GPU-based parallelism for ASP-solving

Andrea Formisano

Dept. of Mathematics and Computer Science
University of Perugia
andrea.formisano@unipg.it

DECLARE 2019 — Cottbus, September 9–13, 2019

Joint research with A. Dal Palù, A. Dovier, E. Pontelli, F. Vella

Outline

General Purpose GPU

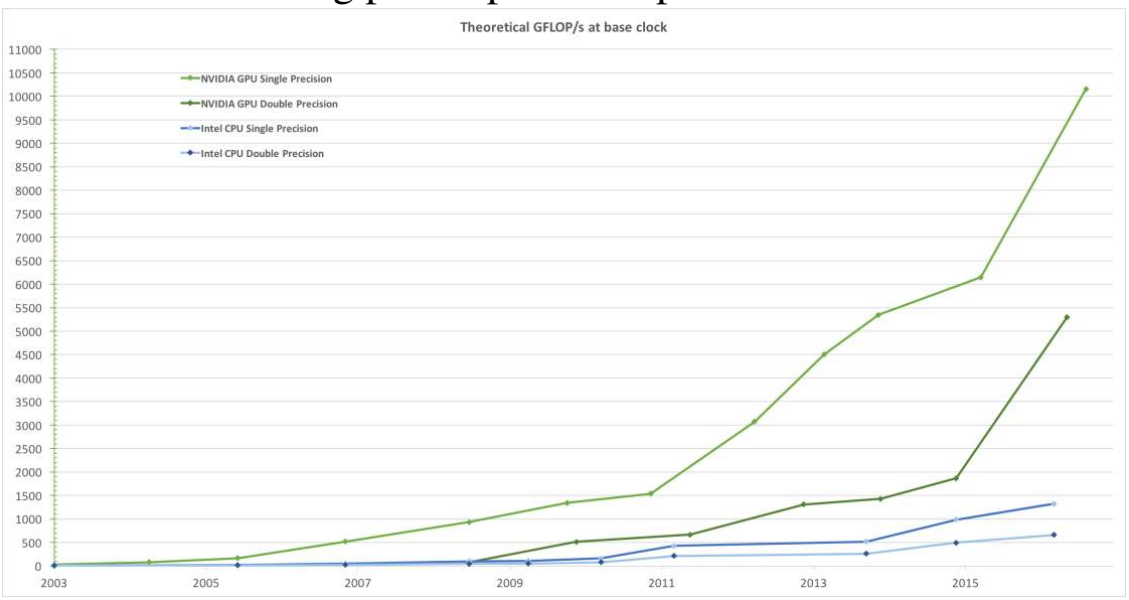
- Graphic Processing Units (GPUs) have been originally conceived for graphic processing (e.g., realtime high resolution 3D graphics)
- highly parallel multi-threaded many-core processors with high memory bandwidth
- in the last decade GPUs evolved towards a more flexible architecture enabling the use of GPUs for general purpose programming:

  GPU-computing

- GPUs offer great efficiency and high performance (if carefully programmed...)

GPUs vs CPUs

GPU vs CPU: Floating point operations per second

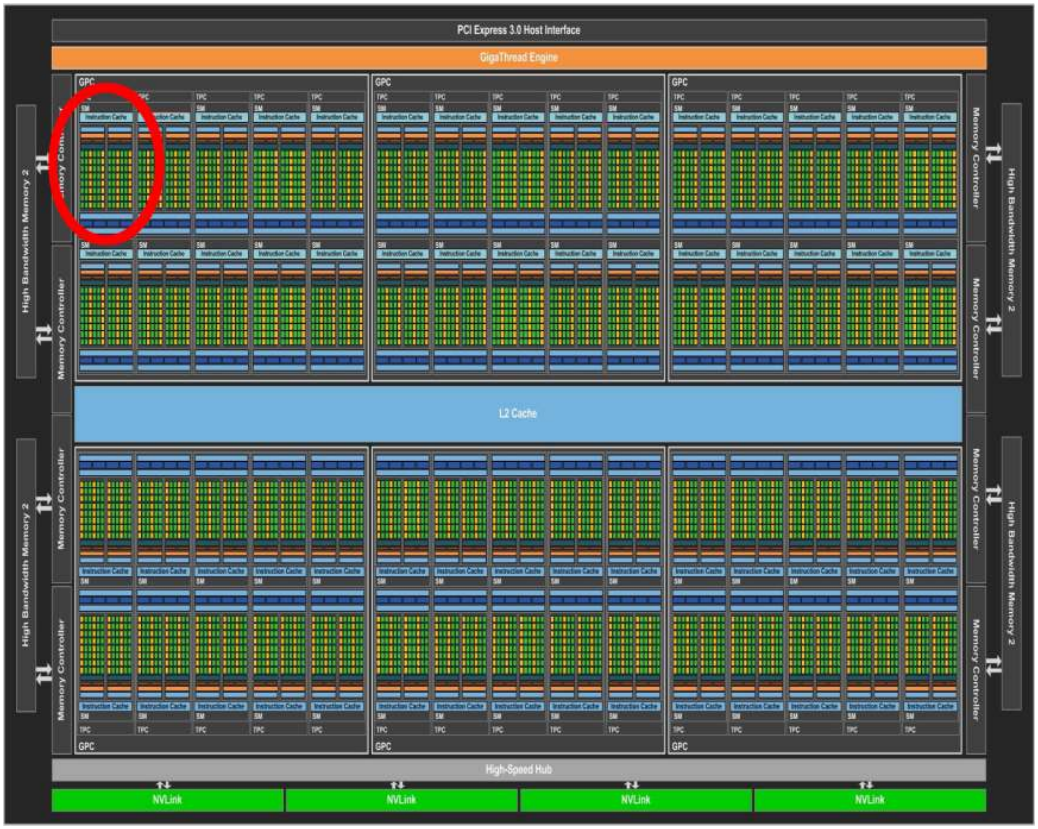


This and the following pictures regarding GPUs are from *CUDA C Programming Guide*, Nvidia, www.nvidia.com, or from the Nvidia web site.

CPU: complex control logic, out-of-order exec., branch-prediction, speculative exec., powerful ALU, large cache to reduce latency in memory access, ...

GPU: many cores, massive floating point parallel computations, larger bandwidth in memory access, simple control logic, no branch prediction or out-of-order exec., ...

Transistors are mainly devoted to data processing rather than data caching and flow control



Zoom in: A stream multiprocessor (SM)

Zoom in: Each SM includes

- cores
- registers
- shared memory
- scheduling and dispatch units
- special function units
- LD/ST units
- cache, ...

Some Tesla GPUs:
K40: 15 SM, 2880 cores
M40: 24 SM, 3072 cores
P100: 56 SM, 3584 cores
V100: 80 SM, 5120+460 cores

CUDA framework for GPU computing

CUDA — Compute Unified Device Architecture
(introduced by NVIDIA in 2006)

- general-purpose parallel computing platform and programming model
- provides compilers, libraries, API, drivers, devel. tools, ...
- exposes GPU compute capabilities of any CUDA-enabled GPU
- enable access to GPU memory
- extends common programming languages (C, C++, Fortran, ...) in a minimal way. E.g. in CUDA-C:
  DEFINITION:
  ```
  __global__ void procName(FormalArgs){...C code...}
  ```
  CALL:
  ```
  procName<<<Grid,TPB>>>(ActualArgs)
  ```


Execution model (CUDA-style)

The underlying execution model is named

SIMT: single-instruction multiple-thread

- multiple independent threads execute a single instruction
- SIMT = SIMD+multithreading

The SIMT model differs from SIMD in various aspects:

- each thread has its own program counter
- each thread has its own register state (i.e., it has a register set)
- each thread can have an independent execution path (namely, threads in the same *group* can execute independently)

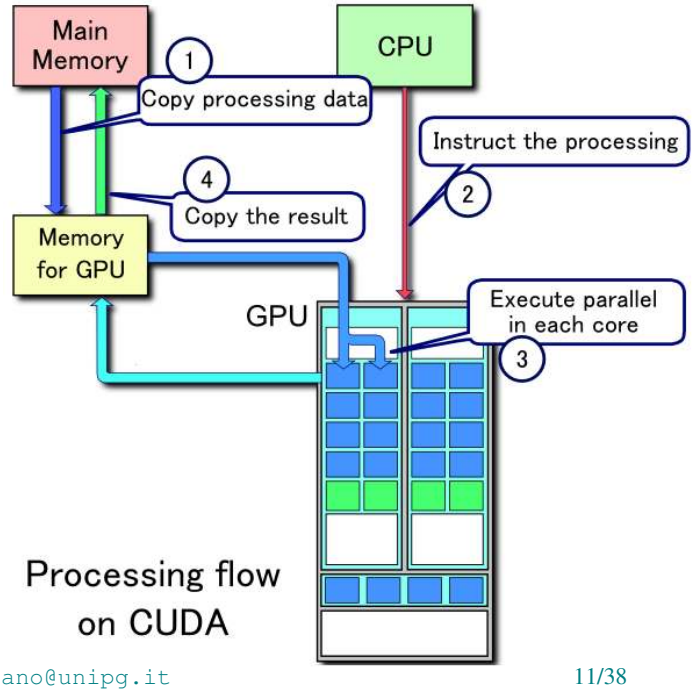
Execution model (CUDA-style)

The computation can proceeds both on the host (CPU) and on the device (GPU)

- The programmer writes a kernel that will be run on the device
- Each thread executes an instance of the kernel

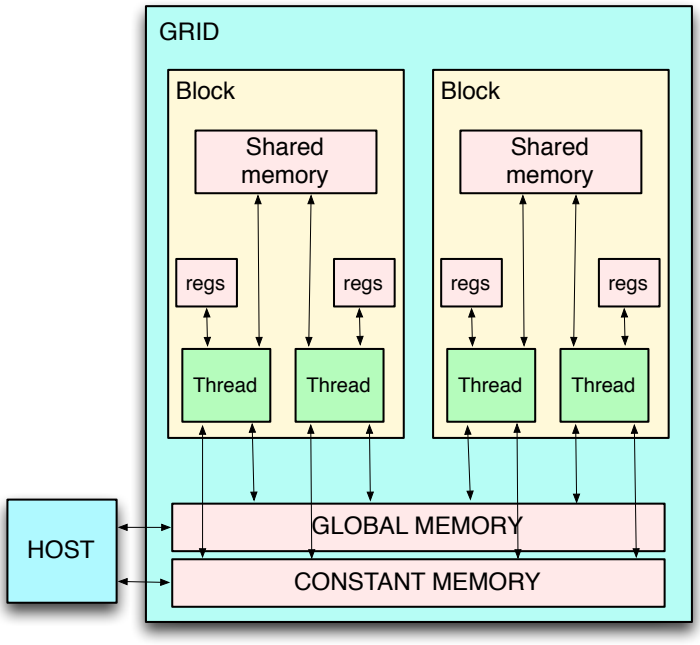
The host instructs the device:

1. copy data, host⇒device
2. kernel call
3. kernel execution on GPU
4. retrieve results, host⇐device



Thread and memory hierarchies (CUDA-style)

- Each core executes a thread
  - registers
  - local memory
- warp: 32 threads
  - work in lock-step: share instruction fetch
- block: a group of threads
  - scheduled on a SM
  - shared memory (banks)
  - synchronization support
- grid: a group of blocks
  - global memory
  - constant, texture mem.

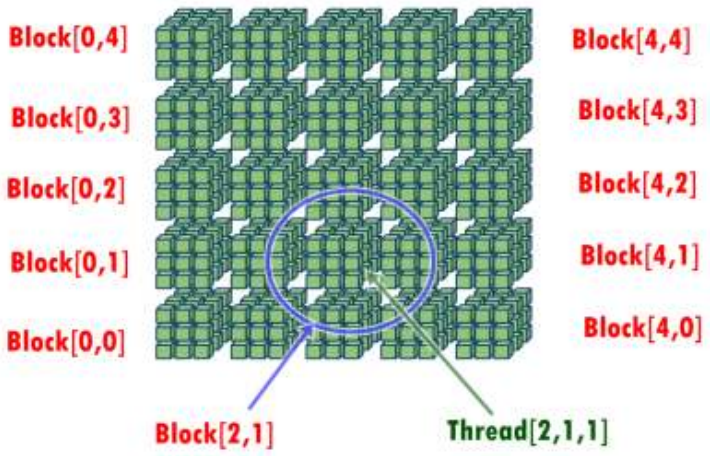


Grids, blocks, and threads

The host calls a grid (of kernels) to be executed on the device:

```
kernelName<<<GridDim,BlockDim>>>(ActualArgs)
```

- a grid is a 1D/2D/3D matrix of blocks
- in turn, each block is a (1D/2D/3D) matrix of threads
- each thread has its own ID (usually used to address different data)
- example: 2D grid of 3D blocks



Efficiency in CUDA programming

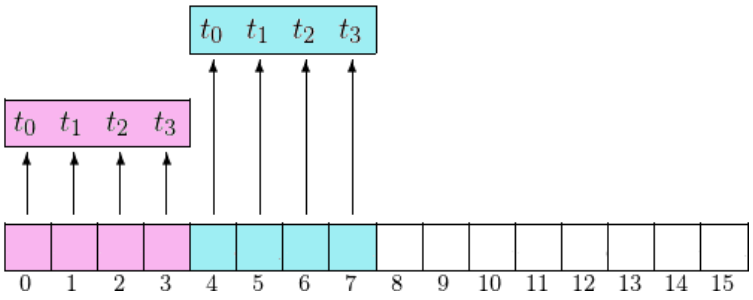
The rigid memory hierarchy and the SIMT execution model require careful programming...
...many things to take care of:

- maximize number of active threads, occupancy
  $\Rightarrow$ configure grid/block dimensions, shared mem., registers,...
- maximize device utilization, overlap memory transfers and computations
  $\Rightarrow$ use streams, concurrent kernels,...
- prefer shared memory to global memory
- use intra-warp communication/synchronization (shuffling, voting, matching functions)
- avoid thread divergence
- impose coalescence in global memory accesses
  $\Rightarrow$ use coalesced accesses
- avoid bank conflicts in shared memory accesses
  $\Rightarrow$ use strided accesses

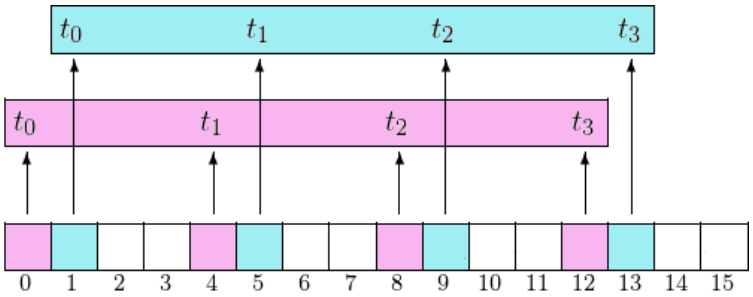
Strided and coalesced accesses

Example: Four threads accessing elements of an array:

coalesced:



strided:



Answer Set Programming (ASP)

- A successful form of Logic Programming paradigm
- Knowledge representation and Non-monotonic reasoning (default negation)
  - Logical theories serve as problem specifications
  - Solutions are described by models of the theories
- Strong theoretical foundation: it originates from extensive research on semantics of LP with negation
- Expressive power: it captures (in its simplest form) the *NP* complexity class
- Efficient inference engines

ASP syntax

An ASP program Π is a collection of propositional rules of the form

$$r: \quad p \leftarrow p_1, \ldots, p_m, \mathit{not}\ p_{m+1}, \ldots, \mathit{not}\ p_n$$

- p and $\{p_1, \ldots, p_m, \mathit{not}\ p_{m+1}, \ldots, \mathit{not}\ p_n\}$ are denoted by $\mathit{head}(r)$ and $\mathit{body}(r)$, resp.
- $\{p_1, \ldots, p_m\}$ is denoted by $\mathit{body}^+(r)$
- $\{p_{m+1}, \ldots, p_n\}$ is denoted by $\mathit{body}^-(r)$

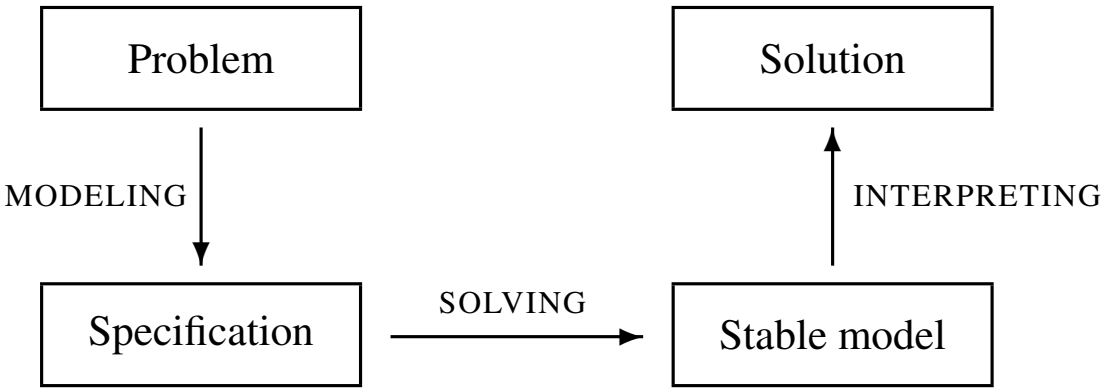
ASP semantics

- Semantics of ASP program Π is given in terms of stable models (or *answer sets*)
- A set M of atoms is a stable model for Π if it is the least Herbrand model of the reduct Π^M obtained by
  - removing from Π all rules r such that $M \cap \mathit{body}^-(r) \neq \emptyset$; and
  - removing all negated atoms from the remaining rules

Model-based problem solving in ASP

Typical approach in ASP:

- An ASP program (i.e., a logical theory) serves as problem specification
- Each stable model encodes a solution



Models, Completion, and Loop-formulas

Given a program Π, its completion Π_{cc} is defined as:

$$\Pi_{cc} = \left\{\beta_r \leftrightarrow \bigwedge_{a\in body^+(r)} a \wedge \bigwedge_{b\in body^-(r)} \neg b \mid r \in \Pi\right\} \cup \left\{p \leftrightarrow \bigvee_{r\in body_\Pi(p)} \beta_r \mid p \in atom(\Pi)\right\}$$

Given a set of atoms U, the set of external bodies for U is defined as

$$EB_\Pi(U) = \{\beta_r \mid r \in \Pi, body^+(r) \cap U = \emptyset\}$$

The loop formula of U is

$$\Pi_{lf}(U) = \bigvee_{p\in U} p \rightarrow \left(\bigvee_{\beta\in EB_\Pi(U)} \beta\right)$$

The set of loop formulas for Π is

$$\Pi_{lf} = \{\Pi_{lf}(U) \mid U \text{ is a loop in } \mathcal{D}^+_\Pi\}$$

Property:

A set of atoms is a stable model of Π iff it satisfies $\Pi_{cc} \wedge \Pi_{lf}$

Nogoods ...in a nutshell

Intuitively, a nogood δ is a forbidden set/conjunction of literals

An assignment A (i.e., a set of lits.) is a solution for δ if $\delta \not\subseteq A$

A nogood δ such that $\delta \cap A = \{p\}$ can be used to infer the need to add $\bar{p}$ to A

The completion Π_{cc} can be "compiled" into a collection $\Delta_{\Pi_{cc}}$ of completion nogoods of the forms:

- $\{not\ \beta_r\} \cup \{a \mid a \in body^+(r)\} \cup \{not\ b \mid b \in body^-(r)\}$
- $\{\beta_r, not\ a\}$ for each $a \in body^+(r)$ and $\{\beta_r, b\}$ for each $b \in body^-(r)$

for each r in Π, and

- $\{not\ p, \beta_r\}$ for each $r \in body_\Pi(p)$, for each head p in Π
- $\{p\} \cup \{not\ \beta_r \mid r \in body_\Pi(p)\}$, for each head p in Π

Similarly one introduces a set Λ_Π of loop nogoods to reflect loop formulas

Nogood-driven ASP-solving

Considering a program Π and a complete assignment A, we have that

A corresponds to a stable model iff it is a solution for $\Delta_{\Pi_{cc}} \cup \Lambda_\Pi$

Then,

- given $\Delta_{\Pi_{cc}} \cup \Lambda_\Pi$, a *DPLL-like* procedure can be used to find the stable models of Π
- PROBLEM: there can be *too many* loop nogoods
- SOLUTION: distinguish between completion nogoods (static) and loop nogoods (dynamic)
- generate static nogoods by compiling Π
- lazy generation of dynamic nogoods as soon as unfounded sets (i.e., *loops lacking any external support*) are detected

The state-of-the-art ASP-solver `clasp` uses a conflict-driven procedure and fruitfully adapts SAT-technology (conflict analysis, learning, backjumping, forgetting, ...)

Ingredients for a nogood-driven solver

Considering the basic solving procedure exploited in `clasp`, one identifies these main sub-tasks:

- Preprocessing: parse the input; compute completion nogoods, dependency graph, statistics for heuristics,...
- Selection: select/assign next branching atom (decision step)
- Propagation: propagate the consequences of decision steps
- Nogood-Check: look for violations of nogoods
- Unfounded-Set-Check: add dynamic nogoods, if any unfounded set is detected
- Conflict-Analysis: in case of conflict, learn new nogoods
- Backjumping: in case a conflicting partial assignment is reached, update data structures consequently

Question: can we design efficient CUDA-based parallel versions of these tasks?

Ingredients for a nogood-driven solver

Parallelization is "natural" for tasks that are executed for collections of data, such as:

- Preprocessing: nogood generation, heuristics evaluation, ...
- Selection: ranking of candidate selectable atoms, in parallel
- Propagation: perform all propagations in parallel
- Nogood-Check: check nogoods for violations, in parallel
- Backjumping: update data structures, in parallel (and similarly for other tasks, such as *forgetting*, *restarting*,...)

Some tasks turn out to be hardly parallelizable, in particular

Unfounded-Set-Check and Conflict-Analysis

Because they

- involve intrinsically sequential computations
- perform highly irregular and hardly predictable accesses to data

let's find some alternatives...

Avoiding loop-formulas: ASP-computations

An alternative characterization of stable models:

An ASP-computation is a sequence of sets of atoms $I_0 = \emptyset, I_1, I_2, \ldots$ s.t.

- $I_i \subseteq I_{i+1}$ for all $i \geq 0$ (Persistence of Beliefs)
- $I_\infty = \bigcup_{i=0}^{\infty} I_i$ is such that $T_\Pi(I_\infty) = I_\infty$ (Convergence)
- $I_{i+1} \subseteq T_\Pi(I_i)$ for all $i \geq 0$ (Revision)
- if $p \in I_{i+1} \setminus I_i$ then there is a rule $p \leftarrow body$ in Π such that $I_j \models body$ for each $j \geq i$ (Persistence of Reason)

(where T_Π is the usual *immediate consequence operator* of definite LP)

Prop: M is a stable model of Π iff there exists an ASP-computation that converges to M, namely, $M = \bigcup_{i=0}^{\infty} I_i$

Parallelizing conflict analysis

Intuitively, in `clasp` conflict analysis proceeds as follows:

1. a conflict arises whenever two nogoods δ, ε propagate opposite values for an atom $p \in \delta$
2. a new (intermediate) nogood is obtained
$$\delta' = (\delta \setminus \{p\}) \cup (\varepsilon \setminus \{\overline{p}\})$$
3. δ' is conflicting as well, hence update $\delta = \delta'$ and repeat the process until the 1UIP is reached: such δ is the learned nogood (in practice: stop as soon as δ contains exactly one atom assigned at the conflict decision level)

Parallel implementations of this schema exhibits poor performance

Efficient parallel conflict analysis

A simple procedure, alternative to the resolution-based learning, is as follows:

1. introduce a set $Deps(p)$ for each atom p
2. whenever a decision step selects p, set $Deps(p) = \{p\}$
3. whenever an atom p is propagated because of a nogood δ, set $Deps(p) = \bigcup_{d \in (\delta \setminus \{p\})} Deps(d)$
4. if a conflict arises because two nogoods δ, ε propagate opposite values for $p \in \delta$, then obtain a learned nogood as: $Deps(p) = \bigcup_{d \in (\delta \cup \varepsilon)} Deps(d)$

All steps can be executed in parallel by exploiting:

- a bitmap representation of $Deps(p)$
- a logarithmic reduction schema for computing unions
- shared memory for intermediate results
- shuffling functions for fast intra-warp data exchange

GPU-based ASP-computation exploiting nogoods

Summing up: `yasmin` is a prototypical solver that:

- exploits GPUs and the CUDA framework
  $\Rightarrow$ massive parallelism for all tasks
- adopts a nogood-driven approach
  $\Rightarrow$ SAT/ASP technology, heuristics, learning,...
- relies on ASP-computations
  $\Rightarrow$ focus on completion nogoods (avoid loop nogoods)
- uses parallelizable conflict analysis
  $\Rightarrow$ alternative learning strategy

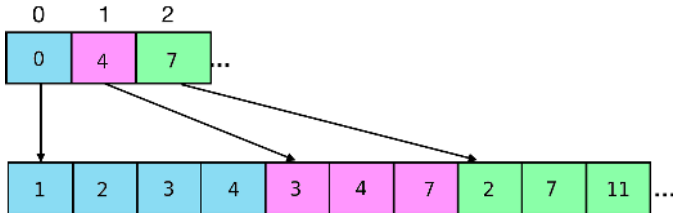
Basic schema of the CUDA application

Algorithm 1: Host code of YASMIN (simplified)

procedure YASMIN(Δ: Nogoods, P: Program)
$cdl \leftarrow 1$; `reset` (A) `/* set initial values */`
`InitialPropagation<<<b,t>>>`$(A, \Delta, Viol)$ `/* check input units sat */`
if $Viol$ **then return no-answer-set**
else loop
  `PropagateAndCheck`$(A, \Delta, cdl, Viol)$ `/* update` A `and flag` $Viol$ `*/`
  if $Viol \wedge (cdl = 1)$ **then return no-answer-set** `/* Violation at first dec.level */`
  else if $Viol$ **then** `/* Violation at level` $cdl{>}1$ `*/`
    `Learning<<<b,t>>>`(Δ, A, cdl) `/* conflict analysis: update` Δ, cdl `*/`
    `Backjump<<<b,t>>>`(A, cdl) `/* update` A `and` cdl `*/`
  if *(A is not total)* **then**
    `Decision<<<b,t>>>`(Δ, A, Lit) `/* If possible, rank/select/extend` A `*/`
    if *no-selection* **then** `/* no applicable rules */`
      `CompleteAssignment<<<b,t>>>`(A) `/* falsify unassigned atoms */`
  else return $A^T \cap atom(P)$ `/* stable model found */`

Implementation details: Data representation

- literals are represented by (signed) integers
- assignments are represented by arrays of signed integers (dl+sign)
- literals of each nogood are stored contiguously and nogoods are stored in contiguous locations using a CSR-like representation.
  For example, the nogoods $\{1, 2, 3, 4\}$, $\{3, 4, 7\}$, $\{2, 7, 11\}$,... are stored as:



  This ensures coalescence when entire nogood has to be retrieved

Moreover

- contiguous nogoods/literals are processed by contiguous threads
- nogoods are sorted and partitioned w.r.t. their size
  This improves uniformity of workload for threads of the same block

(see the paper for more technical details)

Implementation details: Propagate-and-Check

The propagation and the check-for-violation tasks are performed by the same code. In particular,

- 1-to-1 mapping between nogoods and threads is adopted
- a standard technique based on *watched literals* is used
- nogoods of different length are processed by different grids (uniform workload)
- special kernels designed for unary, binary, and ternary nogoods
- check/propagate processes only nogoods affected by the last propagation step
- learned nogoods are processes similarly (different data structure)

Implementation details: Selection and Learning

The selection/decision procedure is designed to implement ASP-computations

- 1-to-1 mapping between rules and threads to detect applicable rules
- 1-to-1 mapping between rules and threads to rank applicable rules
- logarithmic reduction to determine best choices (shared mem. and shuffling)

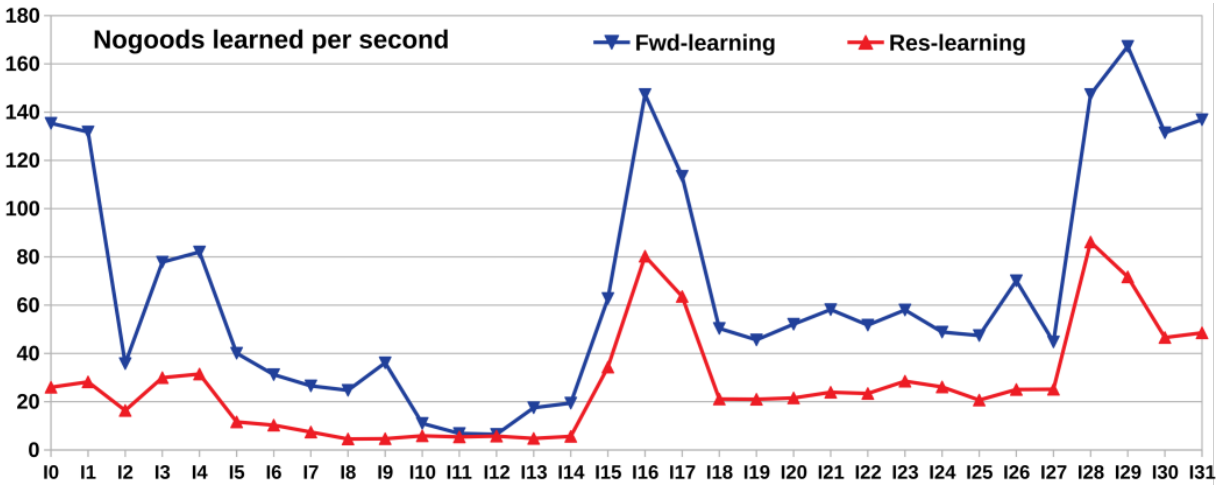
Two alternative learning procedures:

RES: parallel version of `clasp`-like resolution-based

FWD: learning schema based on dependencies

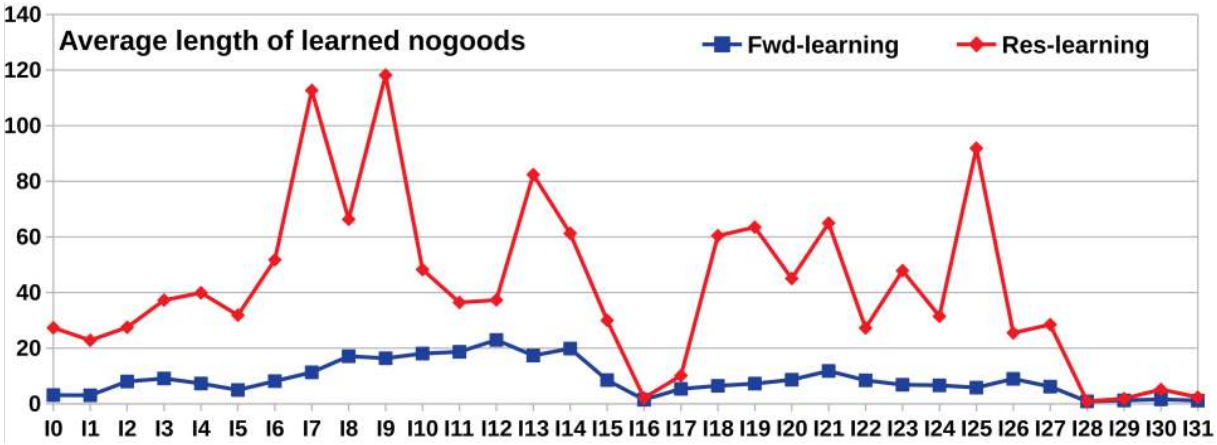
Experimental comparison of the two learning procedures

FWD-learning is faster:

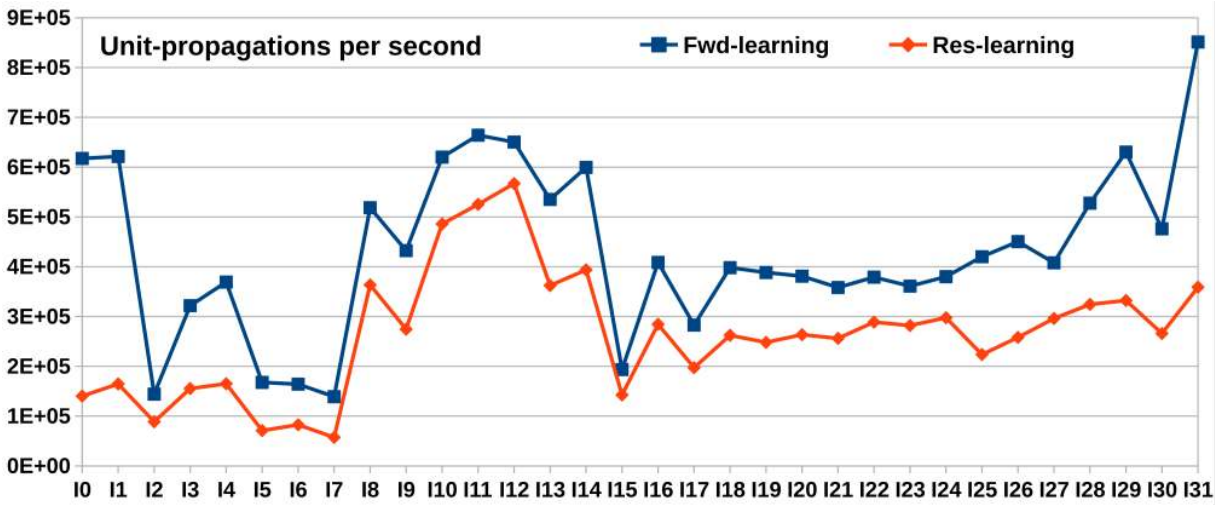


RES-learning vs FWD-learning

...and generates shorter nogoods:

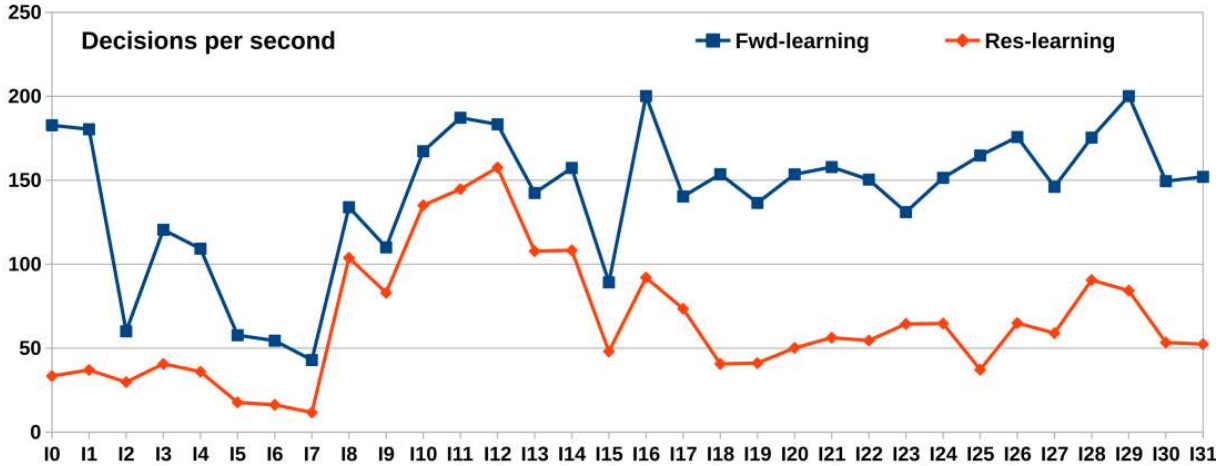


Consequently, FWD-learning speeds-up the propagation procedure



RES-learning vs FWD-learning

...and the entire search for stable models:



Conclusions

- We investigated the parallelization of ASP-solving on GPUs
- The design and implementation of an ASP-solver exploiting GPUs is possible but challenging, because of
  - the specific character of the satisfiability problem under stable-model semantics (similarly, with SAT solving)
  - the particular characteristics of the HW and the constraints imposed by the execution model
- Performance of the implemented prototype scales with the computing power of the GPUs
- but more has to be done:
  - we focused only on basic components of the solver
  - missing: all smart heuristics used in state-of-the-art solvers
  - missing: multi-level parallelism (see next)

More parallelism?

Themes for future/ongoing work: Multi-level parallelism:

- program splitting (e.g., relying on the splitting theorem) and process sub-programs in parallel
- space-search splitting (lookahead techniques)
- compute multiple solutions in parallel
- partitioned solving exploiting multiple devices and heterogeneous architectures