Parallel Metaheuristics on GPU

Advisors: Nouredine MELAB and El-Ghazali TALBI
I. Scientific Context
   1. Parallel Metaheuristics
   2. GPU Computing

II. Contributions
   1. Efficient CPU-GPU Cooperation
   2. Efficient Parallelism Control
   3. Efficient Memory Management
   4. Extension of ParadisEO for GPU-based metaheuristics

III. Conclusion and Future Works
Optimization problems

(Mono-Objective) \[ \min f(x) \quad x \in S \]

(Multi-Objective) \[
\begin{align*}
\min f(x) &= (f_1(x), f_2(x), \ldots, f_n(x)) \\
\text{Const.} \quad x &\in S
\end{align*}
\]

- High-dimensional and complex optimization problems in many areas of industrial concern
  - Telecommunications, Transport, Biology, ...
A taxonomy of optimization methods

- **Exact methods**: optimality but exploitation on small size problem instances
- **Metaheuristics**: near-optimality on larger problem instances, but ...
  - Need of **massively parallel computing** on very large instances
Knapsack problem

- Solution: binary encoding
- Neighborhood example: Hamming distance of one
Population-based metaheuristics

**Evolutionary algorithms**

<table>
<thead>
<tr>
<th>Population of solutions</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4</td>
</tr>
<tr>
<td>0 1 0 0 1</td>
</tr>
<tr>
<td>1 1 1 0 0</td>
</tr>
<tr>
<td>0 0 1 0 0</td>
</tr>
<tr>
<td>0 1 0 1 0</td>
</tr>
<tr>
<td>1 0 1 0 1</td>
</tr>
<tr>
<td>0 1 1 0 1</td>
</tr>
</tbody>
</table>

**Initialization**

<table>
<thead>
<tr>
<th>Individuals</th>
</tr>
</thead>
<tbody>
<tr>
<td>I0 I1 I2 ... In</td>
</tr>
</tbody>
</table>

**Pre-treatment**

**Population evaluation**

**Post-treatment**

**Replacement**

no

End ?

yes

Mutation

<table>
<thead>
<tr>
<th>0 1 2 3 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 1 0</td>
</tr>
</tbody>
</table>

Crossover

<table>
<thead>
<tr>
<th>0 1 2 3 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1 0</td>
</tr>
</tbody>
</table>

**Evolutionary algorithms**

<table>
<thead>
<tr>
<th>0 1 2 3 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 0 0</td>
</tr>
</tbody>
</table>
Parallel models of metaheuristics

Algorithmic-level

M_1 \rightarrow M_2 \rightarrow M_3 \rightarrow M_4 \rightarrow M_5

Iteration-level

sol_1 \rightarrow sol_2 \rightarrow sol_3 \rightarrow sol_n

Solution-level

f_1(sol_n) \rightarrow \ldots \rightarrow f_m(sol_n)
Previous parallel approaches

- Parallelization concepts of metaheuristics
  - S-metaheuristics: simulated annealing [Chandy et al. 1996], tabu search [Crainic et al. 2002], GRASP [Aiex et al. 2003]
  - P-metaheuristics: genetic programming [André et al. 1996], ant colonies [Gambardella et al. 1999], evolutionary algorithms [Alba et al. 2002]
  - Unified view of parallel metaheuristics [Talbi et al. 2009]

- Implementations on parallel and distributed architectures
  - Massively Parallel Processors [Chakrapani et al. 1993]
  - Shared memory or SMP machines [Bevilacqua et al. 2002]
  - Large-scale computational grids [Tantar et al. 2007]
Graphics Processing Units (GPU)

- Used in the past for graphics and video applications ...
- ... but now popular for many other applications such as scientific computing [Owens et al. 2008]
- Popularity due to the publication of the CUDA development toolkit allowing ...
  - ... GPU programming in a C-like language [Garland et al. 2008]
GPU trends

Floating-point operations per second

Memory bandwidth

Theoretical GB/s

Theoretical GB/s

GPU trends

Pentium 4
Core 2
Nehalem
GTX 280
GeForce 8800 GTX
GeForce 8600 GT
Tesla M2050

Tesla M2050
GTX 280
GeForce 8800 GTX
GeForce 8600 GT
Xeon Nehalem

Pentium 4
Core 2
Nehalem
GTX 280
GeForce 8800 GTX
GeForce 8600 GT
Xeon Nehalem
Hardware repartition

- CPU: complex instructions, flow control
- GPU: compute intensive, highly parallel computation
The CPU is considered as a host and the GPU is used as a device coprocessor.

Data must be transferred between the different memory spaces via the PCI bus express ...

<table>
<thead>
<tr>
<th>Configuration</th>
<th>CPU -&gt; GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core 2 Duo T5800</td>
<td>1.76 \times 10^{-2} s</td>
</tr>
<tr>
<td>GeForce 8600M GT</td>
<td></td>
</tr>
<tr>
<td>Core 2 Quad Q6600</td>
<td>1.66 \times 10^{-2} s</td>
</tr>
<tr>
<td>GeForce 8800 GTX</td>
<td></td>
</tr>
<tr>
<td>Xeon E5450</td>
<td>1.25 \times 10^{-2} s</td>
</tr>
<tr>
<td>GeForce GTX 280</td>
<td></td>
</tr>
<tr>
<td>Xeon E5620</td>
<td>0.81 \times 10^{-2} s</td>
</tr>
<tr>
<td>Tesla M2050</td>
<td></td>
</tr>
</tbody>
</table>

One transfer of 30 MB

... many data transfers might become a bottleneck in the performance of GPU applications.
Kernel execution is invoked by CPU over a compute grid:
- Subdivided in a set of thread blocks
- Containing a set of threads with access to shared memory

All threads in the grid run the same program:
- Individual data and individual code flow (Single Program Multiple Data)
Execution model: SIMD model

- GPU architectures are based on hyper-threading
- Single instruction executed on multiple threads (SIMD). Instructions are issued per warp (32 threads).
- A large number of threads are required to cover the memory access latency ...
  - ... an issue is to control the generation of threads
  - Context switching ...
    - ... between warps when stalled (e.g. an operand is not ready)
    - ... enables to minimize stalls with little overhead
- Highly parallel multi-threaded many core
- High memory bandwidth compared to CPU
- Different levels of memory (different latencies)
Objective and challenging issues

- Re-think the parallel models of metaheuristics to take into account the characteristics of GPU
  - The focus on: iteration-level (MW) and algorithmic-level (PC)
  - Three major challenges ...

- Challenge 1: efficient CPU-GPU cooperation
  - Work partitioning between CPU and GPU, data transfer optimization

- Challenge 2: efficient parallelism control
  - Threads generation control (memory constraints)
  - Efficient mapping between work units and threads IDs

- Challenge 3: efficient memory management
  - Which data on which memory (latency and capacity constraints)?
Outline

I. Scientific Context
   1. Parallel Metaheuristics
   2. GPU Computing

II. Contributions
   1. Efficient CPU-GPU Cooperation
   2. Efficient Parallelism Control
   3. Efficient Memory Management
   4. Extension of ParadisEO for GPU-based metaheuristics

III. Conclusion and Future Works
## Taxonomy of major works

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>CPU-GPU cooperation</th>
<th>Parallelism control</th>
<th>Memory management</th>
<th>38 works</th>
</tr>
</thead>
<tbody>
<tr>
<td>Panmictic</td>
<td>Evaluation on GPU</td>
<td>One thread per individual</td>
<td>Global memory</td>
<td>8</td>
</tr>
<tr>
<td>2D toroidal Grid</td>
<td>Evaluation, full parallelization on GPU</td>
<td>One thread per individual</td>
<td>Global, shared and texture memory</td>
<td>10</td>
</tr>
<tr>
<td>Island model</td>
<td>Evaluation, full parallelization on GPU</td>
<td>One block per population</td>
<td>Global, shared and texture memory</td>
<td>4</td>
</tr>
<tr>
<td>Multi-start</td>
<td>Full parallelization on GPU</td>
<td>One thread per algorithm</td>
<td>Global and texture memory</td>
<td>3</td>
</tr>
<tr>
<td>Single solution-based</td>
<td>Generation and evaluation on GPU</td>
<td>One thread per neighbor</td>
<td>Global and texture memory</td>
<td>5</td>
</tr>
<tr>
<td>Hybrid</td>
<td>Generation and evaluation, full parallelization on GPU</td>
<td>One thread per individual / neighbor</td>
<td>Global and texture memory</td>
<td>6</td>
</tr>
<tr>
<td>Multiobjective optimization</td>
<td>Generation and evaluation on GPU</td>
<td>One thread per individual / neighbor</td>
<td>Global and texture memory</td>
<td>2</td>
</tr>
</tbody>
</table>
Optimization problems

- Permutated perceptron problem (PPP)
  - Cryptographic identification scheme
  - Binary encoding
- Quadratic assignment problem (QAP)
  - Facility location or data analysis
  - Permutation
- The Weierstrass continuous function
  - Simulation of fractal surfaces
  - Vector of real values
- Traveling salesman problem (TSP)
  - Planning and logistics
  - Permutation (large instances)
- The Golomb rulers
  - Interferometer for radio astronomy
  - Vector of discrete values
Hardware configurations

- **Configuration 1: laptop**
  Core 2 Duo 2 Ghz + 8600M GT
  (4 multiprocessors - 32 cores)

- **Configuration 2: desktop**
  Core 2 Quad 2.4 Ghz + 8800 GTX
  (16 multiprocessors - 128 cores)

- **Configuration 3: workstation**
  Intel Xeon 3 Ghz + GTX 280
  (30 multiprocessors - 240 cores)

- **Configuration 4: workstation**
  Intel Xeon 3.2 Ghz + Tesla M2050
  (14 multiprocessors - 448 cores)
1. Efficient CPU-GPU Cooperation


Objective and challenging issues

- Re-think the parallel models of metaheuristics to take into account the characteristics of GPU
  - The focus on: iteration-level (MW) and algorithmic-level (PC)
  - Three major challenges...

- **Challenge 1: efficient CPU-GPU cooperation**
  - Work partitioning between CPU and GPU, data transfer optimization

- **Challenge 2: efficient parallelism control**
  - Threads generation control (memory constraints)
  - Efficient mapping between work units and threads IDS

- **Challenge 3: efficient memory management**
  - Which data on which memory (latency and capacity constraints)?
Need of **massively parallel computing** on very **large** solutions set
Work partitioning

- CPU (host) controls the whole sequential part of the metaheuristic
- GPU evaluates the solutions set in parallel
Optimize **CPU→GPU data transfer**

### Issue for S-metaheuristics
- Where the neighborhood is generated?
- Two approaches:
  - **Approach1**: generation on CPU and evaluation on GPU
  - **Approach2**: generation and evaluation on GPU (parallelism control)

### Tabu search - Hamming distance of two
- Where the neighborhood is generated?
- Two approaches: $n(n-1)/2$ neighbors
  - **Approach1**: additional $O(n^3)$ transfers
  - **Approach2**: additional $O(n)$ transfers
Application to the PPP

Evaluation on GPU

Generation and evaluation on GPU

- **8600M GT**
- **8800 GTX**
- **GTX 280**
- **Tesla M2050**

- **CPU LS** process
- **data transfers**
- **GPU kernel**

Evaluation on GPU

Generation and evaluation on GPU

Application to the PPP

Speed-up

Evaluation on GPU

Generation and evaluation on GPU

- **Speed-up**
- **Evaluation on GPU**
- **Generation and evaluation on GPU**

- **8600M GT**
- **8800 GTX**
- **GTX 280**
- **Tesla M2050**

- **CPU LS** process
- **data transfers**
- **GPU kernel**

Application to the PPP
Optimize **GPU→CPU** data transfer

### Issue for S-metaheuristics
- Where the selection of the best neighbor is done?
- Two approaches:
  - **Approach1**: on CPU i.e. transfer of the data structure storing the solution results
  - **Approach2**: on GPU i.e. use of the reduction operation to select the best solution

### Hill climbing - Hamming distance of two neighbors
- Two approaches: $n(n-1)/2$ neighbors
  - **Approach1**: additional $O(n^2)$ transfers
  - **Approach2**: additional $O(1)$ transfer
GPU reduction to select the best solution

- **Binary tree-based reduction mechanism** to find the minimum of each block of threads
- Cooperation of threads of a same block through the shared memory (latency: ~10 cycles)
- Performing iterations on reduction kernels allows to find the minimum of all neighbors
- Complexity: $O(\log_2(n))$, $n$: size of the neighborhood
Application to the Golomb ruler

**No reduction**

- CPU LS
- Process data transfers
- GPU kernel

**Reduction on GPU**

- CPU LS process
- Data transfers
- GPU kernel
Comparison with other parallel architectures

- Emergence of heterogeneous COWs and computational grids as standard platforms for high-performance computing.
- Application to the permuted perceptron problem
- Hybrid OpenMP/MPI implementation

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Configuration 3</th>
<th>Machines</th>
<th>gflops</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU</td>
<td>Intel Xeon E5450, GeForce GTX 280</td>
<td></td>
<td>981.12</td>
</tr>
<tr>
<td>COWs</td>
<td>11 Intel Xeon E5440, 88 CPU cores</td>
<td></td>
<td>995.236</td>
</tr>
<tr>
<td>Grid</td>
<td>2 Intel Xeon E5520, 2 AMD Opteron 2218, 2 Intel Xeon E5520, 4 Intel Xeon E5520, Intel Xeon X5570, Intel Xeon E5520, 96 CPU cores</td>
<td></td>
<td>979.104</td>
</tr>
</tbody>
</table>
Application to the PPP

Tabu search

Analysis of transfers including synchronizations (COWs)

Iterated local search based on a Hill climbing with first improvement (asynchronous)
2. Efficient Parallelism Control

- Thé Van Luong, Nouredine Melab, El-Ghazali Talbi. *Large Neighborhood Local Search Optimization on Graphics Processing Units*. 23rd IEEE International Parallel & Distributed Processing Symposium (IPDPS), Workshop on Large-Scale Parallel Processing (LSPP), Atlanta, US, 2010

Objective and challenging issues

- Re-think the parallel models of metaheuristics to take into account the characteristics of GPU
  - The focus on: iteration-level (MW) and algorithmic-level (PC)
  - Three major challenges ...

- Challenge1: efficient CPU-GPU cooperation
  - Work partitioning between CPU and GPU, data transfer optimization

- Challenge2: efficient parallelism control
  - Threads generation control (memory constraints)
  - Efficient mapping between work units and threads IDs

- Challenge3: efficient memory management
  - Which data on which memory (latency and capacity constraints)?
Thread control (1)

- Traveling salesman problem
  - Large instances failed at execution time due to memory overflow (e.g. hardware register limitation or max number of threads exceeded)
  - Such errors are hard to predict at compilation time since they are specific to a configuration

- Need a thread control for the generation of threads to meet the memory constraints at execution time ...
Thread control (2)

Solution set

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
</tr>
<tr>
<td>11</td>
<td>5</td>
<td>8</td>
<td>19</td>
<td>12</td>
<td>19</td>
<td>13</td>
<td>11</td>
<td>13</td>
<td>14</td>
<td>19</td>
<td>29</td>
<td>14</td>
</tr>
<tr>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td>32</td>
<td>33</td>
<td>34</td>
<td>35</td>
<td>36</td>
<td>13</td>
<td>16</td>
</tr>
</tbody>
</table>

- Increase the threads granularity ...
  - ... with associating each thread to MANY solutions
  - ... to avoid memory overflow (e.g. hardware register limitation)
### Dynamic heuristic for parameters auto-tuning

- To prevent the program from crashing
- To obtain extra performance
Thread control (4)

- Application to the traveling salesman problem (tabu search)

No thread control

<table>
<thead>
<tr>
<th>Application to the traveling salesman problem (tabu search)</th>
<th>No thread control</th>
<th>Thread Control on GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speed-up</td>
<td></td>
<td></td>
</tr>
<tr>
<td>No thread control</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Speed-up</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Application to the traveling salesman problem (tabu search)
According to the threads spatial organization ...

- a unique id must be assigned to each thread to compute on different data

For S-metaheuristics, the challenging issue is to say ...

- which neighbor is assigned to which thread id (required for the generation of the neighborhood on GPU)

Representation-dependent
Mappings are proposed for 4 well-known representations (binary, discrete, permutation, real vector)

- Neighborhood based on a Hamming distance of one
  - The thread with $id=i$ generates and evaluates a candidate solution by flipping the bit number $i$ of the initial solution
  - At most, $n$ threads are generated for a solution of size $n$
Finding a mapping can be challenging

Neighborhood based on a Hamming distance of two

- A thread *id* is associated with two indexes *i* and *j*
- *At most, n x (n-1) / 2* threads are generated for a solution of size *n*
The increase of the neighborhood size may improve the quality of the obtained solutions [Ahuja et al. 2007] ...

... but mostly CPU-time consuming. This mechanism is not often fully exploited in practice.

Large neighborhoods are unusable because of their high computational cost ...

... GPU computing might allow to exploit parallelism in such algorithms.

Application to the permuted perceptron problem (configuration 3: GTX 280)
<table>
<thead>
<tr>
<th>Problem</th>
<th>73 x 73</th>
<th>81 x 81</th>
<th>101 x 101</th>
<th>101 x 117</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fitness</td>
<td>9.8</td>
<td>11.3</td>
<td>20.7</td>
<td>16.8</td>
</tr>
<tr>
<td># iterations</td>
<td>59891</td>
<td>72345</td>
<td>166650</td>
<td>260130</td>
</tr>
<tr>
<td># solutions</td>
<td>11/50</td>
<td>4/50</td>
<td>0/50</td>
<td>0/50</td>
</tr>
<tr>
<td>CPU time</td>
<td>4 s</td>
<td>6 s</td>
<td>16 s</td>
<td>29 s</td>
</tr>
<tr>
<td>GPU time</td>
<td>9 s</td>
<td>13 s</td>
<td>33 s</td>
<td>57 s</td>
</tr>
<tr>
<td>Acceleration</td>
<td>x 0.44</td>
<td>x 0.46</td>
<td>x 0.48</td>
<td>x 0.51</td>
</tr>
</tbody>
</table>

Neighborhood based on a Hamming distance of one

Tabu search
\( n \times (n-1) \times (n-2) / 6 \) iterations

<table>
<thead>
<tr>
<th>Problem</th>
<th>73 x 73</th>
<th>81 x 81</th>
<th>101 x 101</th>
<th>101 x 117</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fitness</td>
<td>15.5</td>
<td>16.2</td>
<td>13.1</td>
<td>12.7</td>
</tr>
<tr>
<td># iterations</td>
<td>42143</td>
<td>65421</td>
<td>133211</td>
<td>260130</td>
</tr>
<tr>
<td># solutions</td>
<td>22/50</td>
<td>17/50</td>
<td>13/50</td>
<td>0/50</td>
</tr>
<tr>
<td>CPU time</td>
<td>81 s</td>
<td>174 s</td>
<td>748 s</td>
<td>1947 s</td>
</tr>
<tr>
<td>GPU time</td>
<td>10 s</td>
<td>16 s</td>
<td>44 s</td>
<td>105 s</td>
</tr>
<tr>
<td>Acceleration</td>
<td>x 8.2</td>
<td>x 11.0</td>
<td>x 17.0</td>
<td>x 18.5</td>
</tr>
</tbody>
</table>

Neighborhood based on a Hamming distance of two

Tabu search
\( n \times (n-1) \times (n-2) / 6 \) iterations

<table>
<thead>
<tr>
<th>Problem</th>
<th>73 x 73</th>
<th>81 x 81</th>
<th>101 x 101</th>
<th>101 x 117</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fitness</td>
<td>2.5</td>
<td>3.2</td>
<td>5.8</td>
<td>7.1</td>
</tr>
<tr>
<td># iterations</td>
<td>19341</td>
<td>40636</td>
<td>100113</td>
<td>214092</td>
</tr>
<tr>
<td># solutions</td>
<td>39/50</td>
<td>33/50</td>
<td>22/50</td>
<td>3/50</td>
</tr>
<tr>
<td>CPU time</td>
<td>1202 s</td>
<td>3730 s</td>
<td>24657 s</td>
<td>88151 s</td>
</tr>
<tr>
<td>GPU time</td>
<td>50 s</td>
<td>146 s</td>
<td>955 s</td>
<td>3551 s</td>
</tr>
<tr>
<td>Acceleration</td>
<td>x 24.2</td>
<td>x 25.5</td>
<td>x 25.8</td>
<td>x 26.3</td>
</tr>
</tbody>
</table>

Neighborhood based on a Hamming distance of three

Tabu search
\( n \times (n-1) \times (n-2) / 6 \) iterations
3. Efficient Memory Management


- Thé Van Luong, Nouredine Melab, El-Ghazali Talbi. *GPU-based Parallel Hybrid Evolutionary Algorithms*. IEEE Congress on Evolutionary Computation (CEC), Barcelona, Spain, 2010
Objective and challenging issues

- Re-think the parallel models of metaheuristics to take into account the characteristics of GPU
  - The focus on: iteration-level (MW) and algorithmic-level (PC)
  - Three major challenges ...

- Challenge 1: efficient CPU-GPU cooperation
  - Work partitioning between CPU and GPU, data transfer optimization

- Challenge 2: efficient parallelism control
  - Threads generation control (memory constraints)
  - Efficient mapping between work units and threads Ids

- Challenge 3: efficient memory management
  - Which data on which memory (latency and capacity constraints) ?
Memory coalescing is not always feasible for structures in optimization problems ...

... use of texture memory as a data cache

- Frequent reuse of data accesses in evaluation functions
- 1D/2D access patterns in optimization problems (e.g. matrices or vectors)
Application to the QAP

- Iterated local search with a tabu search

Global memory only

Texture memory optimization
Algorithmic-level model: PC

- PC model:
  - Emigrants selection policy
  - Replacement/Integration policy
  - Migration decision criterion
  - Exchange topology
Scheme 1: Parallel evaluation of the population

**EA 1 CPU**
- Local population i
  - Initialization
  - Pre-treatment
  - Solutions Evaluation
  - Post-treatment
  - Replacement
  - Migration?
- Copy
- Evaluation function
- Global Memory: Global population, global fitnesses, auxiliary structures

**EA 2 CPU**
- Local population i+1
  - Initialization
  - Pre-treatment
  - Solutions Evaluation
  - Post-treatment
  - Replacement
  - Migration?
- Copy
- Evaluation function
- Global Memory: Global population, global fitnesses, auxiliary structures

- Migration? (no) → End yes
- Migration? (yes) → Copy

**GPU**
- Threads Blocks: T0, T1, T2, ..., Tn, Tn+1, Tn+2, Tn+3, ..., Tn+m
- Evaluation function

(1) → (2) → (1) → (2) → (1)
Scheme 2: Full distribution on GPU

GPU

Global Memory
- Global population, global fitnesses, auxiliary structures

Threads block → local pop i
- T0
- T1
- T2
- ... Tn
- Initialization
- Pre-treatment
- Solutions Evaluation
- Post-treatment
- Replacement
- Migration ?

Threads block → local pop i+1
- T0
- T1
- T2
- ... Tn
- Initialization
- Pre-treatment
- Solutions Evaluation
- Post-treatment
- Replacement
- Migration ?

Pre-treatment

Migration ?

End ?

no

yes

no

End ?

yes
Scheme 3: Full distribution on GPU using shared memory

**GPU**

Global Memory
- Global population, global fitnesses, auxiliary structures

**Threads block -> local pop i**
- Shared Memory
  - Local population, local fitnesses
  - Initialization
  - Pre-treatment
  - Solutions Evaluation
  - Post-treatment
  - Replacement
- Migration ?

**Threads block -> local pop i+1**
- Shared Memory
  - Local population, local fitnesses
  - Initialization
  - Pre-treatment
  - Solutions Evaluation
  - Post-treatment
  - Replacement
- Migration ?

EA 1

No

EA 2

No

End ?

Yes
Issues of distributed schemes

- Sort each local population on GPU (bitonic sort)
- Find the minimum of each local population on GPU (parallel reduction)
- Local threads synchronization for interacting solutions
- Mechanisms of global synchronization of threads if a synchronous migration is needed
- Find efficient topologies between the different local populations according to the threads block organization
Migration on GPU for distributed schemes

Local population i

Shared Memory

Threads Block -> local pop i

Global Memory

Local population i+1

Shared Memory

Threads Block -> local pop i+1

I0 I1 I2 In-1 In

T0 T1 T2 Tn-1 Tn

I0 I1 I2 In-1 In

T0 T1 T2 Tn-1 Tn

2 bests

2 worsts

migration

copy

migration

copy

migration

copy

2 bests

2 worsts
Application to the Weierstrass function (1)

Island model for evolutionary algorithms
(GTX 280 - 64 islands – 128 individuals per island)
Island model for evolutionary algorithms (GTX 280 - 64 islands – 128 individuals per island)

Application to the Weierstrass function (2)
4. Extension of ParadisEO for GPU-based metaheuristics


Software framework

- EO: Design and implementation of population-based metaheuristics
- MO: Design and implementation of solution-based metaheuristics
- MOEO: Design and implementation of multi-objective metaheuristics
- PEO: Design and implementation of parallel models for metaheuristics

Conceptual objectives

- Clear separation between resolution methods and problems at hand, maximum code reuse, flexibility, large panels of methods and portability

Parallel and distributed deployment

Transparent parallelization and distribution

- Clusters and networks of workstations: Communication library MPI
- SMP and Multi-core: Multi-threading Pthreads
- Grid computing:
  - High-performance Grids: Globus, MPICH-G
  - Desktop Grids: Condor (checkpointing & Recovery)
- GPU computing

Iteration-level implementation

- Parallel model which provides generic concepts ...
- ... transparent and parallel evaluation of solutions on GPU
- MW model which does not change the original semantics of algorithms
Layered architecture of ParadisEO-GPU

1. Allocate and copy of data
2. Parallel evaluation
3. Copy of evaluation results
ParadisEO User-defined components

- Solution representation
- Population or Neighborhood
- Problem data inputs
- Solution evaluation

ParadisEO-GPU User modifications

- Keywords
- Explicit call to mapping function
- Keywords
- Explicit calls to allocation wrapper
- Keywords
- Explicit calls to allocation wrapper
- Linearizing multidimensional arrays

ParadisEO-GPU Generic and transparent components provided by the framework

- Memory allocation and deallocation
- Data transfers
- Parallel evaluation of solutions
- Solution results
- Mapping functions
- Memory management

Problem-dependent

Problem-independent
Performance of ParadisEO-GPU (1)

- Application to the quadratic assignment problem
- Tabu search using a neighborhood based on pairwise-exchange operator
- Intel Core i7 3.2 Ghz + GTX 480

![Speed-up Graph]

- ParadisEO-GPU
- Opt GPU
Performance of ParadisEO-GPU (2)

- Application to the permuted perceptron problem
- Tabu search using a neighborhood based on a Hamming distance of two
- Intel Core i7 3.2 Ghz + GTX 480
Outline

I. Scientific Context
   1. Parallel Metaheuristics
   2. GPU Computing

II. Contributions
   1. Efficient CPU-GPU Cooperation
   2. Efficient Parallelism Control
   3. Efficient Memory Management
   4. Extension of ParadisEO for GPU-based metaheuristics

III. Conclusion and Future Works
Conclusion

- GPU-based metaheuristics require to re-design existing parallel models: iteration-level (MW) and algorithmic-level (PC)

- Efficient CPU-GPU cooperation
  - Task repartition, optimization of data transfers for S-metaheuristics (generation of the neighborhood on GPU and reduction)

- Efficient parallelism control
  - Mapping between work units and threads Ids, thread control for the threads generation (parameters tuning, fault-tolerance)

- Efficient memory management
  - Use of texture memory for optimization structures, parallelization schemes for parallel cooperative P-metaheuristics (global and shared memory)
 Perspectives

- **Heterogeneous computing for metaheuristics**
  - Efficient exploitation of all available resources at disposal (CPU cores and many GPU cards)
  - Arrival of GPU resources in COWs and grids ...
  - ... conjunction of GPU computing and distributed computing to fully exploit the hierarchy of parallel model of metaheuristics

- **Multiobjective optimization**
  - Parallel archiving of non-dominated solutions represents a prominent issue in the design of multiobjective metaheuristics ...
  - ... SIMD parallel archiving on GPU with additional synchronizations, non-concurrent writing operations and dynamic allocations on GPU
Publications

- 2 international journals: IEEE Transactions on Computers, parallel processing letters
- 9 international conference proceedings: GECCO, EVOCOP, IPDPS ...
- 1 national conference proceeding.
- 2 conference abstracts
- 8 workshops and talks.
- 1 research report.

THANK YOU FOR YOUR ATTENTION
Additional slides
### Performances of optimization problems (1)

<table>
<thead>
<tr>
<th>Problem</th>
<th>Data inputs</th>
<th>Evaluation</th>
<th>Δ-evaluation</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Time</td>
<td>Time</td>
<td>Space</td>
</tr>
<tr>
<td></td>
<td></td>
<td>complexity</td>
<td>complexity</td>
<td>complexity</td>
</tr>
<tr>
<td>Permuted perceptron</td>
<td>One matrix</td>
<td>O(n²)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Quadratic assignment</td>
<td>Two matrices</td>
<td>O(n²)</td>
<td>O(1) / O(n)</td>
<td>O(n²)</td>
</tr>
<tr>
<td>Weierstrass function</td>
<td>-</td>
<td>O(n²)</td>
<td>O(n²)</td>
<td>-</td>
</tr>
<tr>
<td>Traveling salesman</td>
<td>One matrix</td>
<td>O(n)</td>
<td>O(1)</td>
<td>-</td>
</tr>
<tr>
<td>Golomb rulers</td>
<td>-</td>
<td>O(n³)</td>
<td>O(n²)</td>
<td>O(n²)</td>
</tr>
</tbody>
</table>
Performances of optimization problems (2)

- Memory bound
  - Traveling salesman problem
  - Quadratic assignment problem
  - Permutable perceptron problem
- Compute bound
  - Golomb rulers
  - Weierstrass function
Performances of optimization problems (3)

For the Weierstrass function
Performances of optimization problems (4)

Instruction1 → Instruction2

Instruction3 → Instruction4

Instruction4 → Instruction5

Instruction5 → Instruction6, Instruction7

Clock cycles

Instruction1, Instruction2, Instruction3 → Instruction4, Instruction5

Instruction6, Instruction7

Cache miss

Cache hit
Performances of optimization problems (5)

Analysis of data cache
Island model for evolutionary algorithms
GTX 280 – Instance 10 – 64 islands – 128 individuals

- CPU implementation
  - Valgrind + cachegrind
  - L1 cache misses: 84% (around 10 clock cycles per miss)
  - L2 cache misses: 71% (around 200 clock cycles per miss)

- AGPUShared implementation
  - CUDA profiler
  - Shared memory (around 10 clock cycles per access)
  - 16KB per multiprocessor (30 multiprocessors)
  - Population fit into the shared memory: $128 \times 10 \times 4 \approx 5$ KB per island

Memory bound

Compute bound
Irregular application (1)

- Evaluation function of the quadratic assignment
  - Weakly irregular
- S-metaheuristics based on a pair-wise exchange operator
  - $2n-3$ neighbors can be evaluated in $O(n)$
  - $(n-2) \times (n-3) / 2$ neighbors can be evaluated in $O(1)$
Irregular application (2)

Local search based on best improvement

\[
\begin{align*}
T_0 & \quad T_1 & \quad T_2 & \quad T_3 & \quad T_4 \\
(2,3) & \quad (2,4) & \quad (2,5) & \quad (2,6) & \quad (2,7)
\end{align*}
\]

Local search based on first improvement

\[
\begin{align*}
T_0 & \quad T_1 & \quad T_2 & \quad T_3 & \quad T_4 \\
(1,4) & \quad (0,5) & \quad (3,6) & \quad (2,3) & \quad (2,7)
\end{align*}
\]
Cryptanalysis techniques **PPP**

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-1</td>
<td>1</td>
<td>1</td>
<td>-1</td>
<td>-1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>-1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>-1</td>
<td>1</td>
<td>-1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>-1</td>
<td>1</td>
<td>1</td>
<td>-1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>-1</td>
<td>1</td>
<td>1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>-1</td>
<td>1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>-1</td>
<td>1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Majority vector**

-1 1 1

**Probabilistic vector** (EDAs)

1 0.5 0.66 1 0.33 0 0.5 0.83 1
Memory coalescing (1)

Coalescing accesses to global memory (matrix vector product)

```
sum[id] = 0;
for (int i = 0; i < m; i++) {
    sum[id] += A[i * n + id] * B[id];
}
```

- `sum[0] = A[i * n + 0] * B[0]`

SIMD: 1 memory transaction
Memory coalescing (2)

Uncoalesced accesses to global memory for evaluation functions

```c
sum[id] = 0;
for (int i = 0; i < m; i++) {
    sum[id] += A[i * n + id] * B[p[id]];
}
```

6 memory transactions

Because of LS methods structures, memory coalescing is difficult to realize

⇒ it can lead to a significantly performance decrease.
## Pros and cons of parallel (memory management)

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Parameters</th>
<th>Limitation of the local population size</th>
<th>Limitation of the instance size</th>
<th>Limitation of the total population size</th>
<th>Speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Heterogeneous</td>
<td>Not limited</td>
<td>Very Low</td>
<td>Very Low</td>
<td>Slow</td>
</tr>
<tr>
<td>CPU+GPU</td>
<td>Heterogeneous</td>
<td>Not limited</td>
<td>Low</td>
<td>Low</td>
<td>Fast</td>
</tr>
<tr>
<td>GPU</td>
<td>Homogeneous</td>
<td>Size of a threads block</td>
<td>Low</td>
<td>Medium</td>
<td>Very Fast</td>
</tr>
<tr>
<td>GPU Shared Memory</td>
<td>Homogeneous</td>
<td>Limited to shared memory</td>
<td>Limited to shared memory</td>
<td>Medium</td>
<td>Lightning Fast</td>
</tr>
</tbody>
</table>