PUGACE, A Cellular Evolutionary Algorithm framework on GPUs

Nicolás Soca, José Luis Blengio,

Martín Pedemonte y Pablo Ezzatti

Instituto de Computación, Facultad de Ingeniería,

Universidad de la República, Uruguay

2010 IEEE World Congress on Computational Intelligence

Barcelona, Spain

Outline

- Motivation & Objectives
- Graphic processing units
- Cellular Evolutionary Algorithms
- Related work
- PUGACE
- Experimental results
- Conclusions and future work

Motivation & Objectives

- **Parallel Evolutionary Algorithms:**
  - decrease execution time
  - not only speed up the search: new exploration patterns
- **Graphic Processing Units (GPUs):**
  - low cost platform for implementing parallel algorithms
  - complex architecture
- **Objective:**
  - build a tool for easily developing cellular Evolutionary Algorithms (cEAs) on GPUs

Graphic Processing Units

- **Architecture is intrinsically parallel**
- **Shared memory multi-core processors**
- **Memory hierarchy:**
  - **registers**
  - **shared block memory**
  - **local memory**
  - **global memory**
- **Programming tools for general purpose computing: CUDA and OpenCL**

Cellular Evolutionary Algorithms

- Single population structured in many small overlapped neighborhoods
- Each individual belongs to several neighborhoods
- An individual can only be mated for reproduction with individual of its neighborhood
- High-quality solution gradually spreads (diffusion)

Related work

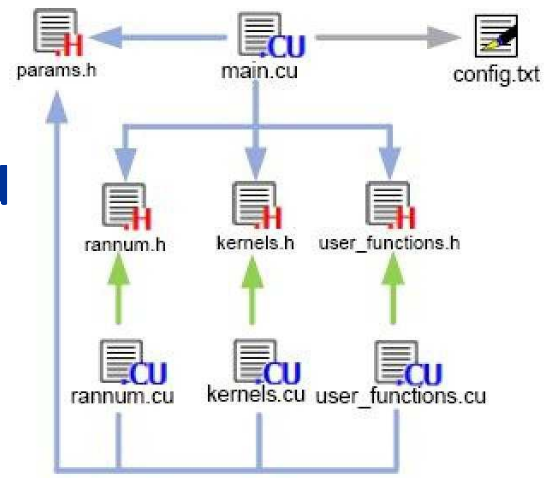
- All standard parallel strategies for Evolutionary Computation have already been implemented successfully on GPUs:
  - master-slave
  - island model
  - cellular model
- cEAs on GPUs obtained good speedup values
- EASEA:
  - generates code that automatically exploits GPU capabilities
  - follows a master-slave model for evaluation of the population
- No proposals of generic framework

PUGACE

- Generic framework for implementing cEAs on GPUs
- Problem related features must be implemented
- In line with: Mallba, JCell, ParadisEO, etc.
- Implemented in C and CUDA (version 2.1).
- Supports different problem encoding, selection policies, crossover operators and mutation operators
- Supports a local search method
- Can be extended to incorporate additional operators

PUGACE (2)

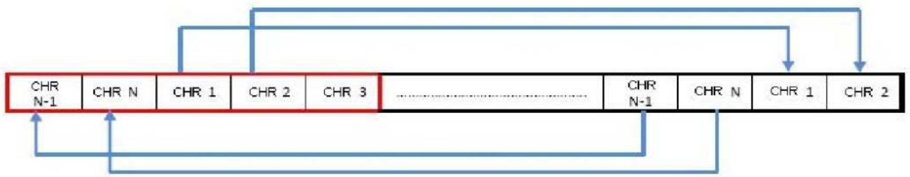
- **Design:**
  - extensible: new evolutionary operators and neighborhood structures can be incorporated
  - easy to use: implementation separated in several modules encapsulating different functionalities (CUDA limitations)



- **First version: generality of the design favored over efficiency**
- **GPU aspects not considered in this version:**
  - maximizing the usage of shared block memory
  - coalescing the access to memory

PUGACE (3)

- **Population:**
  - always resided in the device memory
  - arranged in a circular 1-dimensional structure
  - individuals from both ends are copied to opposite end



- **Each individual executes in a different thread (blocks of varying size)**
- **Neighborhood: configurable number of individuals to the left and right**
- **Application of crossover and mutation operator is decided at block level (to avoid thread divergence)**
- **Problem information preloaded on constant memory**

PUGACE (4)

- Fitness values are stored in an auxiliary vector
- Fitness function evaluation uses an independent thread for each chromosome
- Generational replacement: each parent is replaced by the best one of its children
- Random numbers could be generated:
  - in the CPU and transferred to GPU in each generation (CPU idle times)
  - in the GPU with a specific algorithm based on a linear congruential method

Experimental results

- Quadratic Assignment Problem with a simple approach:
  - permutation representation
  - proportional selection
  - partially mapped crossover
  - mutation operator: randomly swap two values
  - local search: randomly selects a position and makes the best exchange between the selected position and the rest
- Parameters:
  - population = 2048, neighborhood length = 4
  - thread blocks = 32
  - thread per block = 64
- Pentium dual-core 2.5 GHz with 2 GB RAM and a nVidia GeForce 9800 GTX+

Experimental results (2)

- Best known solution in 13 out of 14 instances
- More than 5 Hits in 10 runs for instances with less than 30 facilities
- Less than 5 Hits in 10 runs for instances with more than 30 facilities
- Acceptable for a simple approach

Experimental results (3)

- Tests performed to evaluate reductions in runtime obtained by implementing a cEA on a GPU rather than on a CPU
- Runtime reductions ranged between 15 and 19
- Increase in number of individuals impacts in a sublinear increase in the execution time (10% when doubling the population size)

Conclusions and future work

- **Conclusions:**
  - Proposal of a tool for easily implementing cEA on GPUs
  - High reductions on execution time
- **Future work:**
  - second version:
    - coalescing the access to memory
    - maximizing the usage of shared block memory
    - upgrade to CUDA 3.1
  - use the framework to solve a concrete problem
  - new experiments on different devices

Thank you for your attention