Progress in Artificial Intelligence*, pages 421–432, Cham. Springer International Publishing.

Varela, G. N. and Sinclair, M. C. (1999). Ant colony optimisation for virtual-wavelength-path routing and wavelength allocation. In *Proceedings of the 1999 Congress on Evolutionary Computation-CEC99 (Cat. No. 99TH8406)*, volume 3, pages 1809–1816.

Wason, A. and Kaler, R. S. (2007). Wavelength assignment problem in optical wdm networks. In *IJCSNS International Journal of Computer Science and Network Security*.

Zang, H. and Jue, J. P. (2000). A review of routing and wavelength assignment approaches for wavelength-routed optical wdm networks. *Optical Networks Magazine*, 1:47–60.