Population Parallel GP on the G80 GPU

D. Robilliard, V. Marion-Poty, C. Fonlupt

Univ. Littoral Côte d'Opale, France.

Supported by Interreg IIIA project 182b

Summary

- **Introduction: objectives and experimental framework**
- **An overview of the G80 GPU architecture**
- **Population parallel model and implementation**
- **Benchmarks and Results**
- **Conclusions**

GPU basics

- **Powerful and cheap**
- **Designed for graphics:**
  - **likely to be available on most computers**
  - **SIMD architecture**
- **Suitable for generic computations**
- **Previous works about running GP on GPUs:**
  - [Harding, Banzhaf] EuroGP 2007
    - *Speedup not measured on full evolutionary runs*
  - [Chitty] GECCO 2007
    - *Uses a graphic API*

Objectives

- **Previous works showed GPU speedups:**
  - for large training sets: up to 65,000 cases
  - for large GP trees: up to 10,000 nodes
- **What about small training sets ?**
  - supervised training data are often costly/ difficult to collect (e.g. medical data)

  => benchmarks using between 64 and 2048 cases
- **What about "typical" GP trees ?**
  - Evaluate speedups for GP trees occurring in standard evolutionary runs
  - On 3 sets of benchmarks parameters, we observed tree sizes ranging from 30 to 208 nodes

Experimental Framework

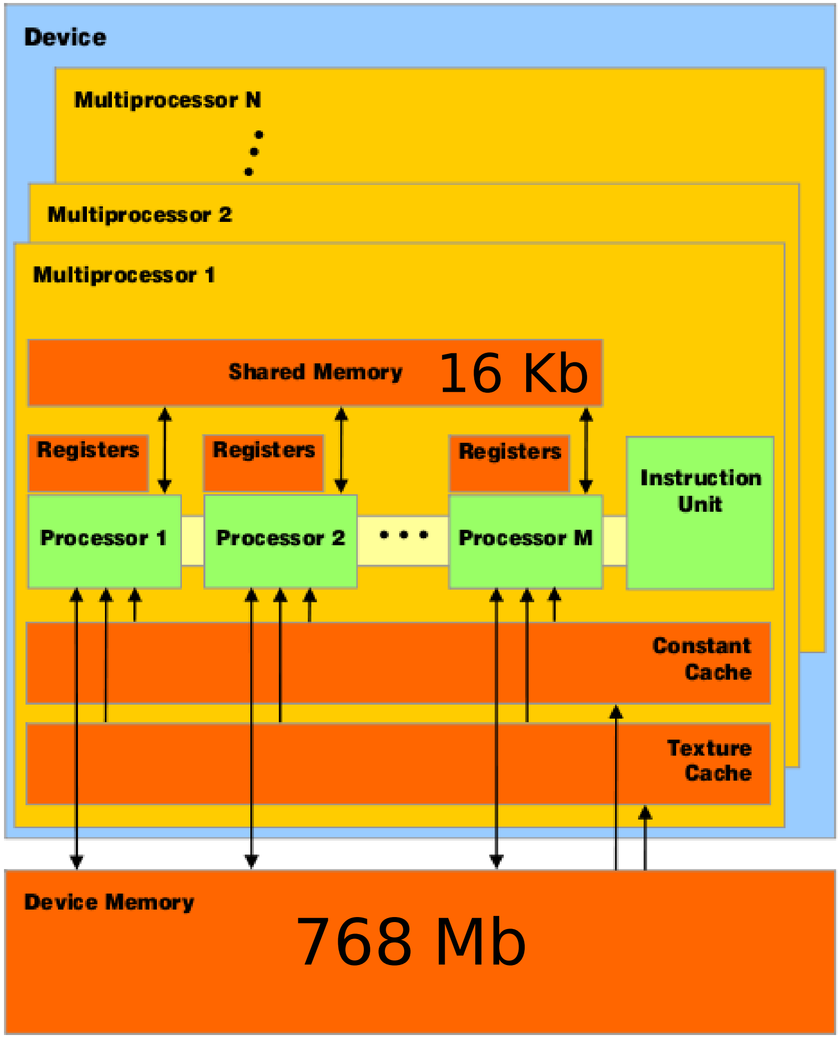
- **Interfacing GPU with the ECJ library**
  - Is it possible to keep the flexibility of ECJ?

  => Only the evaluation phase will be ported on GPU
  - Is it worth it ?

  => Speedup measured for full evolutionary runs
- **Hardware: nVidia GTX 8800 (G80 GPU)**
- **GPU language: CUDA**
  - Free software (although proprietary)
  - Only available for the nVidia G80 family of GPUs
  - Close to C language
  - Several general purpose libraries available (linear algebra, FFT, ...)
  - Fine grain access to the G80

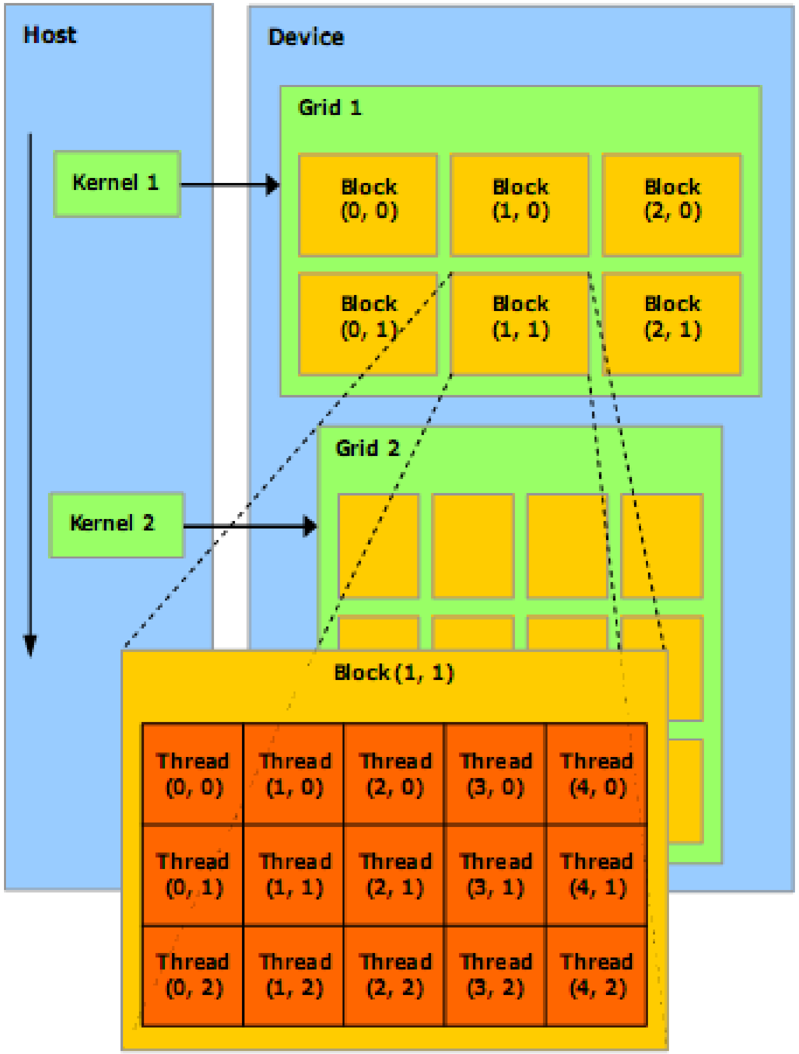
G80 Architecture / GeForce 8800GTX

- **16 multiprocessors x 8 internal stream processors**
  - = 128 stream processors at 1.35 Ghz
  - Other circuits at 675 Mhz
- **multiprocessor :**
  - 16 ko of fast memory shared between stream processors
  - 8 ko of texture and constant caches
  - independent instruction register
- **stream processor :**
  - SIMD mode
  - local memory for registers



Execution Model

- **GRID : set of computations**
- **a GRID is divided into BLOCKS :**
  - independent subset of computations, to be run on one multiprocessor
  - no fixed order of execution between blocks:
    - parallel execution if enough multiprocessors
    - or else time sharing
- **a BLOCK is divided into THREADS :**
  - instances of the program (a.k.a. kernel), to be run on the stream processors
  - on the G80 the number of threads on a multiprocessor is a multiple of 32 ("warp size")
  - no fixed order of execution between threads (time sharing)



Université du Littoral Côte d'Opale

GP Parallel Model ?

A) **Parallelizing training cases**

- See e.g. [Harding, Banzhaf 07]
- Same GP program is run on all stream processors => it can be compiled
- Training cases are divided between all stream processors: few training cases => underexploited stream processors

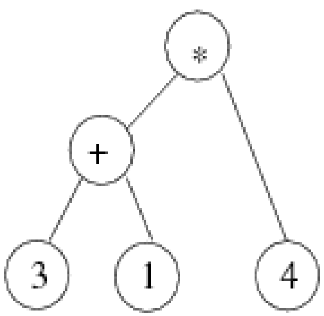
B) <u>**Parallelizing GP programs**</u>

- Increase the ALUs load...
- ... But we need to execute different programs (i.e. GP solutions) on a SIMD machine !
- Solution: use an interpreter (see [Juillé, Pollack 97], GP on SIMD "MASPAR")

The interpreter

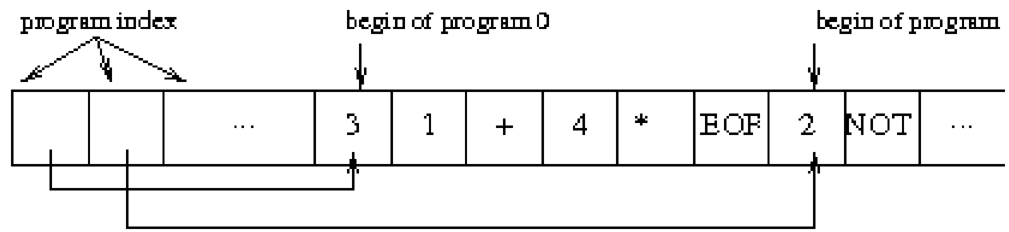
- a loop fetches every instruction
- a switch processes specific instructions
- we used postfixed code with a stack (simple, no recursion)

GP Tree



Postfixed translation

PUSH 3
PUSH 1
ADD
PUSH 4
MUL




```
sp = 0 ;
for each instruction {
  switch (instruction) {
    case OPERAND: push operand;
    case ADD: stack[sp-2]=stack[sp-1]
                                +stack[sp-2];
                   sp--;
    ... } }
```

- **see also [Sanders,1994] for optimization of interpreters :**

  e.g. desynchronize programs/fitness cases even on same multiprocessor

The SIMD trap: divergence

- Divergence occurs when two (or more) parallel threads need to perform different instructions
  - both threads executes the interpreter "switch" statement on their respective GP programs, which are different.
  - => they are required to execute two different branches of the switch
- Divergent parts of code are executed sequentially => efficiency loss.
  - Note: even if both threads interpret the same program, they can diverge if the function set includes an "if" statement...

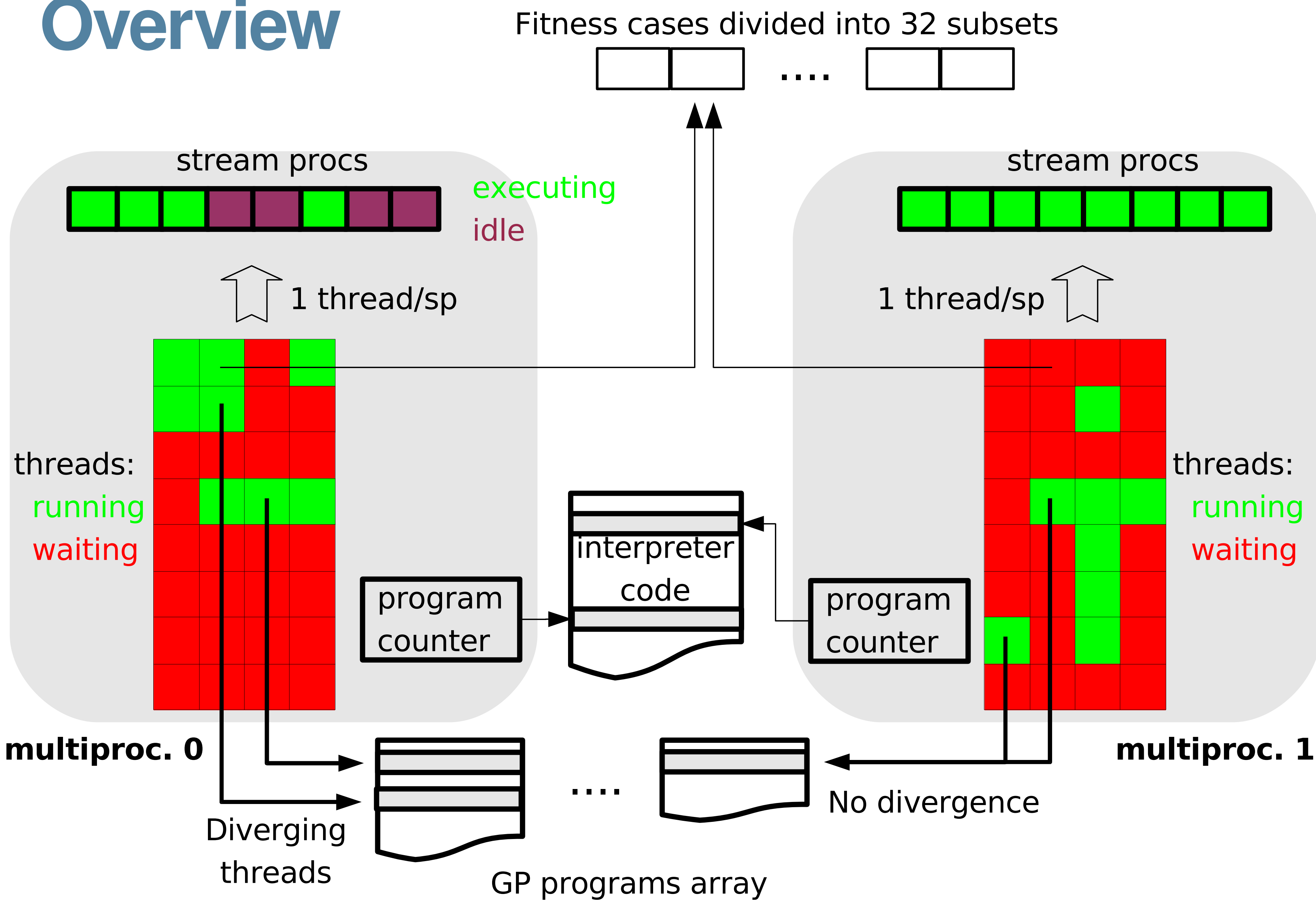
Parallelizing programs on the G80

- The G80 is SPMD rather than SIMD:
  - only one program: the interpreter
  - one program-counter per multiprocessor => no divergence at the multiprocessor level
  - stream processors on any given multiprocessor work in true SIMD mode.

Implementation tip:

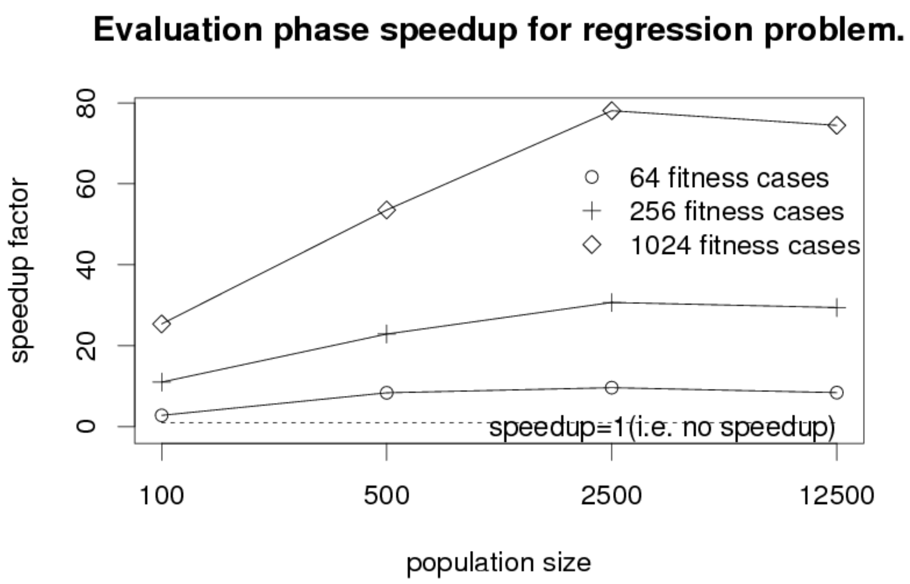
- Dispatch GP programs on different multiprocessors
- Share the fitness cases evaluation on the stream processors (possible divergence depending on function set)

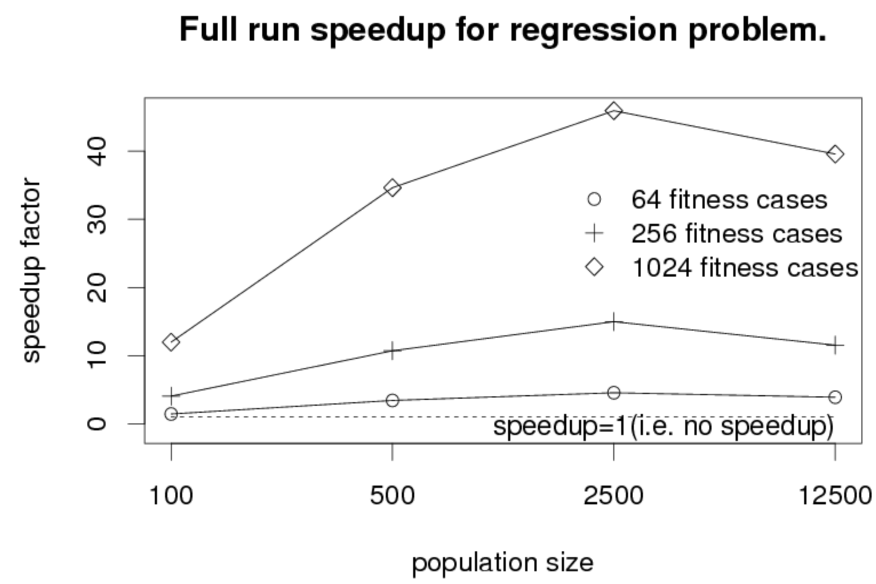
Overview


Fitness cases divided into 32 subsets
....
stream procs
executing
idle
1 thread/sp
stream procs
1 thread/sp
threads:
running
waiting
threads:
running
waiting
interpreter code
program counter
program counter
multiproc. 0
multiproc. 1
....
No divergence
Diverging threads
GP programs array

Regression problem : $x^6 - 2x^4 + x^2$

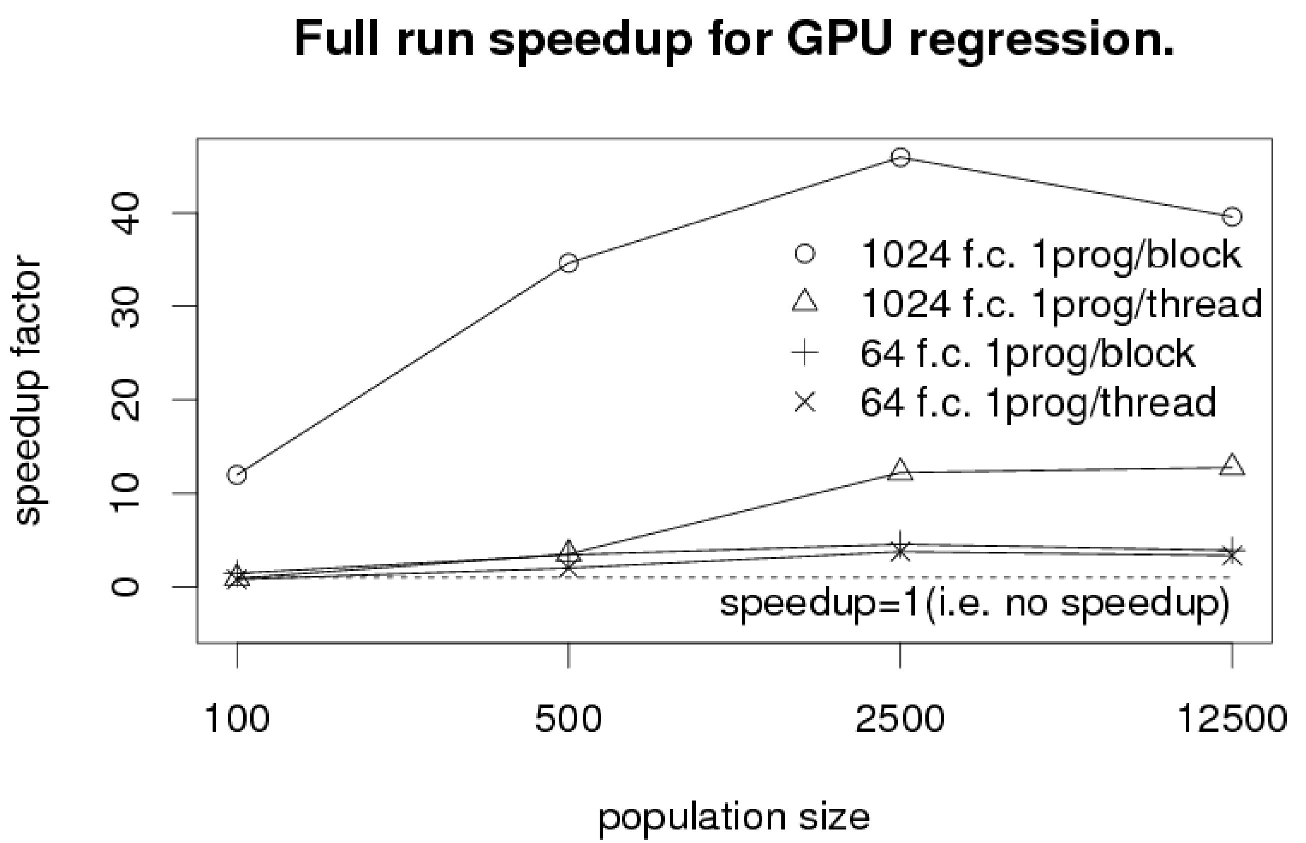
- Function set : {+,*,-,/,sin,cos,exp,log}+{constants,X} => no divergence
- Average tree sizes : 30 to 66
- 50 generations, averaged on 30 independent runs





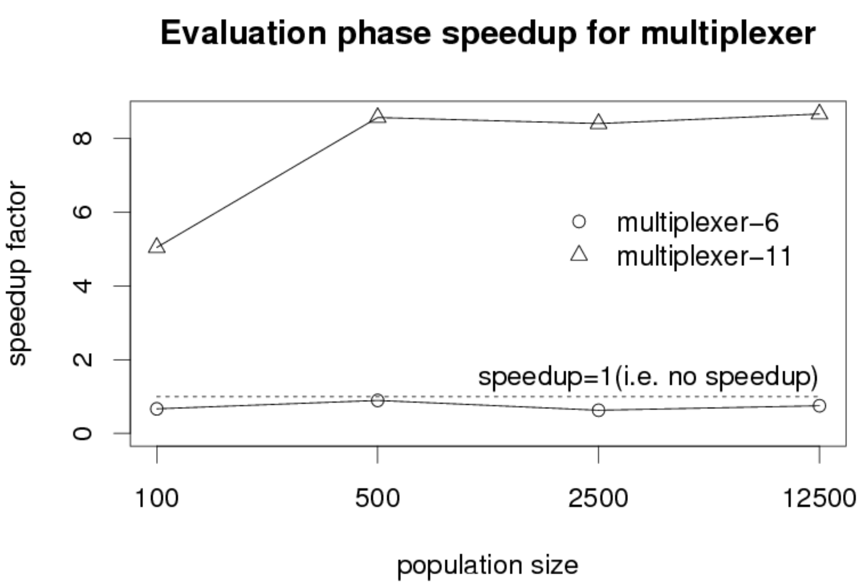
Alternative parallelization scheme

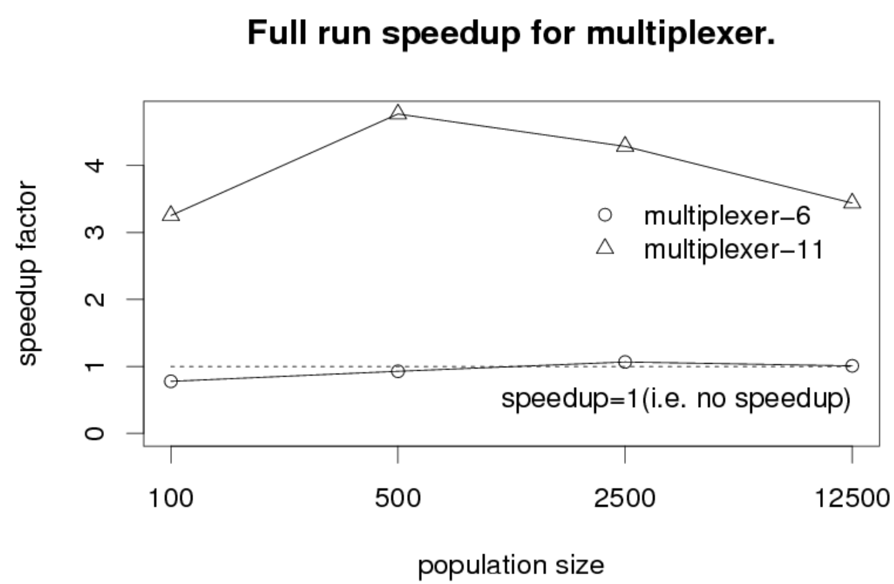
- **1 prog / block vs 1 prog / thread : Regression (64 & 1024 fitness cases)**



Multiplexer 6 bits & 11 bits

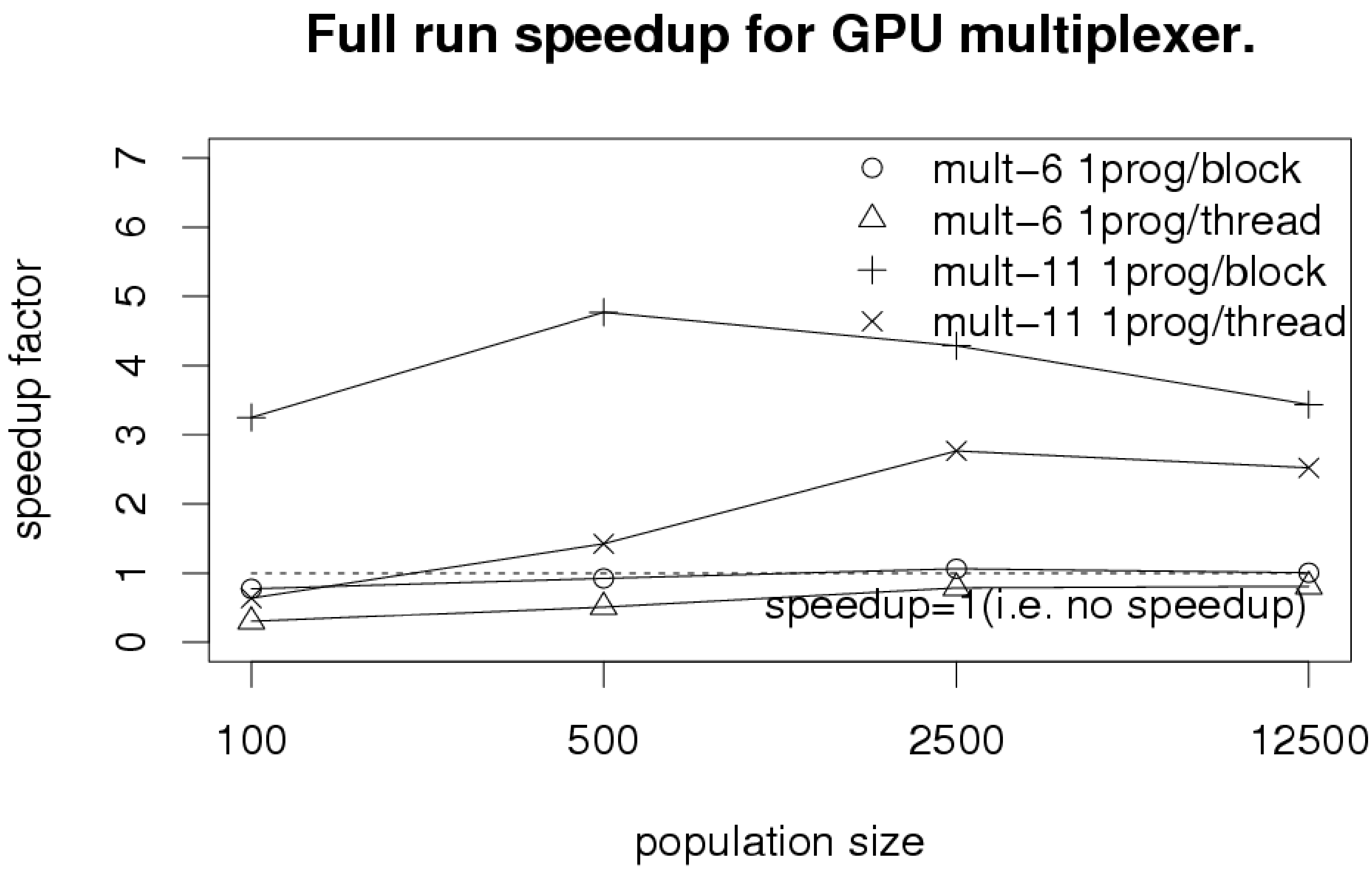
- **Function set :**
  - functions = {And, Or, Not, If} => divergence
  - terminals = {A0-A1, D0-D3} resp. {A0-A2, D0-D7}
- **Average tree sizes :** 112 à 157
- **# Fitness Cases :** 64 (Mult-6) ; 2048 (Mult-11)





Alternative parallelization scheme

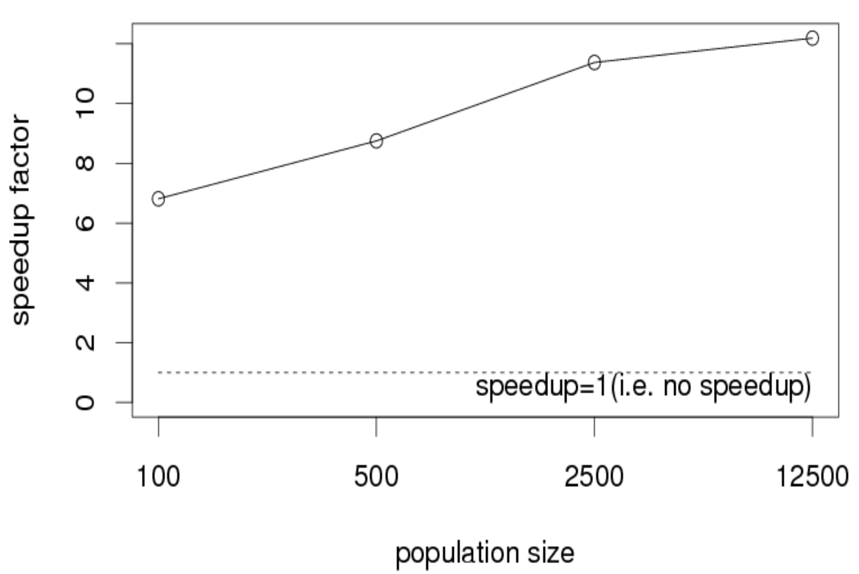
- **1 prog / block vs 1 prog / thread : Multiplexer 6 & 11**



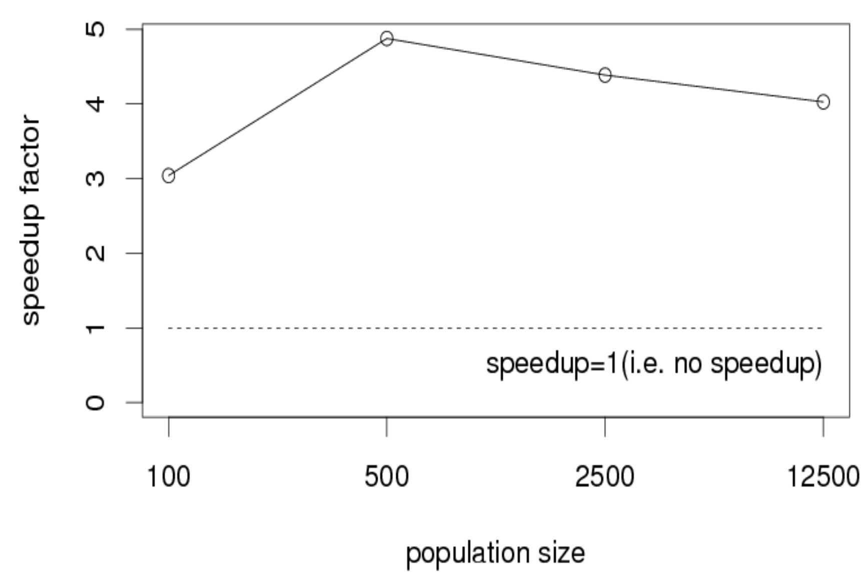
Intertwined Spirals

- **Function set :**
  - functions = {+,-,*,/,cos,sin,If-Ite} => divergence
  - terminals = {real constants, X1, X2}
- **Average tree sizes :** 119 à 208
- **# Fitness Cases :** 194

Evaluation phase speedup for intertwined spirals.

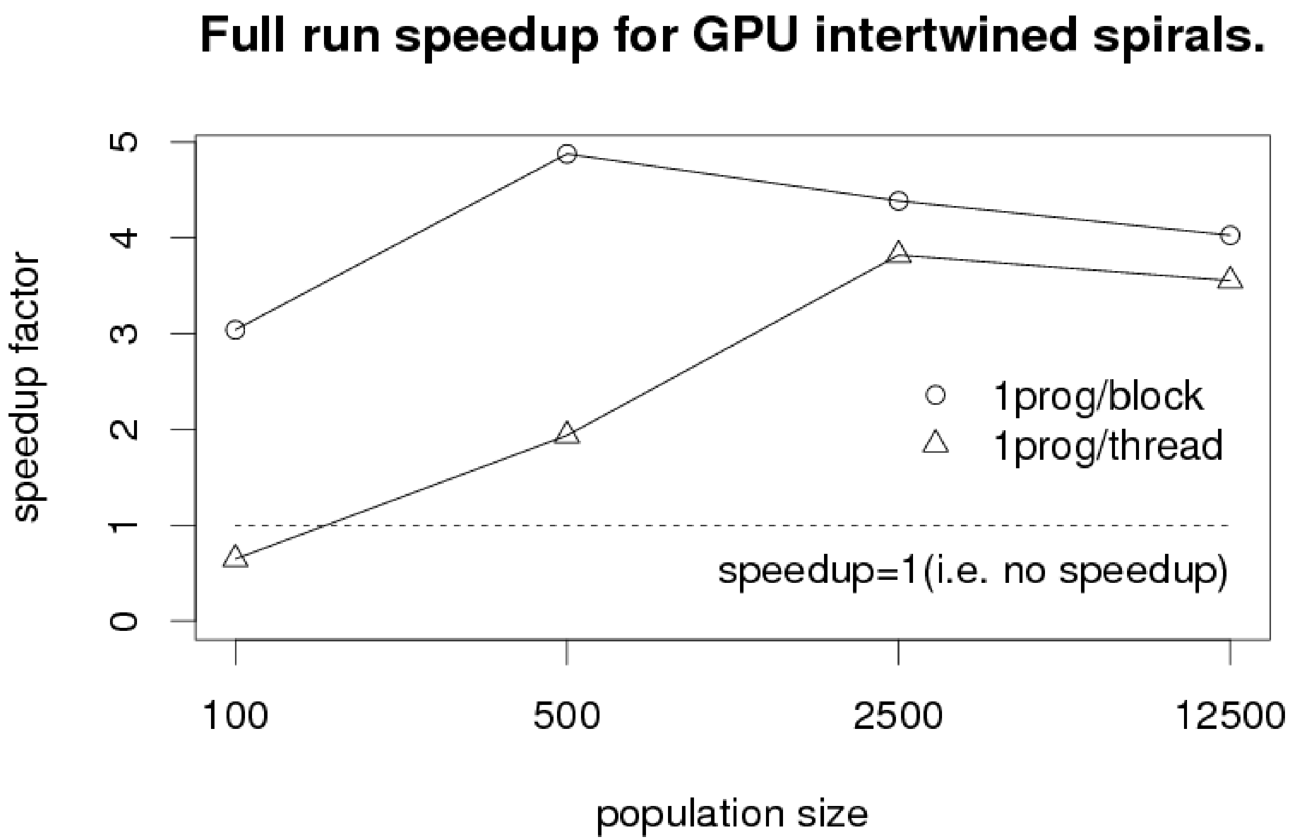


Full run speedup for intertwined spirals.



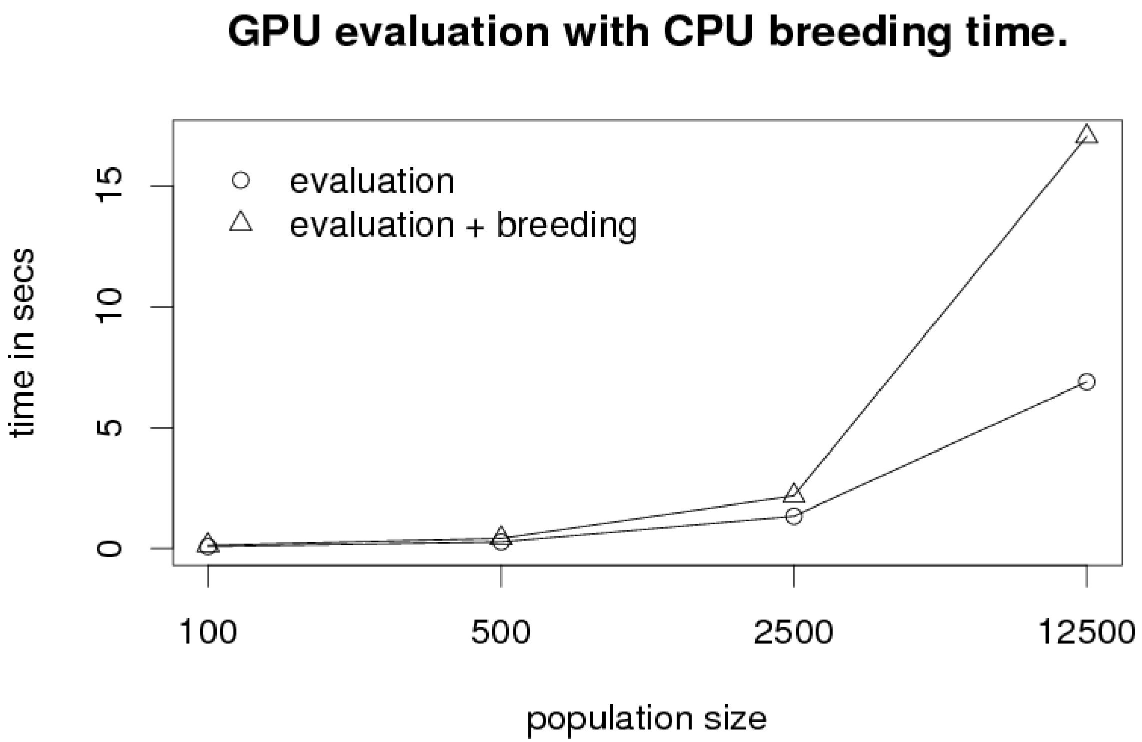
Alternative parallelization scheme

- **1 prog / block vs 1 prog / thread : Spirals**



Why does speedup decreases with larger populations ?

- **ECJ breeding cost dominates evaluation cost when populations grow larger !**
  - Here for regression, 1024 fitness cases:



Conclusions (I)

- Parallelize programs (not only training cases) in order to exploit a large number of elementary processors
- Use GPU architecture to achieve best speedups ... Available with all toolkits ?
- Divergence reduces significantly GPU performance
- 10 < speedup < 80 for small training sets and small programs for non-diverging function sets
- Best measured speed (evaluation phase, including memory transfer + postfixed translation): 120 millions GP nodes/s (vs 1.6 million GP nodes/s on CPU)

Conclusions (II)

- Integration into ECJ available:

http://www-lil.univ-littoral.fr/~robillia/EuroGP08/gpuregression.tgz

- Some lessons:
  - One needs to transfer data from Java to C via JNI (efficiency loss)
  - "Java is a memory hog" (S. Luke) => large populations need HUGE memory => add even more delay (garbage collecting,...)
  - as a result, CPU breeding time dominates GPU evaluation time for large populations...
  - porting breeding on GPU would mean a major fork from ECJ library...