A GPU Implementation of Large Neighborhood Search for Solving Constraint Optimization Problems

F. Campeotto[1,2] A. Dovier[1] F. Fioretto[1,2] E. Pontelli[2]

1. Univ. of Udine

2. New Mexico State University

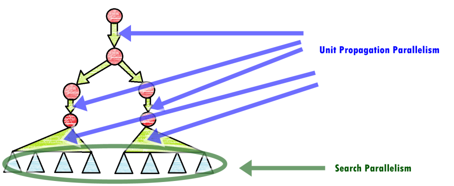
Prague, August 22nd, 2014

Introduction

- Every new desktop/laptop comes equipped with a powerful, programmable, graphic processor unit (GPU).
- For most of their life, however, there GPUs are absolutely idle (unless some kid is continuously playing with your PC)
- Auxiliary graphics cards can be bought with a very low price per computing core
- Their HW design is made for certain applications

Introduction

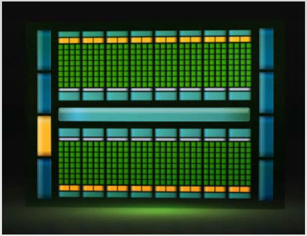
- In the last years we have experienced the use of GPUs for SAT solvers, exploiting parallelism either for deterministic computation or for non-deterministic search [CILC 2012–JETAI 2014]



- We have also used GPU for an ad-hoc implementation of LS solver for the protein structure prediction problem [ICPP13]
- We present here how we have converted our previous experience in the developing of a constraint solver with LNS.

GPUs, in few minutes

A GPU is a parallel machine with a lot of computing cores, with shared and local memories, able to schedule the execution of a large number of threads.

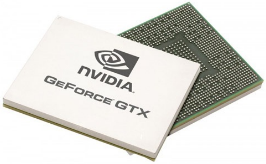


However, things are not that easy. Cores are organized hierarchically, and slower than CPUs, memories have different behaviors, . . . *it's not easy to obtain a good speed-up*

Do not reason as: 394 cores $\Rightarrow \sim 400\times$

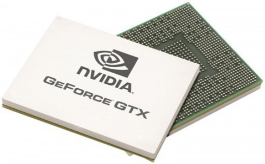
$10\times$ would be great!!!

CUDA: Compute Unified Device Architecture

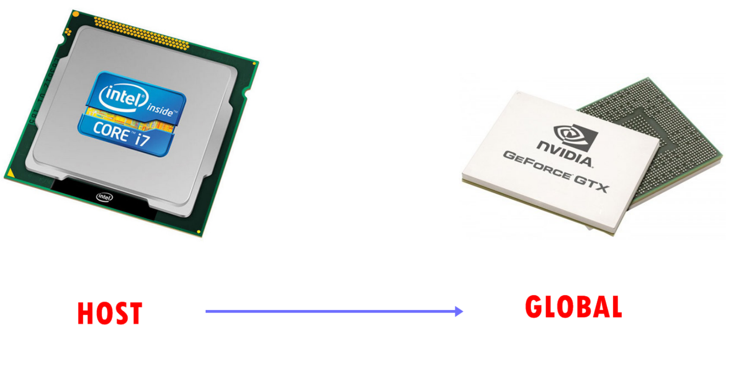


CUDA: Compute Unified Device Architecture

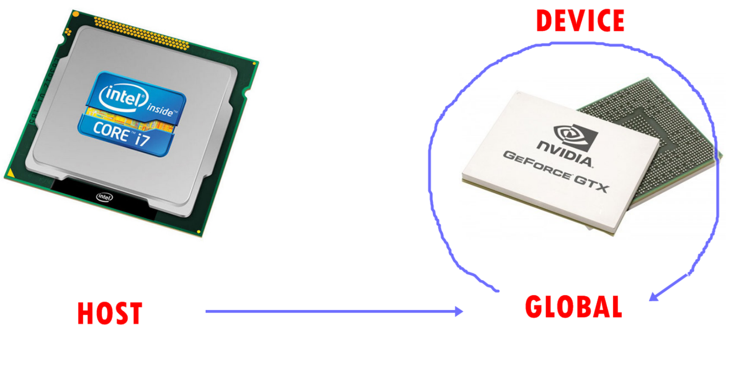
HOST



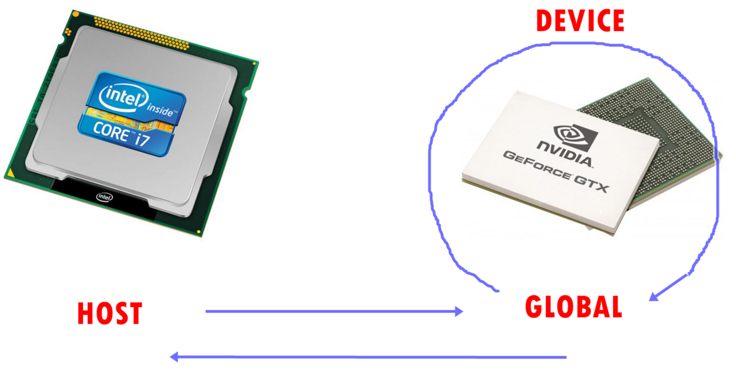
CUDA: Compute Unified Device Architecture



CUDA: Compute Unified Device Architecture

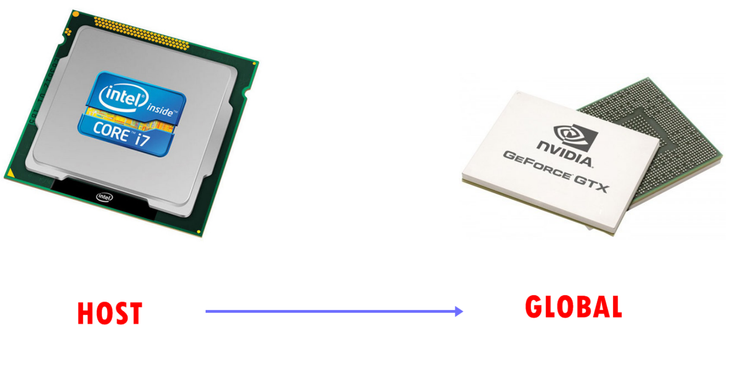


CUDA: Compute Unified Device Architecture

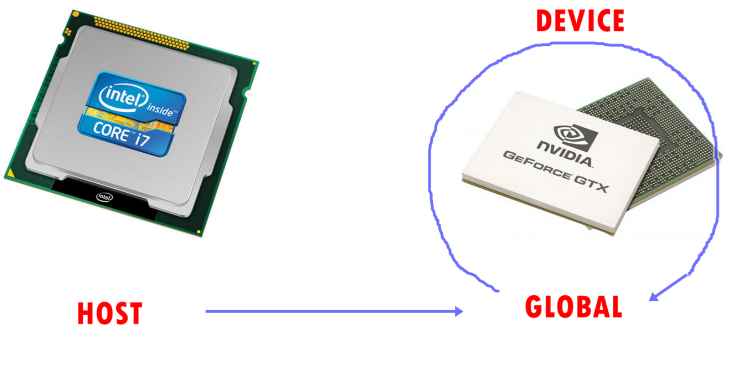


CUDA: Compute Unified Device Architecture

CUDA: Compute Unified Device Architecture



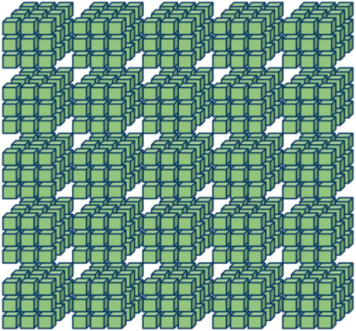
CUDA: Compute Unified Device Architecture



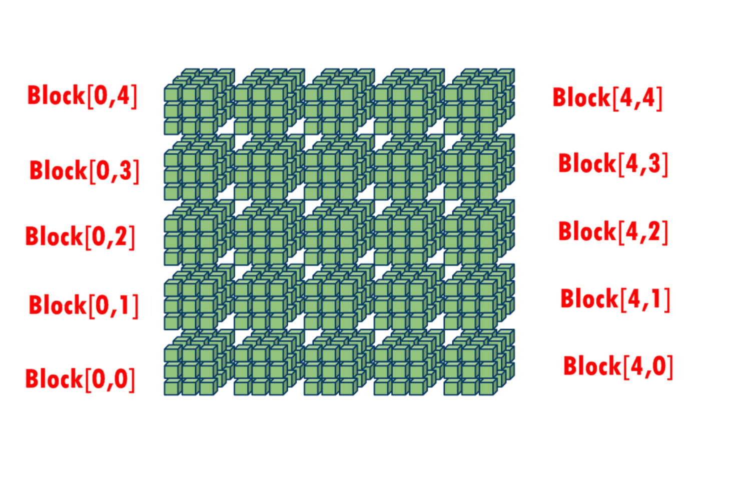
CUDA: Grids, Blocks, threads

- When a global (*kernel*) function is invoked, the number of parallel executions is established
- The set of all these executions is called a *grid*.
- A grid is organized in *blocks*
- A block is organized in a number of *threads*.
- The thread is therefore the *basic parallel unit* and it has a unique identifier (an integer number, a pair, or a triple):
  - its block `blockIdx` and
  - its position in the block `threadIdx`.
- This identifier is typically used to address different portions of a matrix
- The scheduler works with sets of 32 threads (warp) per time. A warp used SIMD (Single Instruction Multiple Data) in a warp: this must be exploited!

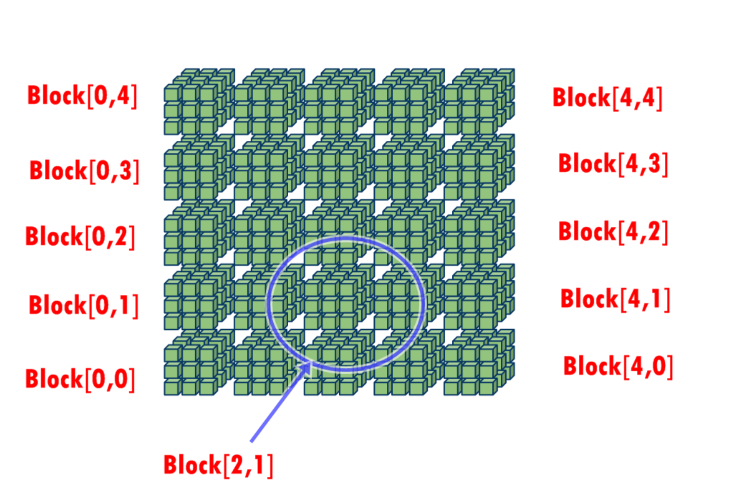
CUDA: Host, Global, Device



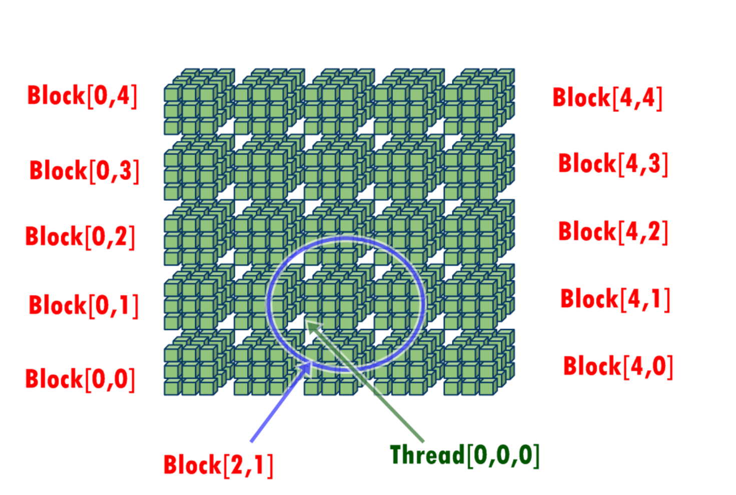
CUDA: Host, Global, Device



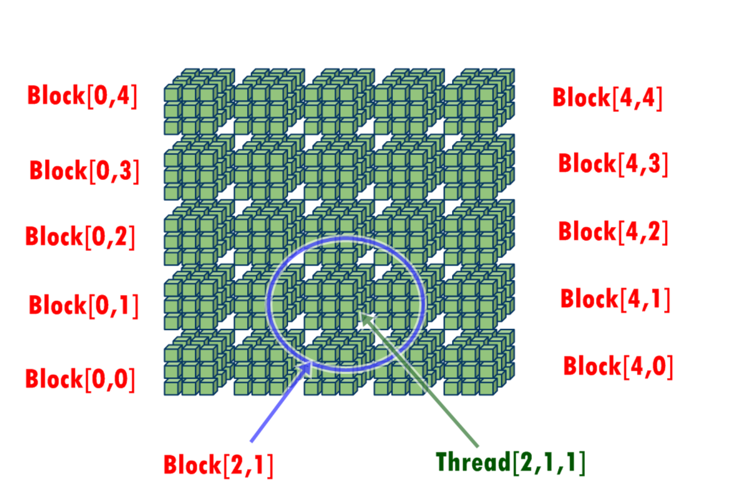
CUDA: Host, Global, Device



CUDA: Host, Global, Device

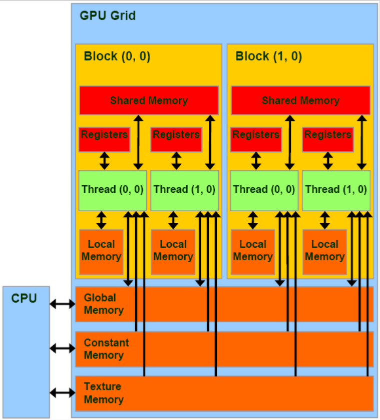


CUDA: Host, Global, Device



CUDA: Memories

The device memory architecture is rather involved, with 6 different types of memory (plus a new feature in CUDA 6)



The Solver iNVIDIOSO

NVIDIa-based cOnstraint SOlver

- Modeling Language: *MiniZinc*, to define a COP $\langle \vec{X}, \vec{D}, C, f \rangle$
- Translation from MiniZinc to *FlatZinc* is made by standard front-end (available in the MiniZinc distribution)
- We implemented *propagators* for “simple” FlatZinc constraints (most of them!)
- plus specific propagators for *some* global constraints
- There is a *device* function for each propagator (plus some alternatives)
- MiniZinc is becoming the *standard* constraint modeling language (e.g., for competitions)

The Solver iNVIDIOSO

Recent and current work

- We are exploiting GPUs for constraint propagation (more effective for “complex” constraints)
- We have comparable running time w.r.t. state of the art propagators (JaCoP, Gecode) but sensible speed-ups for some global constraints such as *table [PADL2014]*

The Solver iNVIDIOSO

Recent and current work

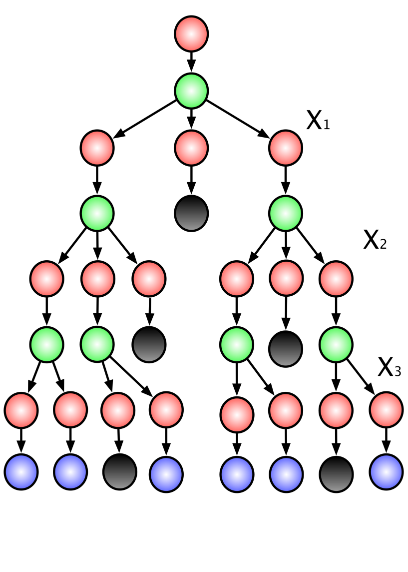
- We are exploiting GPUs for constraint propagation (more effective for “complex” constraints)
- We have comparable running time w.r.t. state of the art propagators (JaCoP, Gecode) but sensible speed-ups for some global constraints such as *table [PADL2014]*
- We have not (yet) implemented a real-complete parallel search (GPU SIMT is not made for that even if SAT experiments show that for suitable sizes it can work)

The Solver iNVIDIOSO

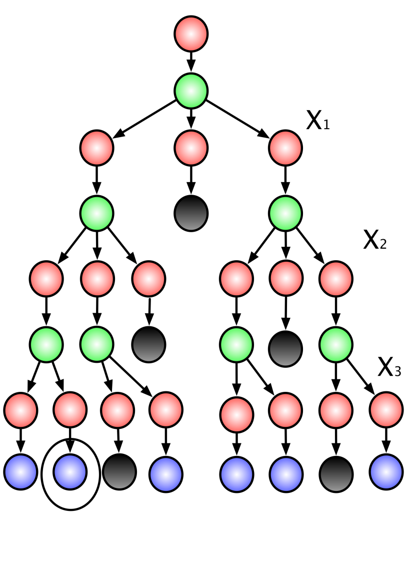
Recent and current work

- We are exploiting GPUs for constraint propagation (more effective for “complex” constraints)
- We have comparable running time w.r.t. state of the art propagators (JaCoP, Gecode) but sensible speed-ups for some global constraints such as *table [PADL2014]*
- We have not (yet) implemented a real-complete parallel search (GPU SIMT is not made for that even if SAT experiments show that for suitable sizes it can work)
- Rather, we have implemented a Large Neighborhood Search (LNS) on GPU *[this contribution]*
- LNS hybridizes Constraint Programming and Local Search for solving optimization problems (COPs).
- Exploring a neighborhood for improving assignments fits with GPU parallelism

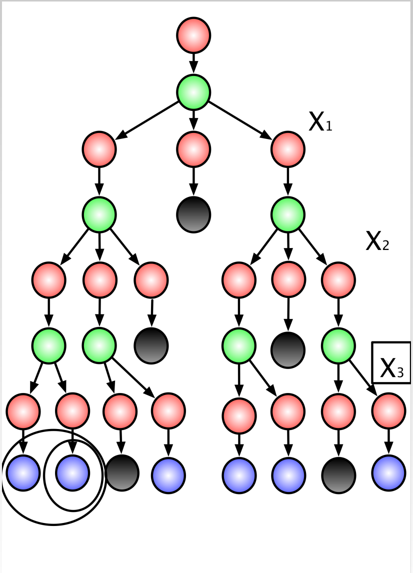
Small and Large Neighboorhood with CP



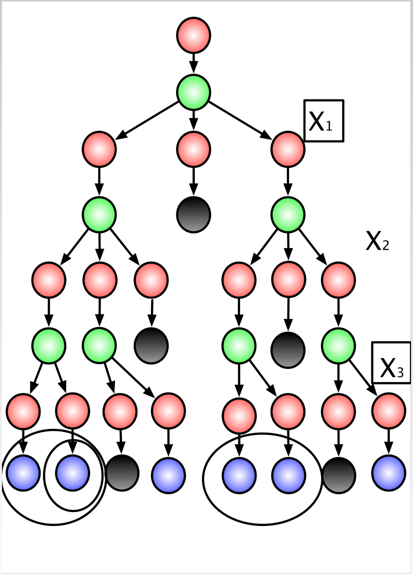
Small and Large Neighboorhood with CP



Small and Large Neighboorhood with CP



Small and Large Neighboorhood with CP



Large Neighborhood Search

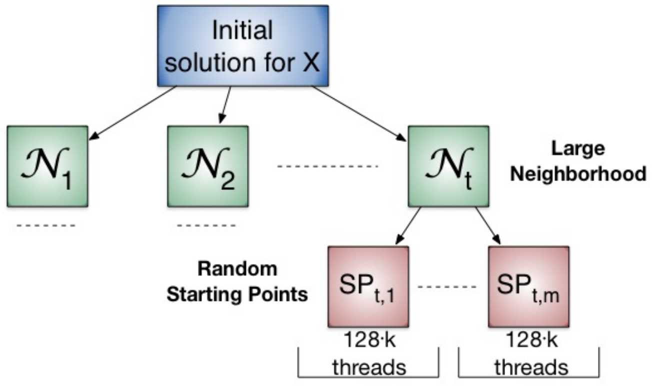
- Given a solution $\vec{s}$ for the COP $\langle \vec{X}, \vec{D}, C, f \rangle$ we can "unassign" some of the variables, say $\mathcal{N} \subseteq \vec{X}$
- The set of values for $\mathcal{N}$ that are a solution of the COP constitutes a *neighborhood* of $\vec{s}$ (including $\vec{s}$)
- Given the COP, $\mathcal{N}$ identifies uniquely a neighborhood (that should be explored)

Large Neighborhood Search

- Given a solution $\vec{s}$ for the COP $\langle \vec{X}, \vec{D}, C, f \rangle$ we can "unassign" some of the variables, say $\mathcal{N} \subseteq \vec{X}$
- The set of values for $\mathcal{N}$ that are a solution of the COP constitutes a *neighborhood* of $\vec{s}$ (including $\vec{s}$)
- Given the COP, $\mathcal{N}$ identifies uniquely a neighborhood (that should be explored)
- With GPUs we can consider many (large) neighborhoods in parallel each of them randomly chosen
- For each of them we consider different "starting points" (randomly chosen) from which starting the exploration of the neighborhood.
- We use parallelism to implement local search (and constraint propagation) within each neighborhood considering each starting point to cover (sample) large parts of the search space.

LNS: implementation

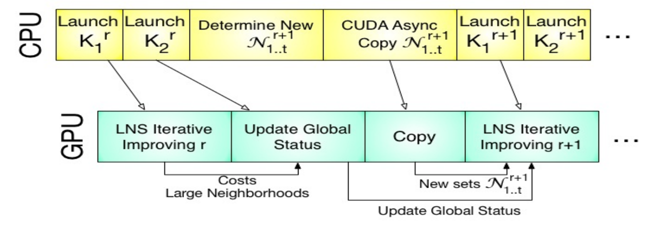
Parallelizing local search



LNS: implementation

Some details

- All constraints and initial domains are communicated to the GPU *once, at the beginning of the computation*
- The CPU calls a sequence of kernels K_i^r with $t \cdot m$ blocks (t subsets, m fixed number of initial assignments). r ranges with the number of improving steps.
- A block contains $128k$ threads ($1 \leq k \leq 8$ fixed)—$4k$ warps
- CPU and GPU work in parallel



LNS: implementation

Within each block

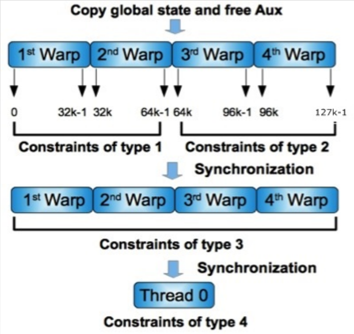
A block contains 128k threads, i.e., 4k warps (for simplicity assume now $k = 1$)

VARIABLES:

- FD (from the model)
- OBJ (one)
- AUX (for the obj function)

CONSTRAINTS:

- involving FD only
- involving FD and 1 AUX
- involving 2 or more AUX
- involving OBJ



LS techniques implemented

Random Labeling: randomly assigns values to the variables of the neighborhoods (i.e., MonteCarlo), possibly propagating constraints after each single assignment

Random Permutation: random permutation of the starting points

Two-exchange Permutations: swaps the values of all the pairs of variables in a neighborhood

Gibbs Sampling: Markov Chain Monte Carlo algorithm, used to solve a maximum a-posteriori estimation problem. Let s be the current solution and ν its cost. *for each* variable x in $\mathcal{N}$, choose a *random* candidate $d \in D^x \setminus \{s(x)\}$; then determine the new value ν' of the cost function, and accept or reject the candidate d with probability $\frac{\nu'}{\nu}$. Repeat p times.

Iterated Conditional Mode: similar to Gibbs but it performs gradient descent (hill climbing)

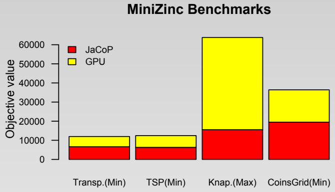
Complete exploration: try all possible combinations of assignments (unpractical!)

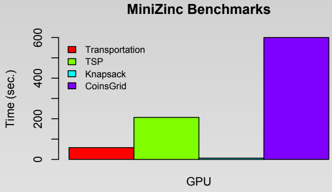
LNS: results

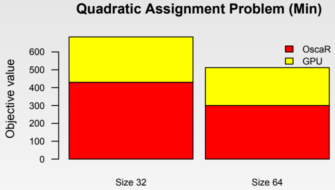
- LNS is developed for GPU.
- However, we have made some tests comparing the implementation with a CPU implementation of the same technique.
- Detailed results on the paper. *Speed-up from 2.5x to 40x* (best results on random labeling and on complete assignment)
- Let us see a comparison with JaCoP and Oscar

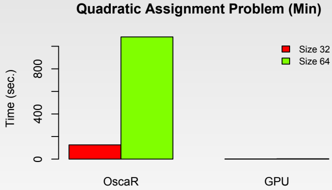
LNS: results

Graphs are one over the other, not behind!





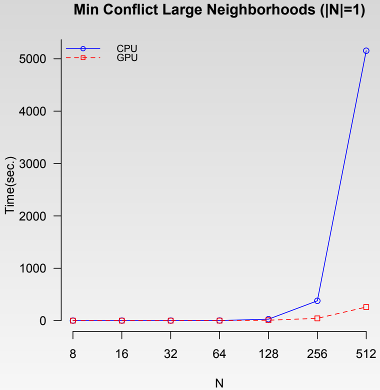


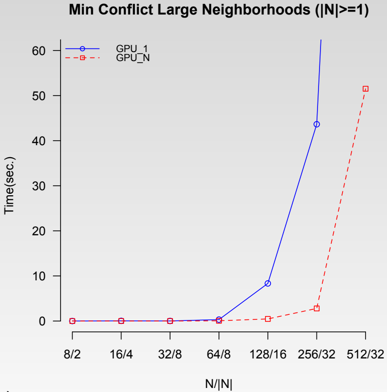


We tested CoinsGrid on OscaR (LNS). Both tools reach the timeout (600 s); we compute 25036 while Oscar 123262 (5x).

NEW: Some results on work in progress

LEFT: standard implementation of min-conflict ($|\mathcal{N}_i| = 1$)
RIGHT: Min Conflict Heuristic on Large Neighborhoods.





Tests on N queens (naive modeling)

Conclusions

- We have developed a constraint solver running (mostly) on GPU.
- Speed-up wrt sequential implementation
- Comparable with state-of-the-art solvers in the worst case, faster when (some) global constraints or LNS is used
- We are working for moving all computation to the GPU
- and/or we will try to exploit the (new, in CUDA 6) Unified Memory
- Standard (complete) search options and other basic constraints are now implemented
- GPUs will be used for parallel “search look-ahead” for choosing dynamically the most promising search strategy for a complete search
- The parallel propagation of other global constraints (e.g., alldifferent, circuit, cumulative, sets) will be soon investigated

Extra slides

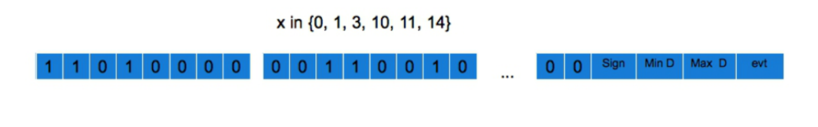
Just in case . . .

Some Remarks

- Heuristics chosen for the test with JaCoP are those that perform better for JaCoP (combination of first-fail/indomain_min, etc).
- TSP instances are on 240 cities (and some flux constraints)
- Knapsack instances are of 100 elements and made hard using an on-line generator (link in the paper) — few constraints.
- CoinsGrid problem instead has many constraints

Domain Representation

- Domain as a Bitset
- 4 extra variables are used: (1) sign, (2) min, (3) max, (4) event
- The use of bit-wise operators on domains reduces the differences between the GPU cores and the CPU cores



- Offsets are used (e.g. if $x \in \{-1000, -999\}$)
- The status is stored in a vector of nM integer (M a multiple of 32, n number of variables)