Exploring the Use of GPUs in Constraint Solving

A Preliminary Investigation

Federico Campeotto[1,2] Alessandro Dal Palù[3]
Agostino Dovier[1] Ferdinando Fioretto[1,2] Enrico Pontelli[2]

1. Università di Udine

2. New Mexico State University

3. Università di Parma

San Diego CA, January 2014

Introduction

- Every new desktop/laptop comes equipped with a powerful graphic processor unit (GPU)
- These GPUs are general purpose (i.e., we can program them)
- For most of their life, however, they are absolutely idle (unless some kid is continuously playing with your PC)
- The question is: can we exploit this computation power for constraint solving?
- We present a preliminary investigation, focusing on constraint solving

Constraint Satisfaction Problems

A *Constraint Satisfaction Problem* (*CSP*) is defined by:

- $X = \{x_1, \ldots, x_n\}$ is a n-tuple of variables
- $D = \{D^{x_1}, \ldots, D^{x_n}\}$ set of variable's domains
- C finite set of constraints over X: $c(x_{i_1}, \ldots, x_{i_m})$ is a relation $c(x_{i_1}, \ldots, x_{i_m}) \subseteq D^{x_{i_1}} \times \ldots \times D^{x_{i_m}}$.

A *solution* of a CSP is a tuple $\langle s_1, \ldots, s_n \rangle \in \times_{i=1}^{n} D^{x_i}$ such that for each $c(x_{i_1}, \ldots, x_{i_m}) \in C$, we have $\langle s_{i_1}, \ldots, s_{i_m} \rangle \in c$.
CSP solvers alternate 2 steps:

1. *Labeling*: select a variable and (non-deterministically) assign a value from its domain
2. *Constraint propagation*: propagate the assignment through the constraints, and possibly detect inconsistencies

Consistency techniques

Idea: replace the current CSP by a "simpler" one, yet equivalent

Definition (Arc Consistency)

The most common notion of local consistency is *arc consistency* (AC). Let us consider a binary constraint $c \in C$, where $\text{scp}(c) = \{x_i, x_j\}$ and $x_i, x_j \in X$. We say that c is arc consistent if:

- $\forall a \in D^{x_i} \exists b \in D^{x_j} (a, b) \in c$;
- $\forall b \in D^{x_j} \exists a \in D^{x_i} (a, b) \in c$;

- It is possible to ensure AC by iteratively removing all the values of the variables involved in the constraint that are not consistent with the constraint until a fixpoint is reached

Consistency techniques

Idea: replace the current CSP by a "simpler" one, yet equivalent

Definition (Arc Consistency)

The most common notion of local consistency is *arc consistency* (AC). Let us consider a binary constraint $c \in C$, where $\text{scp}(c) = \{x_i, x_j\}$ and $x_i, x_j \in X$. We say that c is arc consistent if:

- $\forall a \in D^{x_i} \exists b \in D^{x_j} (a, b) \in c$;
- $\forall b \in D^{x_j} \exists a \in D^{x_i} (a, b) \in c$;

- It is possible to ensure AC by iteratively removing all the values of the variables involved in the constraint that are not consistent with the constraint until a fixpoint is reached
- The propagation engine computes a mutual fixpoint of all the constraint

Consistency techniques

Idea: replace the current CSP by a "simpler" one, yet equivalent

Definition (Arc Consistency)

The most common notion of local consistency is *arc consistency* (AC). Let us consider a binary constraint $c \in C$, where $\text{scp}(c) = \{x_i, x_j\}$ and $x_i, x_j \in X$. We say that c is arc consistent if:

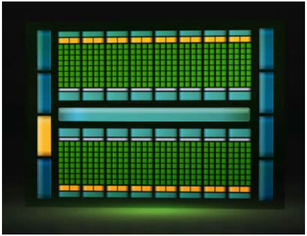
- $\forall a \in D^{x_i} \exists b \in D^{x_j} (a, b) \in c$;
- $\forall b \in D^{x_j} \exists a \in D^{x_i} (a, b) \in c$;

- It is possible to ensure AC by iteratively removing all the values of the variables involved in the constraint that are not consistent with the constraint until a fixpoint is reached
- The propagation engine computes a mutual fixpoint of all the constraint
- Several algorithms based on fixpoint loop iteration to achieve (Arc) Consistency: *AC3*, *AC4*, *AC6*, etc.

GPUs, in few minutes

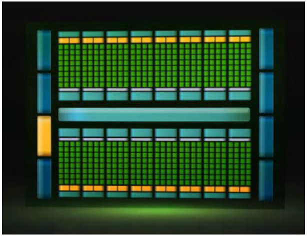
GPUs, in few minutes

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



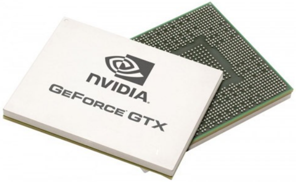
GPUs, in few minutes

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

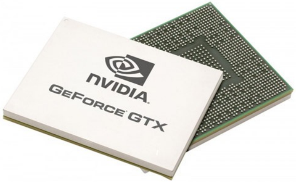


However, things are not that easy. Cores are organized hierarchically, memories have different behaviors, . . . it's not easy to obtain a good speed-up.

CUDA: Host, Global, Device

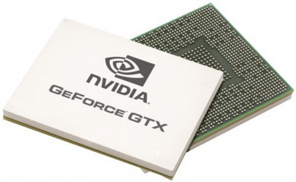


CUDA: Host, Global, Device



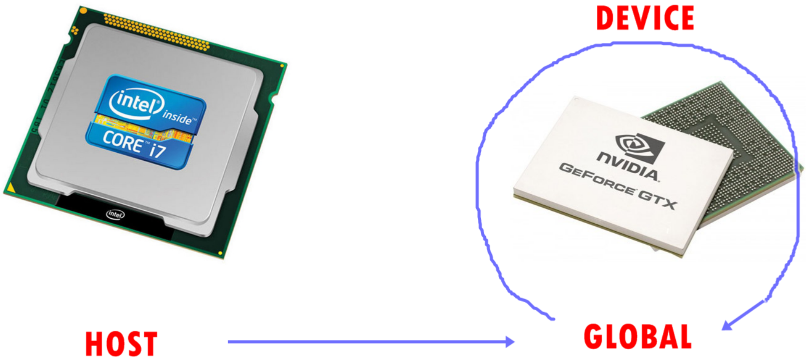
HOST

CUDA: Host, Global, Device

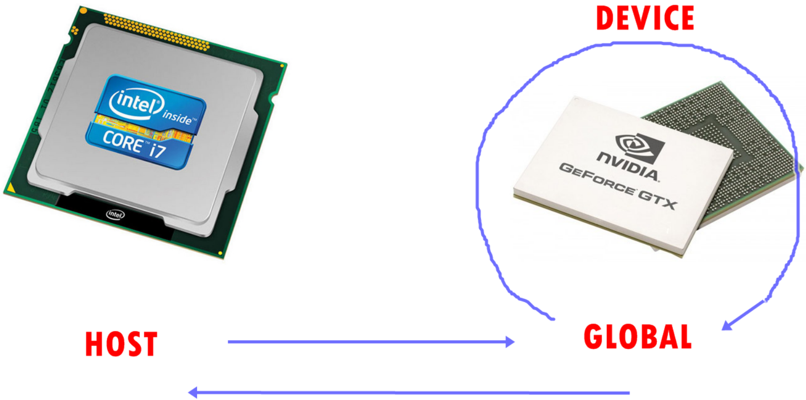


HOST → **GLOBAL**

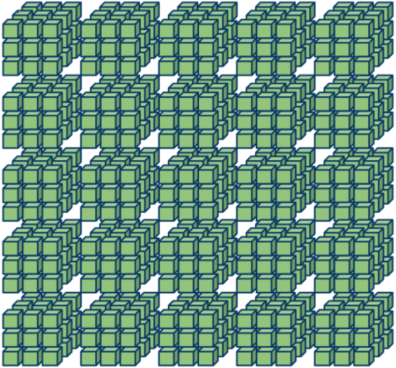
CUDA: Host, Global, Device



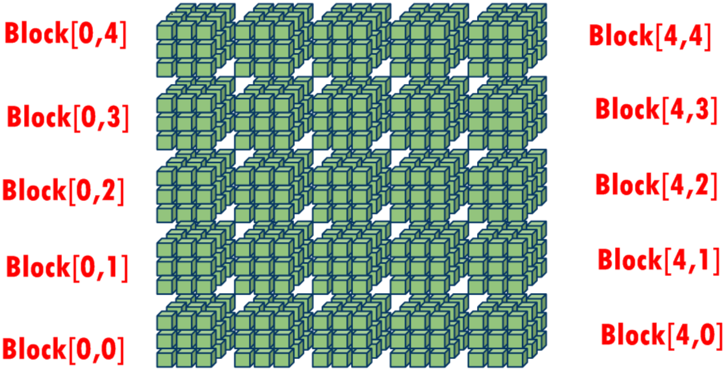
CUDA: Host, Global, Device



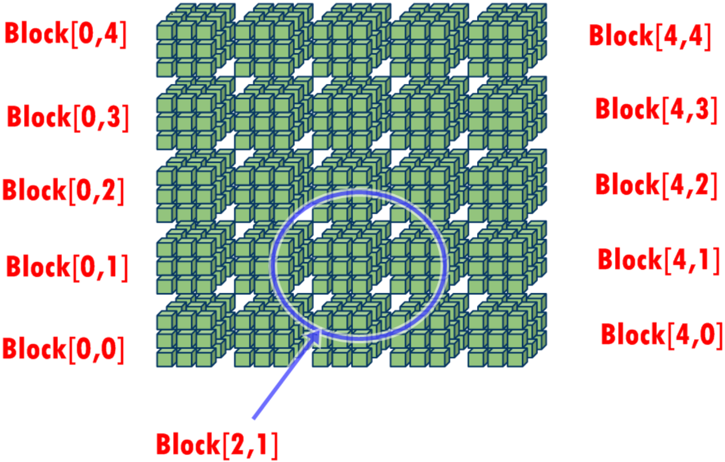
CUDA: Host, Global, Device



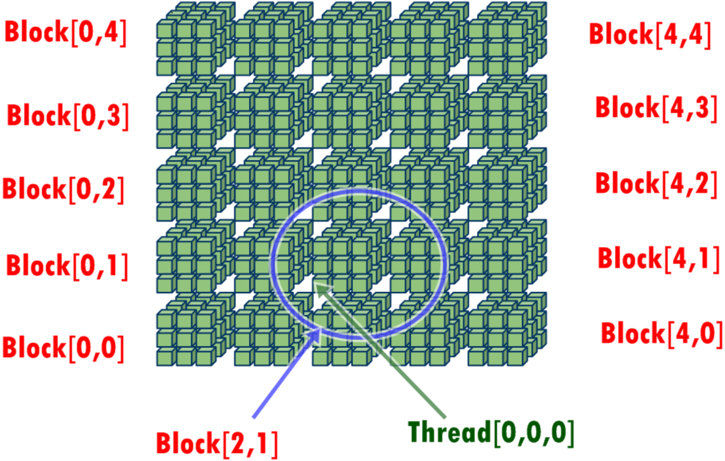
CUDA: Host, Global, Device



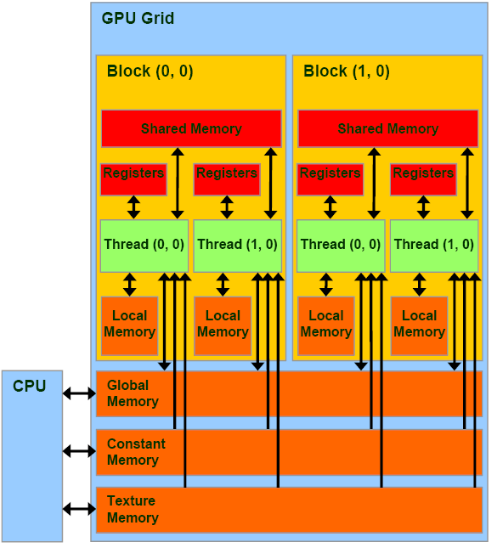
CUDA: Host, Global, Device



CUDA: Host, Global, Device



CUDA: Memories



How to. . .

- Can we perform propagation on GPGPUs?
- We will see a constraint engine that uses GPU to propagate constraints in parallel
- Several issues: memory accesses, slow GPU cores, data transfers, ...
- Different choices
- Preliminary results

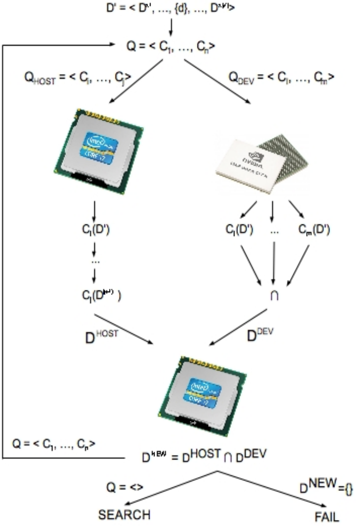
Parallel Constraint Solving: Parallel Consistency

- Establishing arc-consistency is P-complete;
- There are different parallel AC-based algorithms that can achieve $3, 4\times$ speedup;
- Two main parallel strategies:
  1. parallel AC algorithms using shared memory
  2. distributed AC algorithms
- We focus on a shared memory AC algorithm

Parallel AC algorithm - 1

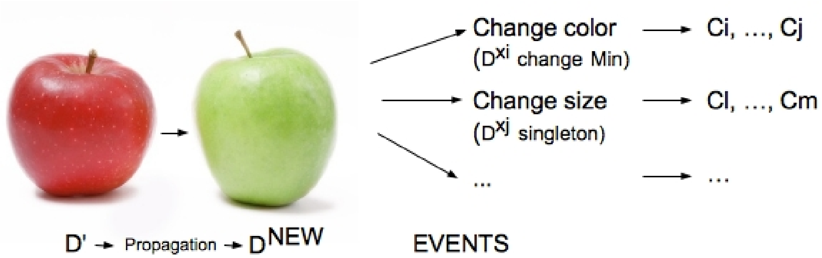
- Parallel algorithms for solving node and (bound) arc consistency;
- Strategy: check for consistency on all the arcs in the constraint queue simultaneously $\rightarrow \mathcal{O}(nd)$ instead of $\mathcal{O}(ed^3)$;
- We adopted 3 level of parallelism
  - *Constraints*: one parallel block for each constraint
  - *Variables*: one parallel thread for each variable
  - *CPU* for efficient propagators and *GPU* for expensive propagators

Parallel AC algorithm - 2



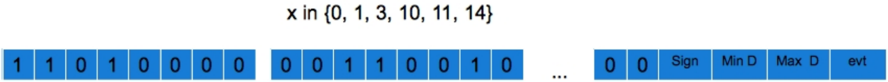
Parallel AC algorithm - 3

- The constraint engine is based on the notion of *events* (not AC3!)
- *Event*: a *change* in the domain of a variable
- The queue of *propagators* is updated accordingly...



Choices: Domain representation

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



Choices: Status representation

- The status of the computation is represented by a vector of $M \cdot |V|$ *integer* values where M is a multiple of 32
- We take advantage of the device cache, since global memory accesses are cached and served as part of $128-$*byte* memory transactions.
- *Coalesced* memory accesses: the accesses to the global memory are coalesced for contiguous locations in global memory

$V = \{x_1, \ldots, x_n\}$

x_1

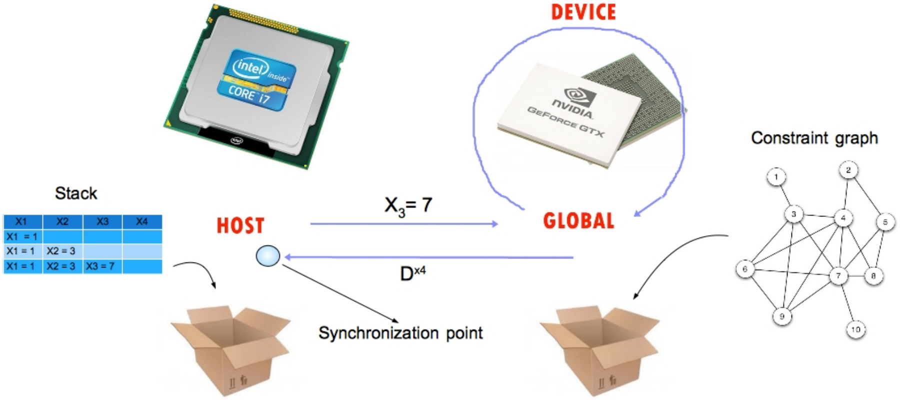
10..11	...	00..10	Sign	Min D_{x1}	Max D_{x1}	evt
1	2...27	28	29	30	31	32

...

x_n

00..01	...	11..10	Sign	Min D_{xn}	Max D_{xn}	evt
1	2...27	28	29	30	31	32

Choices: Data transfers



Choices: Propagators - 1

- Standard language for modelling CP problems: *Minizinc/FlatZinc*
- FlatZinc is a low-level solver-input (translated from Minizinc models)
- Our solver parses FlatZinc models
- We implemented propagators for the FlatZinc constraints plus specific propagators for some *global* constraints
- Every propagator is implemented as a specific device function invoked by a single block

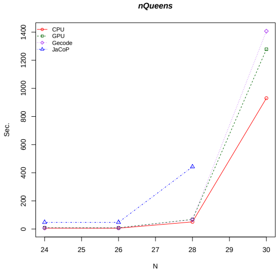
Choices: Propagators - 2

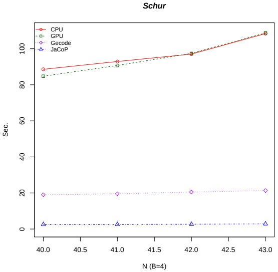
- Intuitive example: the *all_different* constraint C on the variables $x_1, \ldots, x_n$ can be naively encoded as a quadratic number of binary $\neq$ constraints
- It can be implemented by a set of n propagators $p_1, \ldots, p_n$: p_i takes care of the constraints $x_i \neq x_j$ where $j \neq i$

Algorithm 1

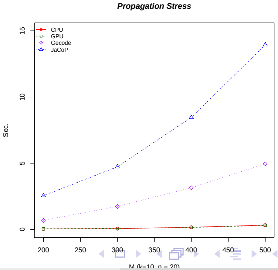
if $threadIdx \neq i$ **then**
    $x_j \leftarrow$ **scp**$(C)[threadIdx]$;
    $D^{x_j}[x_i] \leftarrow 0$;
end if

Results





- *Host*: AMD Opteron 270, 2.01GHz, RAM 4GB
- *Device*: NVIDIA GeForce GTS 450, 192 cores (4MP). Processor Clock 1.566GHz.



Main drawbacks

- Data transfers: many failures imply more backtrack actions and more copies between host and device
- GPU memory latency and coalesced access patterns
- Difference between the GPU clock and the CPU clock
- We can partially reduce some of these issues using an *Upper bound* parameter: if the number of CPU-propagators if higher than a given *upper bound*, they are all propagated on GPU

Main drawbacks

- We can improve the performance using an *Upper bound* parameter: if the number of CPU-propagators if higher than a given *upper bound*, they are all propagated on GPU
- We handle the cases where a large number of efficient propagator are assigned to the CPU, while they could take advantage of parallel propagation

Example: *Golomb* ruler

CPU	UB = 0	UB = 100	UB = 500	UB = 1000	UB = 1500
266.4	223.4	216.4	214.2	210.4	207.8

Global constraints

- A higher speedup can be achieved with expensive constraints: more parallel work to do!
- We considered two expensive global constraints:
  - the *inverse* constraint: $inverse(x, y)$ holds if y is the inverse function of x (and vice versa)
  - the *table* constraint: it enforces that tuple of variables takes a value from a set of tuples

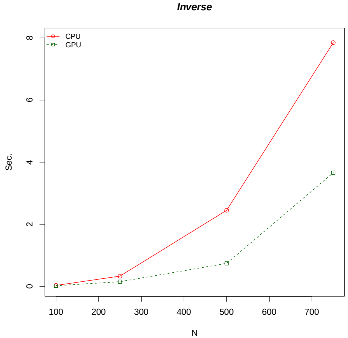


Table Instance	CPU	GPU	Speedup
CW-m1c-lex-vg4-6	0.015	0.005	3.00
langford-2-50	44.06	15.16	2.94
CW-m1c-uk-vg16-20	1.488	0.225	6.61
ModRen_0	0.381	0.154	2.74
CW-m1c-lex-vg7-7	209.4	43.87	4.77
ModRen_49	0.317	0.117	2.74
langford-2-40	136.4	46.39	2.90
RD_k5_n10_d10_m15	0.138	0.053	2.60

Conclusions (or Start?)

- The question was: can we exploit GPGPUs computational power for constraint solving?

Conclusions (or Start?)

- The question was: can we exploit GPGPUs computational power for constraint solving?
- The answer is yes but ...

Conclusions (or Start?)

- The question was: can we exploit GPGPUs computational power for constraint solving?
- The answer is yes but . . .
- First results are encouraging, especially for *global* constraints, so . . .

Conclusions (or Start?)

- The question was: can we exploit GPGPUs computational power for constraint solving?
- The answer is yes but . . .
- First results are encouraging, especially for *global* constraints, so . . .
- GPUs can be used for effective exploitation of parallelism in the case of domain-specific constraints with complex propagation strategies

Conclusions (or Start?)

- The question was: can we exploit GPGPUs computational power for constraint solving?
- The answer is yes but ...
- First results are encouraging, especially for *global* constraints, so ...
- GPUs can be used for effective exploitation of parallelism in the case of domain-specific constraints with complex propagation strategies
- Parallel constraint propagation on GPGPUs should be used depending on the type of problem: real-world problems with complex constraints are good!

Conclusions (or Start?)

- The question was: can we exploit GPGPUs computational power for constraint solving?
- The answer is yes but ...
- First results are encouraging, especially for *global* constraints, so ...
- GPUs can be used for effective exploitation of parallelism in the case of domain-specific constraints with complex propagation strategies
- Parallel constraint propagation on GPGPUs should be used depending on the type of problem: real-world problems with complex constraints are good!
- We experimented with an ad-hoc constraint-based implementation of protein structure prediction via fragment assembly, parallelized on GPUs using similar techniques, with excellent performance results (up to $50\times$)

Conclusions (or Start?)

- The question was: can we exploit GPGPUs computational power for constraint solving?
- The answer is yes but ...
- First results are encouraging, especially for *global* constraints, so ...
- GPUs can be used for effective exploitation of parallelism in the case of domain-specific constraints with complex propagation strategies
- Parallel constraint propagation on GPGPUs should be used depending on the type of problem: real-world problems with complex constraints are good!
- We experimented with an ad-hoc constraint-based implementation of protein structure prediction via fragment assembly, parallelized on GPUs using similar techniques, with excellent performance results (up to $50\times$)
- Future work: combine parallel search with parallel constraint propagation

Conclusions (or Start?)

Meanwhile, using GPU in the correct way: