Bitwise Operations for GPU Implementation of Genetic Algorithms

Martín Pedemonte
Instituto de Computación
Facultad de Ingeniería
Universidad de la República
Montevideo, Uruguay
mpedemon@fing.edu.uy

Enrique Alba
Departmento de Lenguajes y Ciencias de la Computación
Universidad de Málaga
Málaga, Spain
eat@lcc.uma.es

Francisco Luna
Departmento de Lenguajes y Ciencias de la Computación
Universidad de Málaga
Málaga, Spain
flv@lcc.uma.es

ABSTRACT

Research on the implementation of evolutionary algorithms in graphics processing units (GPUs) has grown in recent years since it significantly reduces the execution time of the algorithm. A relevant aspect, which has received little attention in the literature, is the impact of the memory space occupied by the population in the performance of the algorithm, due to limited capacity of several memory spaces in the GPUs. In this paper we analyze the differences in performance of a binary Genetic Algorithm implemented on a GPU using a boolean data type or packing multiple bits into a non boolean data type. Our study considers the influence on the performance of single point and double point crossover for solving the classical One-Max problem. The results obtained show that packing bits for storing binary strings can reduce the execution time up to 50%.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming.

General Terms

Performance

Keywords

CUDA, Evolutionary Computation, GPGPU, GPU, Parallelization, binary-coded Genetic Algorithm.

1. INTRODUCTION

Evolutionary Algorithms are stochastic search methods inspired by the natural process of evolution of species. EAs iteratively evolve a population of individuals representing candidate solutions of the optimization problem. The evolution process is guided by the *survival of the fittest* principle applied to the candidate solutions and it involves the probabilistic application of operators to find better solutions.

Genetic algorithms (GAs) are one of the most popular types of EA. In GAs, individuals are often represented using binary strings, although it is possible to work with non binary encodings [12]. Two alternatives for storing the binary strings in memory when implementing a GA are: using a boolean data type or packing multiple bits in a non boolean data type. The former approach is easier to implement, but wastes memory since the boolean data type is stored in most languages in a byte (the smallest addressable unit of memory). The latter approach does not waste memory but involves working with bitwise operations. Recently, Knuth has drawn attention to the potential offered by working with bitwise operation to speedup general computer programs [6].

Graphics Processing Units (GPUs) were originally designed as specific devices for graphics processing. However, they have become very powerful low-cost platforms for general-purpose computation [11]. For this reason, the study of EAs implementation using GPUs [8] has grown at breathtaking pace as it helps to reduce the runtime of these algorithms by exploiting the massive intrinsic parallelism of such devices.

Threads can access data across multiple memory spaces in current GPUs. Some memory spaces that are among the fastest ones, such as registers and shared block memory, have a very limited size. In this context, storing the same data in less bytes can help to improve the performance. Besides, the savings in the amount of memory used causes that a smaller number of bytes have to be copied when moving the same volume of information. Although there are several works that address the implementation of GA on GPU, they are generally focused on studying how to map one of the existing models of parallelism on a GPU [8]. Few researchers have considered such a low-level implementation issue as the data type used for storing the population [1, 7, 14].

In this work, we make a comparative study of the performance obtained using a boolean data type versus packing multiple bits in a non boolean data type for implementing a binary GA on a GPU. We consider the impact in performance when solving the One-Max problem using the Single Point Crossover (SPX) and the Double Point Crossover (DPX). Our goal is to show that packing bits can significantly reduce the runtime of a binary GA implemented on a GPU, so researchers looking for additional reductions on running times can adopt this type of implementations.

The paper is organized as follow. Next section briefly introduces GPUs and CUDA. Section 3 reviews related work

GECCO'11, July 12–16, 2011, Dublin, Ireland.

on GA implemented on GPUs. Section 4 describes the GA used in this work, commenting aspects related to the bitwise implementation of the GA. Section 5 presents a comprehensive experimental study considering several instance and population sizes. Finally, in Section 6, we outline the conclusions of this work and suggests future research lines.

2. GPUS AND CUDA

GPUs can be viewed as a set of shared memory multicore processors. They are usually considered *many-cores* processors due to the large number of small cores that they contain. GPUs follow the single-program multiple-data parallel programming paradigm in which cores run the same program on multiple parts of the data, but do not have to be executing the same instruction at the same time [3]. The number of threads that currently graphics card can run in parallel is in the order of hundreds and is expected to continue increasing rapidly, what makes these devices a powerful and low cost platform for implementing parallel algorithms.

CUDA [5] is a C language API from nVidia that provides services ranging from common GPU operations in the CUDA library to traditional C memory management semantics in the CUDA runtime, including a device driver dedicated to transfer data between the GPU and CPU. Additionally, nVidia provides a specialized C compiler to build programs developed for the GPU.

A part of an application that runs many times but independently on different data can be isolated in a *kernel* function to be executed on a GPU. When a kernel function is called, a large number of threads are generated on the GPU. The threads generated by a kernel invocation are grouped in blocks that are run concurrently on a single multiprocessor.

Threads can access data on multiple memory spaces during their execution [5]. Registers and shared memory are fast memories. Registers are only accessible by each thread, while shared memory can be accessed by any thread of a block. The local and the global memories are the slowest memories on the graphic card. Constant memory is fast although is read-only for the device. Finally, the texture memory has similar characteristics to constant memory.

3. RELATED WORK

The GPU implementation of most popular EAs has already been studied, being Genetic Programming (GP) the technique with a broader development [8]. Regarding the parallel models implemented, all the standard parallel strategies of EA have already been studied, including the parallel independent runs [15], the master-slave [2], the cellular [13] and the island model [9, 14]. The GPU implementation of GAs [4, 14] and also its hybridization with local searches has also been studied, showing the performance benefits that can be achieved. Although there are many works that address the implementation of GA on GPUs, little effort has been directed to study how the data type used for storing the population influences the performance of the algorithm.

The main effort in this direction was made by Arora et al. [1] that studied a binary-coded (with bit packing) and a real-coded GA on GPU. Arora et al. modified aspects of the algorithm implemented on the GPU to improve the performance, such as generating the initial population and the random numbers on the GPU. Even though the authors report speedups of between 54.2 and 245.4 when solving the One-Max problem with populations of between 512 and 16384 individuals for the binary-coded GA using SPX, the speedup deteriorates when considering instances of size 100.

Recently, Langdon implemented a single instruction multiple data interpreter for GP that exploited sub-machine code level parallelism on GPU [7]. Sub-machine code GP is useful to speedup evaluation of solutions on Boolean problems since it uses bitwise operators to perform Boolean operations in parallel. Langdon reports a peak performance of over 445 billion GP operations per second when solving the 37 Boolean multiplexor.

Finally, Tsutsui and Fujimoto [14] stored the elements of the permutations using an 8 bits data type when tackling the Quadratic Assignment Problem. In this way, the authors managed to load more individuals in shared memory despite limiting the size of the instances that could be considered.

In this work, we focus on analyzing the benefits in performance of a binary GA implemented on a GPU packing multiple bits into a non boolean versus using a boolean data type for storing binary strings. The approach followed aims to make clear for other researchers whether or not to adopt this type of implementations on their works. In this sense we believe this work is a novel contribution.

4. A GA WITH BITWISE OPERATIONS

This section describes the algorithms implemented. The next subsection introduces a high level description of the GA considered in this work and comments on aspects common to the CPU and GPU implementation. Then, the implementation details of the CPU and GPU versions are presented.

4.1 High Level Description of the GA

The GA used in this work is presented in Algorithm 1. The initial population is randomly generated and then the algorithm iterates until the optimal solution is found. In each iteration, a new generation of pop_size individuals is produced by the selection, recombination, and mutation loop, and then replaces the old population, i.e., it is a generational GA. Two parents solutions $p1$ and $p2$ are selected from the population by binary tournament based on their previously computed fitness value. Then, two new solutions $p1'$ and $p2'$ are created by applying a crossover operator (SPX or DPX, in our study) to the parents with a given probability cp. Finally, the bit flip mutation is applied to the recently created solutions with a probability mp (if a solution is mutated, a single randomly selected bit position is flipped).

The most natural representation for candidate solutions in GAs when solving a discrete binary problem is a string of bits. However, there are many alternatives for storing binary strings in memory. In this work, we consider using a boolean data type or packing multiple bits together in a non boolean data type. The first option uses more memory storage but its implementation is straightforward, while the second option saves memory storage but its implementation is more complex and it is not clear a priori if it improves the performance of the algorithm.

The answer to the question of which option is better depends on many aspects such as the architecture of the platform, the operating system, the programming language, and the compiler. In particular, the choice of the programming language does not only influence the performance of the resulting program, but also determines the different data type alternatives. As our goal is analyzing the performance of

Algorithm 1 Genetic algorithm

```
GA(pop_size, cp, mp)
generation = 0;
pop = generateRandomPopulation(pop_size);
evaluate(P(t));
while not optimalSolutionFound do
  for i = 1 to pop_size/2 do
    p1 = selectBinaryTournament(pop);
    p2 = selectBinaryTournament(pop);
    (p1', p2') = crossoverOperator(p1, p2, cp);
    p1'' = bitFlipMutation(p1', mp);
    p2'' = bitFlipMutation(p2', mp);
    newPop[2 * i - 1] = p1'';
    newPop[2 * i] = p2'';
  end for
  pop = newPop;
  evaluate(pop);
  generation = generation + 1;
end while
```

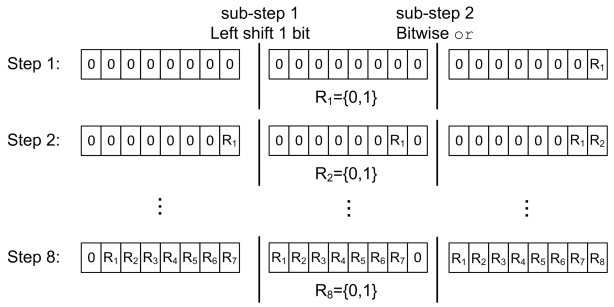

Figure 1: Population initialization in a 8-bit word

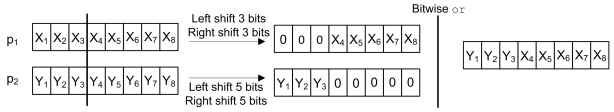

Figure 2: An example of SPX within a 8-bit word

GPU and CPU implementations, we have used CUDA for GPU versions and C for CPU versions. The C programming language provides a boolean data type (`bool`) that uses one byte of space to store one bit of information. The language does not provide any mechanism to automatically handle the packing of bits into data types, and therefore it has to be implemented applying bitwise operations.

Although working at bit level is more complicated, it reduces the amount of memory used. For example, if a population of 480 individuals is used for solving an instance of size 10,000, an implementation using `bool` requires 4,800,000 bytes to store a population, while packing bits in an 8 bits data type needs 600,000 bytes. This means a $\frac{7}{8}$ of reduction.

The savings in the amount of memory used causes that a smaller number of bytes have to be copied back and forth when moving the same volume of information. For example, when a new solution is generated and bits are copied from the parents, a single bit of the individual can be copied at each step using `bool` data type, while 16 bits can be copied together at the same step using `unsigned short`. Moreover, the implementation on a GPU with bit packing can lead to reductions in the execution time of the algorithm because the fastest memory spaces are really small.

Working at bit level impacts on various parts of the GA: the initial generation of the population, the fitness evaluation, and the application of mutation and crossover operators. The main aspects of how bitwise operations impact in the CPU and GPU implementation are commented next.

4.2 CPU Implementation

In the rest of this paper we refer to a *word* to indicate the minimum piece of data of a certain data type. For example, an individual representing a solution for an instance of size 10,000 requires 1,250 words of type `unsigned char` to be stored. If the size of the instance is not a multiple of the size of the data type, there is a valid and also an invalid part in the last word of the individual. For example, an individual representing a solution for an instance of size 10,000 requires 313 words of type `unsigned int` to be stored, but in the final word only 16 of the 32 bits are valid.

To initialize each individual, each word of the individual is initialized with 0s and 1s generated at random. The word (originally empty) is left shifted one bit and a new random bit is added to the word using the bitwise `or` operator. The algorithm iterates until the word is completed. Then, it continues with the next word until the individual is completed. In the final word, if the size of the instance is not a multiple of the size of the data type, additional left shifts have to be computed so valid values are moved into the valid part of the solution. Figure 1 shows how a single word of an individual is initialized when using an 8-bit word.

To evaluate any fitness function, the bits of the solution have to be obtained one by one. In this case, it requires using an auxiliary mask ($2^{dataTypeSize-1}$) to get the bits one by one from each word of an individual (using the bitwise `and` operator). The auxiliary mask is right shifted one bit in every step to get a different bit. In the last word of the solution, if the size of the instance is not a multiple of the size of the data type, only the values in the valid part of the solution have to be considered.

When a bit of the solution has to be mutated, first, we need to calculate in which word of the solution it is stored and which exact position of the word has to be flipped. An auxiliary mask ($2^{dataTypeSize-1}$) is right shifted the required number of positions and the bitwise `xor` operator is applied to the mask and the target word.

To implement SPX, we need to calculate in which word and in which position within the word the crossover point is mapped. If the position within the word is at the beginning of the word, only complete words are copied and no bitwise operations are used. On the contrary, if the position falls within the word, bitwise operations have to be performed in the target word and the rest of the words are copied unchanged. The procedure of SPX within a word is showed in Alg. 2 using the C language notation. The idea of the procedure is to shift the word to one side to discard unnecessary information and then shift it in the opposite direction so the required bits are back in place. Figure 2 shows an example of how SPX works within a word using an 8-bit word.

The DPX implementation is similar to SPX, but there are several special cases that should be considered. In the

Algorithm 2 Single point crossover within a word

$dataLength = 8 * sizeof(Datatype)$;
$realLength = ceil(solutionLength/dataLength)$;
$word = crossoverPoint/dataLength$;
$wordPoint = crossoverPoint\%dataLength$;
$restWP = dataLength - wordPoint$;
$snew1[word] = ((s1[word] >> restWP) << restWP)$
$|((s2[word] << wordPoint) >> wordPoint)$;
$snew2[word] = ((s2[word] >> resttWP) << restWP)$
$|((s1[word] << wordPoint) >> wordPoint)$;

most general case, the two crossover points fall within a different word and the same idea for SPX within a word described is applied on both words. On the other hand, when the two crossover points fall within the same word, the shifts for obtaining the bits required by the operator are made over the same word. Other special cases that deserve attention in order to avoid unnecessary operations are when the two crossover points are in the same position and when one crossover point (or both) falls at the beginning of a word.

4.3 GPU Implementation

Our GPU implementations were designed so that speedup should not degrade when considering an increasing size of the instances. In some approaches, the performance often increases with the size of the population but decreases when the size of the instances increases. The main features of the GPU implementations are described next.

Both the population and the sequences of random numbers (values for the probabilities of crossover and mutation, the crossover point, and the mutate position) are stored in the global memory of the GPU in arrays.

To make a fair comparison between the different CPU and GPU versions with the same crossover operator, the generation and use of the sequence of random numbers is exactly in the same order for all versions. Thus, the results obtained working with the same seed are exactly the same for the different versions. As a consequence, the random number generation and the population initialization procedures of the GPU versions run on the CPU. This restriction can increase the runtime of the GPU versions but isolates the study of the effect in the performance caused by the packing of bits. In this work, we use Mersenne twister [10], one of the most powerful random number generator nowadays.

The GPU implementation has exactly the same behavior as Algorithm 1. Figure 3 shows the structure of the GPU implementation of the GA. The execution starts with the initialization of the population that runs in the CPU and individuals are transferred to the global memory of the GPU. When the algorithm reaches the stop condition, the final population is transferred back from the GPU to the CPU. The random number generation is also executed on the CPU in each iteration and the numbers are also transferred to the global memory of the GPU. On the other hand, the fitness function evaluation, the application of binary tournament, crossover and mutation operators runs entirely on the GPU.

The binary tournament and mutation operator have a straightforward implementation on the GPU. In the binary tournament, each thread gets the fitness of individuals involved in the tournament from the global memory, computes the winner of the tournament, and stores the position of the winner for crossover in an auxiliary structure in global

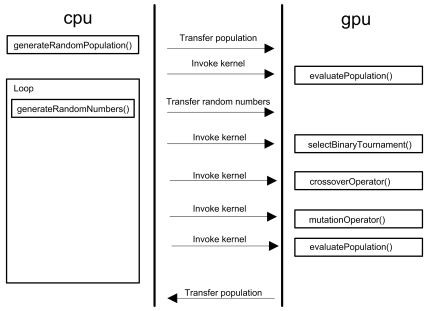

Figure 3: Structure of GPU implementation

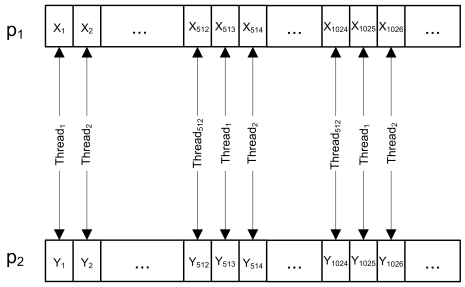

Figure 4: Thread organization for crossover

memory. To execute the binary tournament kernel, as many threads as the size of the population are launched. In the mutation operator, each thread decides whether the individual has to be mutated or not using a random value from the global memory. If the solution has to be mutated, another random value is obtained from the global memory indicating which position has to be modified, and then the bit is flipped. To execute the mutation operator kernel, as many threads as the size of the population are launched.

The fitness evaluation and the application of the crossover operator are organized differently. Each thread processes more than one element of the solution (as long as the size of the solutions is more than 512 words or bits depending on whether the bits are packed or not) but the elements used by a single thread are not contiguous. When the kernels are executed, the maximum number of threads per block is launched (512 in our study).

Algorithm 3 presents the pseudocode of the SPX kernel. Initially, each thread gets the thread block identifier (indicating in which position of the new population the new solutions should be stored), the position of the parents that has to be crossed over, the value for the crossover probability, and the crossover point. When the crossover kernel is executed, as many blocks of threads as half of the population size are launched. Whether the crossover has to be performed or not, the threads copy the first *threadsInBlock* bits, then the second *threadsInBlock* bits and so on until the new solutions are completely generated. Figure 4 shows the thread organization for accessing data in the crossover operator. The access to global memory is coalesced as contiguous threads access to adjacent memory locations. On the other hand, the thread divergence (threads executed sequentially because they are executing different instructions) is mini-

Algorithm 3 SPX kernel pseudocode

```
id = blockId;
p1 = getParentPosition(2 * id);
p2 = getParentPosition(2 * id + 1);
pcValue = xprob[id];
point = xpoint[id];
if pcValue <= PC then
  for i = threadId; i < solLength; i + threadsInBlock
  do
    if i < point then
      copy bit i from parent p1 to child 2*id
      copy bit i from parent p2 to child 2*id+1
    else
      copy bit i from parent p2 to child 2*id
      copy bit i from parent p1 to child 2*id+1
    end if
  end for
else
  for i = threadId; i < solLength; i + threadsInBlock
  do
    copy bit i from parent p1 to child 2*id
    copy bit i from parent p2 to child 2*id+1
  end for
end if
```

mized since threads are usually grouped in 32 for execution and there is a single crossover point, at most a single group of threads diverges (the group that process the positions in which the crossover point falls).

The fitness evaluation function follows the same idea regarding the thread organization and behavior. An auxiliary structure in shared memory stores the partial fitness values computed by each thread. Then, a reduction algorithm is applied to calculate the total value of fitness. When the crossover kernel is executed, as many blocks of threads as the population size are launched.

The incorporation of bitwise operations at each of the operations do not present major difficulties taking as a starting point the GPU implementation using the `bool` data type and the CPU implementation using bit packing. The source code of the GPU kernels is publicly available at `http://www.fing.edu.uy/~mpedemon/bitwise.html`.

5. EXPERIMENTAL RESULTS

This section describes the problem used for the experimental study, the parameters setting and the execution platform. Then, the results obtained are presented and commented.

5.1 The One-Max problem

The One-Max problem is a classical problem in EA literature that has been frequently used in experimental studies since it is quite simple and enables focusing on the features of the algorithm instead of the features of the problem. For this reason, we have selected it for our study.

The problem consists in maximizing the number of 1s in a binary string. Its formulation is presented in Equation 1. The problem has a trivial solution $\overline{x} = (1.....1)$.

$$\max \sum_{i=1}^{N} x_i \qquad x_i = \{0,1\}, \forall i = 1.....N \tag{1}$$

5.2 Parameters setting and test environment

The implemented versions for the experimental study are: Bool (uses the `bool` data type), 8bits (uses the `unsigned char` data type), 16bits (uses the `unsigned short` data type), 32bits (uses the `unsigned int` data type) and 64bits (uses the `unsigned long` data type). Each of these versions was implemented on CPU and GPU for both crossover operators. All versions that use the same crossover operator generate and use the sequence of random numbers in exactly the same order. For this reason, the results obtained working with the same seed are exactly the same.

The GA parameters values used were 0.9 for the crossover probability and 0.1 for the mutation probability (if a solution is mutated, a single randomly selected bit position is flipped). In our experimental study we consider four different instance sizes corresponding to bit sequences: 10000, 20000, 30000, and 40000.

The choice of the population size is an important issue. On one hand, the features of the GPU have to be considered so its potential is not underutilized. On the other hand, working with too large populations can make the comparison unfair for the versions implemented in the CPU. In our study, we consider populations of size 480, 720, and 960 that are not large compared to the values that are often used.

Thirty independent runs were executed for each case of variants and scenarios considered. The execution platform was a PC with a Quad Core Intel Xeon E5530 processor at 2.40 GHz with 48 GB RAM and a Tesla C1060 (240 CUDA cores) using the CentOS Linux 5.4 operating system.

5.3 Experimental analysis

Table 1 and Table 2 show the mean runtime in seconds to find the optimal solution and the Std. Dev. for all versions implemented on CPU using SPX and DPX. The results show that packing multiple bits in a non boolean data type reduces the runtime on CPU implementations. In particular, reductions of up to 20% can be achieved and the 32bits version achieves the larger reduction in runtime in most cases.

Table 3 and Table 4 show the mean runtime in seconds to find the optimal solution and the Std. Dev. for all versions implemented on GPU using SPX and DPX. The results show that also when implementing a GA in a GPU, packing multiple bits in a non boolean data type reduces the runtime. Increasing the number of packed bits up to 32 reduces the runtime of the GA. In particular, reductions of more than 50% can be achieved and the 32bits version achieves systematically the larger reduction in execution time. The 32bits version with SPX or DPX implemented in GPU can solve instances of up to 40,000 variables of the One-Max problem without any specific knowledge of the problem in less than a minute. The 64bit version has a larger execution time than the other packed bit versions but the runtime is shorter than the runtime of Bool version. This result does not seem surprising since streaming multiprocessors in Tesla C1060 graphic card are equipped with 32-bit integer ALUs.

To perform an analysis of the impact on performance in GPU implementations of bit packing, two different indicators are considered. The first indicator is the *Speedup*, presented in Equation 2, that measures the improvement in performance of the GPU implementation versus the CPU implementation of the same version. The second indicator is $Speedup_{II}$, presented in Equation 3, that measures the

Table 1: Runtime in seconds ($mean_{\pm std}$) of CPU implementation with SPX

Pop	Inst	Bool	8bits	16bits	32bits	64bits
480	10000	$220.47_{\pm 11.47}$	$208.36_{\pm 10.76}$	$193.59_{\pm 9.94}$	$\mathbf{188.04}_{\pm 10.33}$	$188.16_{\pm 11.09}$
	20000	$913.83_{\pm 45.76}$	$864.82_{\pm 44.55}$	$800.63_{\pm 40.18}$	$\mathbf{777.15}_{\pm 39.61}$	$777.48_{\pm 43.46}$
	30000	$2121.71_{\pm 82.12}$	$1999.13_{\pm 75.65}$	$1857.48_{\pm 69.56}$	$\mathbf{1798.92}_{\pm 67.07}$	$1830.72_{\pm 69.38}$
	40000	$3820.62_{\pm 196.02}$	$3593.37_{\pm 182.82}$	$3363.48_{\pm 171.45}$	$\mathbf{3244.25}_{\pm 166.03}$	$3294.45_{\pm 168.02}$
720	10000	$273.01_{\pm 9.90}$	$257.87_{\pm 9.09}$	$\mathbf{233.76}_{\pm 9.85}$	$235.96_{\pm 9.88}$	$234.36_{\pm 9.48}$
	20000	$1144.42_{\pm 30.54}$	$1077.75_{\pm 29.01}$	$980.04_{\pm 34.28}$	$\mathbf{964.91}_{\pm 30.03}$	$978.30_{\pm 32.69}$
	30000	$2651.79_{\pm 78.40}$	$2492.08_{\pm 69.92}$	$2271.79_{\pm 75.09}$	$2276.84_{\pm 131.99}$	$\mathbf{2271.71}_{\pm 77.37}$
	40000	$4759.85_{\pm 190.38}$	$4497.17_{\pm 187.72}$	$4205.47_{\pm 163.04}$	$\mathbf{4052.57}_{\pm 158.60}$	$4136.41_{\pm 168.91}$
960	10000	$322.97_{\pm 9.11}$	$304.20_{\pm 7.89}$	$275.01_{\pm 7.23}$	$\mathbf{271.36}_{\pm 8.40}$	$274.58_{\pm 11.54}$
	20000	$1364.14_{\pm 33.57}$	$1297.22_{\pm 47.78}$	$1163.19_{\pm 32.55}$	$\mathbf{1131.38}_{\pm 34.95}$	$1164.95_{\pm 40.41}$
	30000	$3180.68_{\pm 190.95}$	$3048.56_{\pm 144.29}$	$2691.72_{\pm 98.73}$	$\mathbf{2674.68}_{\pm 60.39}$	$2692.71_{\pm 64.82}$
	40000	$5699.59_{\pm 306.77}$	$5318.51_{\pm 168.92}$	$4953.95_{\pm 155.54}$	$\mathbf{4814.22}_{\pm 152.44}$	$4876.88_{\pm 152.97}$

Table 2: Runtime in seconds ($mean_{\pm std}$) of CPU implementation with DPX

Pop	Inst	Bool	8bits	16bits	32bits	64bits
480	10000	$188.24_{\pm 8.30}$	$175.10_{\pm 7.44}$	$171.17_{\pm 10.09}$	$\mathbf{155.98}_{\pm 6.96}$	$161.79_{\pm 7.75}$
	20000	$796.09_{\pm 35.86}$	$740.33_{\pm 28.38}$	$748.11_{\pm 60.22}$	$\mathbf{657.31}_{\pm 29.89}$	$670.87_{\pm 28.03}$
	30000	$1840.05_{\pm 91.63}$	$1705.73_{\pm 87.20}$	$1720.22_{\pm 130.98}$	$\mathbf{1564.12}_{\pm 75.97}$	$1567.36_{\pm 88.42}$
	40000	$3359.88_{\pm 163.06}$	$3176.83_{\pm 195.46}$	$2959.90_{\pm 142.50}$	$\mathbf{2855.10}_{\pm 138.10}$	$2898.93_{\pm 141.45}$
720	10000	$228.07_{\pm 9.32}$	$214.92_{\pm 8.74}$	$197.10_{\pm 10.49}$	$191.57_{\pm 8.81}$	$\mathbf{190.26}_{\pm 8.42}$
	20000	$983.64_{\pm 36.61}$	$926.57_{\pm 34.47}$	$845.46_{\pm 34.22}$	$826.34_{\pm 39.41}$	$\mathbf{821.12}_{\pm 35.77}$
	30000	$2278.43_{\pm 107.33}$	$2094.95_{\pm 90.36}$	$1956.19_{\pm 102.75}$	$1919.88_{\pm 69.28}$	$\mathbf{1880.19}_{\pm 83.51}$
	40000	$4065.13_{\pm 126.30}$	$3853.45_{\pm 133.99}$	$3585.42_{\pm 111.47}$	$\mathbf{3448.41}_{\pm 111.96}$	$3483.69_{\pm 165.18}$
960	10000	$265.92_{\pm 8.47}$	$250.69_{\pm 9.77}$	$229.40_{\pm 9.16}$	$\mathbf{221.81}_{\pm 7.33}$	$226.94_{\pm 7.11}$
	20000	$1128.64_{\pm 38.33}$	$1058.95_{\pm 35.92}$	$971.75_{\pm 34.88}$	$\mathbf{947.86}_{\pm 40.95}$	$954.28_{\pm 39.45}$
	30000	$2609.91_{\pm 62.29}$	$2414.27_{\pm 86.65}$	$2250.25_{\pm 97.92}$	$\mathbf{2209.25}_{\pm 52.27}$	$2226.49_{\pm 70.19}$
	40000	$4693.96_{\pm 180.29}$	$4479.28_{\pm 171.86}$	$4152.77_{\pm 130.58}$	$\mathbf{4032.75}_{\pm 142.14}$	$4086.18_{\pm 129.05}$

Table 3: Runtime in seconds ($mean_{\pm std}$) of GPU implementation with SPX

Pop	Inst	Bool	8bits	16bits	32bits	64bits
480	10000	$9.44_{\pm 0.31}$	$7.94_{\pm 0.22}$	$7.71_{\pm 0.25}$	$\mathbf{7.32}_{\pm 0.18}$	$8.68_{\pm 0.25}$
	20000	$24.12_{\pm 1.02}$	$16.89_{\pm 0.62}$	$15.96_{\pm 0.61}$	$\mathbf{14.58}_{\pm 0.53}$	$18.31_{\pm 0.72}$
	30000	$51.37_{\pm 1.78}$	$30.96_{\pm 1.04}$	$28.43_{\pm 0.90}$	$\mathbf{25.32}_{\pm 0.77}$	$33.82_{\pm 1.16}$
	40000	$84.36_{\pm 4.07}$	$48.77_{\pm 2.26}$	$44.89_{\pm 2.04}$	$\mathbf{39.55}_{\pm 1.78}$	$54.94_{\pm 2.57}$
720	10000	$10.61_{\pm 0.28}$	$8.72_{\pm 0.20}$	$8.40_{\pm 0.19}$	$\mathbf{7.94}_{\pm 0.16}$	$9.24_{\pm 0.19}$
	20000	$29.01_{\pm 0.67}$	$19.62_{\pm 0.42}$	$18.50_{\pm 0.36}$	$\mathbf{16.70}_{\pm 0.35}$	$21.48_{\pm 0.48}$
	30000	$63.04_{\pm 1.64}$	$36.91_{\pm 0.92}$	$33.72_{\pm 0.84}$	$\mathbf{29.67}_{\pm 0.69}$	$40.43_{\pm 1.06}$
	40000	$104.60_{\pm 3.89}$	$59.07_{\pm 2.11}$	$54.04_{\pm 1.90}$	$\mathbf{47.30}_{\pm 1.64}$	$66.74_{\pm 2.40}$
960	10000	$11.80_{\pm 0.22}$	$9.55_{\pm 0.17}$	$9.18_{\pm 0.17}$	$\mathbf{8.61}_{\pm 0.15}$	$10.10_{\pm 0.19}$
	20000	$33.87_{\pm 0.74}$	$22.54_{\pm 0.42}$	$21.10_{\pm 0.41}$	$\mathbf{19.02}_{\pm 0.38}$	$24.58_{\pm 0.49}$
	30000	$74.55_{\pm 1.64}$	$43.08_{\pm 0.89}$	$39.14_{\pm 0.82}$	$\mathbf{34.42}_{\pm 0.67}$	$47.00_{\pm 0.97}$
	40000	$123.63_{\pm 3.70}$	$69.35_{\pm 2.03}$	$63.31_{\pm 1.80}$	$\mathbf{55.20}_{\pm 1.55}$	$78.19_{\pm 2.27}$

Table 4: Runtime in seconds ($mean_{\pm std}$) of GPU implementation with DPX

Pop	Inst	Bool	8bits	16bits	32bits	64bits
480	10000	$8.60_{\pm 0.23}$	$7.38_{\pm 0.15}$	$7.21_{\pm 0.17}$	$\mathbf{6.86}_{\pm 0.13}$	$7.79_{\pm 0.17}$
	20000	$21.47_{\pm 0.68}$	$15.24_{\pm 0.45}$	$14.54_{\pm 0.41}$	$\mathbf{13.24}_{\pm 0.32}$	$16.54_{\pm 0.50}$
	30000	$45.45_{\pm 2.01}$	$27.64_{\pm 1.14}$	$25.59_{\pm 1.02}$	$\mathbf{22.70}_{\pm 0.94}$	$30.19_{\pm 1.26}$
	40000	$75.35_{\pm 3.43}$	$43.98_{\pm 1.95}$	$40.58_{\pm 1.73}$	$\mathbf{35.68}_{\pm 1.56}$	$49.39_{\pm 2.19}$
720	10000	$9.60_{\pm 0.24}$	$8.03_{\pm 0.18}$	$7.85_{\pm 0.16}$	$\mathbf{7.40}_{\pm 0.18}$	$8.50_{\pm 0.18}$
	20000	$25.78_{\pm 0.82}$	$17.70_{\pm 0.48}$	$16.84_{\pm 0.47}$	$\mathbf{15.20}_{\pm 0.43}$	$19.27_{\pm 0.55}$
	30000	$54.90_{\pm 1.75}$	$32.51_{\pm 0.98}$	$29.96_{\pm 0.94}$	$\mathbf{26.38}_{\pm 0.78}$	$35.53_{\pm 1.11}$
	40000	$90.55_{\pm 2.73}$	$51.52_{\pm 1.44}$	$47.35_{\pm 1.32}$	$\mathbf{41.55}_{\pm 1.16}$	$58.07_{\pm 1.66}$
960	10000	$10.61_{\pm 0.21}$	$8.70_{\pm 0.16}$	$8.41_{\pm 0.16}$	$\mathbf{7.99}_{\pm 0.16}$	$9.18_{\pm 0.19}$
	20000	$29.15_{\pm 0.87}$	$19.65_{\pm 0.54}$	$18.56_{\pm 0.48}$	$\mathbf{16.74}_{\pm 0.43}$	$21.29_{\pm 0.58}$
	30000	$63.10_{\pm 1.35}$	$36.84_{\pm 0.75}$	$33.65_{\pm 0.67}$	$\mathbf{29.59}_{\pm 0.57}$	$40.07_{\pm 0.80}$
	40000	$105.38_{\pm 3.17}$	$59.39_{\pm 1.72}$	$54.19_{\pm 1.57}$	$\mathbf{47.57}_{\pm 1.30}$	66.6 ± 1.93

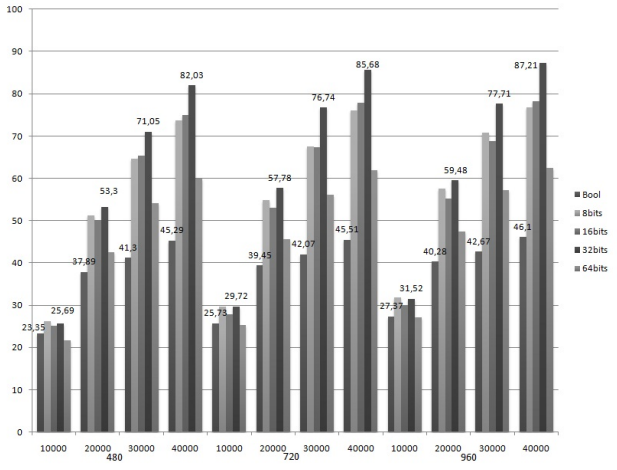

Figure 5: *Speedup* **values of GA with SPX**

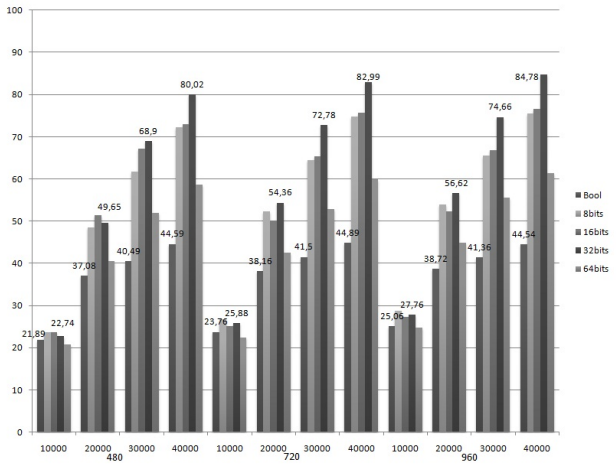

Figure 6: *Speedup* **values of GA with DPX**

improvement in performance of a certain GPU implementation with respect to the CPU implementation of Bool version. Figure 5 and Figure 6 present the *Speedup* of SPX and DPX, respectively. The values of Bool and 32bits versions are indicated.

$$Speedup = \frac{Time_{CPU\ implementation}}{Time_{GPU\ implementation}} \quad (2)$$

$$Speedup_{II} = \frac{Time_{Bool\ on\ CPU}}{Time_{version\ on\ GPU}} \quad (3)$$

The results obtained show that the strategy followed for implementing the GA on the GPU allows speedup values to scale when solving instances of increasing size. For example, the 32bits version with SPX using a population of 480 individuals achieves speedup values of 25.69, 53.30, 71.05, and 82.03 when solving instances of size 10000, 20000, 30000, and 40000, respectively. Even though the speedup values of the implementation without bit packing also scale with the size of the instance, the growth is much higher when the implementation uses bit packing. The Bool version with SPX using a population of 480 individuals only achieves speedup values of 23.35, 37.89, 41.30, and 45.29 when solving instances of size 10000, 20000, 30000, and 40000, respectively. In other words, the 32bits version on a GPU increases at least 79% the speedup with respect to the Bool version for the largest instance considered. The results also scale with the size of the population but with a smaller impact in performance (the 32bits version with SPX on the instance of size 10000 achieves speedup values of 25.69, 29.72, and 31.52 using populations of size 480, 720, and 960, respectively). The speedup values obtained with DPX are slightly worse than those obtained with SPX.

Figure 7 and Figure 8 present the $Speedup_{II}$ of SPX and DPX, respectively. The $Speedup_{II}$ values of Bool version are exactly the same as *Speedup* values. The values of Bool and 32bits versions are indicated. The speedup values of the implementation with bit packing scale with the size of the instance. For example, the 32bits version with SPX using a population of 480 individuals achieves speedup values of 30.12, 62.68, 83.80, and 96.60 when solving instances

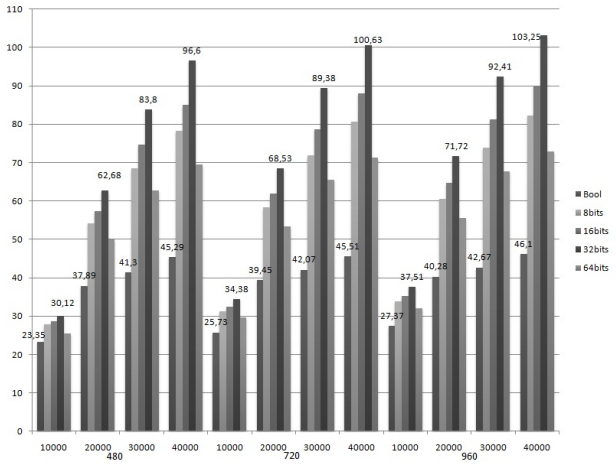

Figure 7: $Speedup_{II}$ **values of GA with SPX**

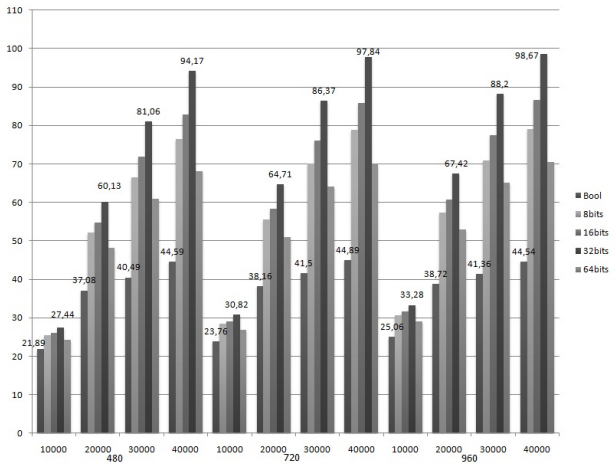

Figure 8: $Speedup_{II}$ **values of GA with DPX**

of size 10000, 20000, 30000, and 40000, respectively. The 32bits version obtains values of up to 100 for the largest instance considered. The speedup values obtained with DPX are slightly worse than those obtained with SPX.

The speedup values reported by Arora et al. [1] are higher than the ones reported in this paper (with a population of 1024 individuals achieves speedup values of 49.75, 125.30, and 105.84 for instances of size 10, 50, and 100, respectively). However, there are a couple reasons that explain the differences. First, the platform used for executions in CPU in this work is a high end CPU (Quad Core Intel Xeon E5530), while the one used by Arora et al. is a low-mid range CPU (AMD Athlon 64 x2 Dual Core 3800+). As the GPU used in both experiments is a Tesla C1060, the comparison of the speedup values is unfair because the runtime of our CPU implementations is shorter. In the second place, Arora et al. generated the initial population and the random numbers on the GPU. As it was mentioned before, in this work both procedures run on the CPU and then the data is transferred to the GPU. If both procedures have been executed on the GPU, the speedup values obtained in this work could be even better.

The results obtained are satisfactory and scale when the size of the population and mainly of the instances increases, showing the potential of the strategy followed for implementing a binary-coded GA on the GPU and the use of bit packing. Although this approach has some difficulties in its implementation, the source code of the GPU kernels of this work are publicly available. This approach makes possible to reduce the runtime up to 50% so it could be useful for solving very large instances in a reasonable time.

6. CONCLUSIONS AND FUTURE WORK

In this paper we have studied the influence in the execution time of using a boolean data type or packing multiple bits in a non boolean data type of a binary-coded GA implemented on a GPU for solving the One-Max problem. We have implemented two classical crossover operators to understand the benefits of this type of implementations.

The results are satisfactory, obtaining the GPU implementation with bit packing in data types of 32 bits speedup values of up to 100. Bit packing is a relatively simple implementation technique that ensures that the resulting GA implementation runs faster. We have tested our proposal with very large instances and showed that reductions on runtime scales when the size of the instances increases. For this reason, and considering that in such scenarios it is extremely important to reduce the execution time, this type of implementations can be very useful.

From this work and the conclusions drawn, three main areas that deserve further work were identified. A first issue that deserves further work is to incorporate constraint handling to the GA because it is a more realistic scenario and can affect the performance of the algorithm. In the second place, the incorporation of specific knowledge to the search process through a local search operator can contribute to improve the performance of the CPU implementation. This mechanism could raise implementation issues that should be carefully evaluated when porting it to the GPU implementation. Finally, solving a more realistic problem than the One-Max used in this paper would help to understand how benefits in performance of the GA generalize to other problems.

7. ACKNOWLEDGMENTS

Enrique Alba and Francisco Luna acknowledge support from the "Consejería de Innovación, Ciencia y Empresa", Junta de Andalucía under contract P07-TIC-03044, the DIRICOM project, and the Spanish Ministry of Science and Innovation and FEDER under contract TIN2008-06491-C04-01, the M* project.

8. REFERENCES

[1] R. Arora, R. Tulshyan, and K. Deb. Parallelization of binary and real-coded genetic algorithms on gpu using cuda. In *Procs. of the IEEE CEC*, pages 1–8, 2010.

[2] T. Clayton, L. Patel, G. Leng, A. Murray, and I. Lindsay. Rapid evaluation and evolution of neural models using graphics card hardware. In *Procs. of GECCO*, pages 299–306, 2008.

[3] F. Darema. The spmd model: Past, present and future. In *Procs. of the 8th Eur. PVM/MPI Users' Group Meeting*, volume 2131 of *LNCS*, page 1, 2001.

[4] K. Fok, T. Wong, and M. Wong. Evolutionary computing on consumer graphics hardware. *Intelligent Systems, IEEE*, 22(2):69–78, 2007.

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

[6] D. Knuth. *The Art of Computer Programming, Volume 4, Fascicle 1: Bitwise Tricks & Techniques; Binary Decision Diagrams*. Addison-Wesley, 2009.

[7] W. Langdon. A many threaded CUDA interpreter for genetic programming. In *Procs. of EuroGP*, volume 6021 of *LNCS*, pages 146–158, 2010.

[8] W. Langdon. Graphics processing units and genetic programming: an overview. *Soft Computing - A Fusion of Foundations, Methodologies and Applications*, pages 1–13, 2011.

[9] T. Lewis and G. Magoulas. Strategies to minimise the total run time of cyclic graph based genetic programming with GPUs. In *Procs. of GECCO*, pages 1379–1386, 2009.

[10] M. Matsumoto and T. Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation*, 8:3–30, 1998.

[11] J. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A. Lefohn, and T. Purcell. A survey of general-purpose computation on graphics hardware. *Comp. Graphics Forum*, 26(1):80–113, 2007.

[12] F. Rothlauf. *Representations for genetic and evolutionary algorithms*. Springer, 2006.

[13] N. Soca, J. Blengio, M. Pedemonte, and P. Ezzatti. PUGACE, a cellular evolutionary algorithm framework on GPUs. In *Procs. of the IEEE CEC*, pages 1–8, 2010.

[14] S. Tsutsui and N. Fujimoto. Solving quadratic assignment problems by genetic algorithms with gpu computation: a case study. In *Procs. of GECCO*, pages 2523–2530, 2009.

[15] S. Tsutsui and N. Fujimoto. An analytical study of gpu computation for solving qaps by parallel evolutionary computation with independent run. In *Procs. of the IEEE CEC*, pages 1–8, 2010.