Turbo: Design of a Constraint Programming Solver on GPU

Pierre Talbot

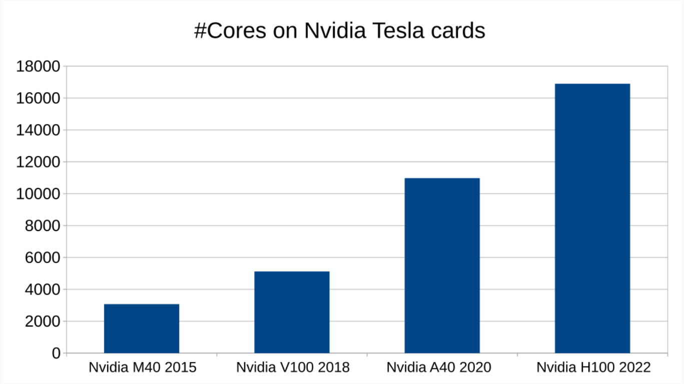
`pierre.talbot@uni.lu`

9th January 2024

University of Luxembourg

Why CP on GPU?

CPU clock speed is stagnating, GPU #cores is increasing quickly each year.



Easy speed-up: same code but faster.

Why CP on GPU?

CPU clock speed is stagnating, GPU #cores is increasing quickly each year

"The biggest lesson that can be read from 70 years of AI research is that general methods that leverage computation are ultimately the most effective, and by a large margin." [a]

[a] The Bitter Lesson, Rich Sutton, 2019, `http://www.incompleteideas.net/IncIdeas/BitterLesson.html`

Easy speed up: same code but faster

Why CP on GPU?

"Leading LLM researchers are already espousing Rich Sutton's bitter lesson; the fact that almost no innovation is required beyond scale."[a]

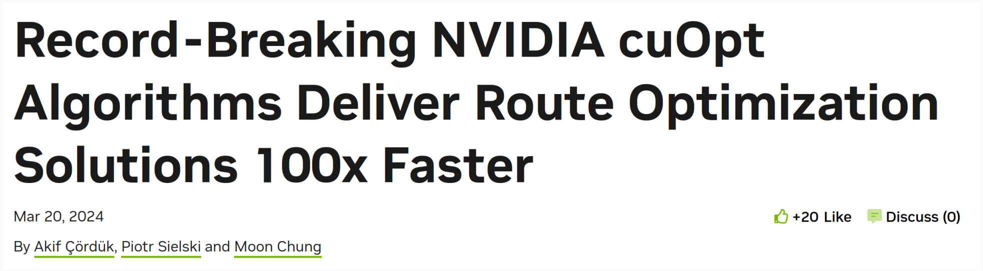
[a] https://medium.com/@felixhill/200bn-weights-of-responsibility-da85a44a2c5e

Why CP on GPU?

- Machine learning (deep learning, reinforcement learning, . . .) has seen tremendous speed-ups (e.g. 100x, 1000x) by using GPU.
- Some (sequential) optimizations on CPU are made irrelevant if we can explore huge state space faster.

Can we replicate the success of machine learning applications to combinatorial optimization?

cuOpt: Nvidia Routing Optimization



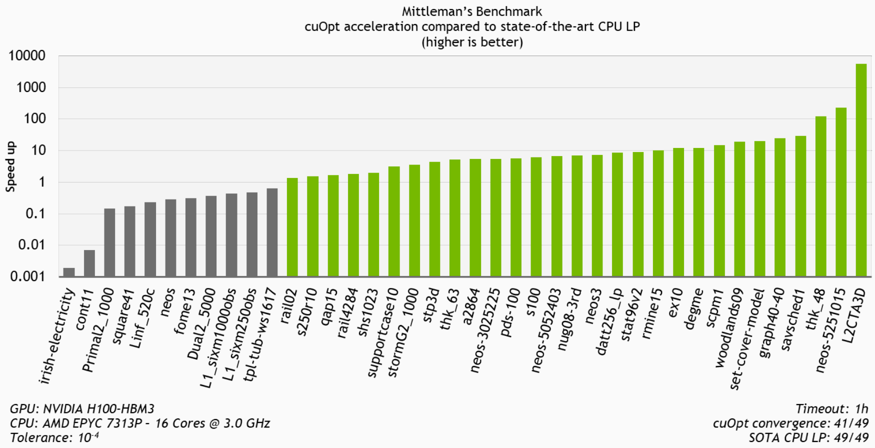
Record-Breaking NVIDIA cuOpt Algorithms Deliver Route Optimization Solutions 100x Faster

Mar 20, 2024

By Akif Çördük, Piotr Sielski and Moon Chung

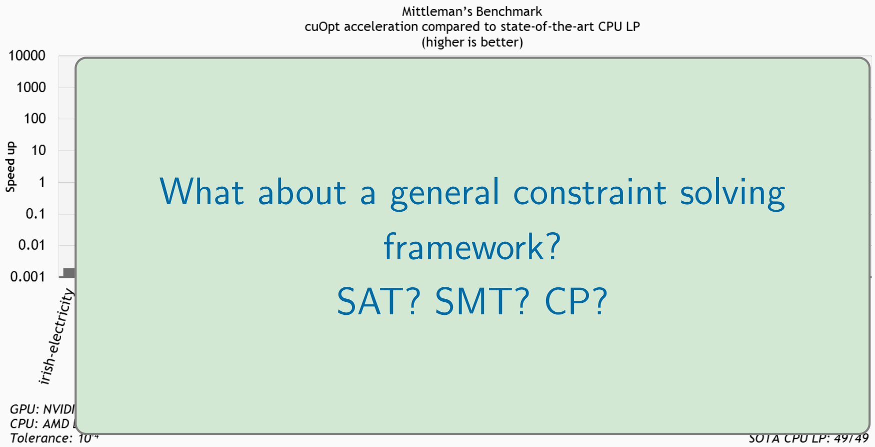
Cool, but specialized combinatorial algorithm for routing.

cuOpt: Nvidia Linear Programming Solver



Better! But only for linear constraints over continuous domain.

cuOpt: Nvidia Linear Programming Solver



What about a general constraint solving framework?
SAT? SMT? CP?

Better! But only for linear constraints over continuous domain.

Combinatorial Optimization on GPU

Very scarce literature, usually:

- **Heuristics**: often population-based algorithms[1].
- **Limited set of problems**[2]
- **Limited GPU parallelization**: offloading to GPU specialized filtering procedures[3,4].
- **Limited expressivity**: solver with max 256 set variables[5].

For CP-based approach: no code and no proof of correctness.

[1] A. Arbelaez and P. Codognet, *A GPU Implementation of Parallel Constraint-Based Local Search*, PDP, 2014.
[2] Jan Gmys. Exactly Solving Hard Permutation Flowshop Scheduling Problems on Peta-Scale GPU-Accelerated Supercomputers. INFORMS Journal on Computing, 2022.
[3] F. Campeotto et al., *Exploring the use of GPUs in constraint solving*, PADL, 2014
[4] F. Tardivo et al., *Constraint propagation on GPU: A case study for the AllDifferent constraint*, Journal of Logic and Computation, 2023.
[5] A. Dovier et al., *CUDA: Set Constraints on GPUs*, 2022.

Our Contributions

Theoretical Foundation

We propose a new parallel model of computation:

- **Correct**: formal proofs of correctness w.r.t. the parallel load/store operations on memory[6].
- **Simple and lock-free**: no mutex, no complicated atomic primitives, just barriers.
- **Expressive**: a process algebra based on *concurrent constraint programming* which supports $\mathbb{Z}, \mathbb{R}, \mathcal{P}(\mathbb{Z}), \ldots$ domains.

[6] P. Talbot et al., *A Variant of Concurrent Constraint Programming on GPU*, AAAI, 2022.

Our Contributions

Theoretical Foundation

We propose a new parallel model of computation:

- **Correct**: formal proofs of correctness w.r.t. the parallel load/store operations on memory[6].
- **Simple and lock-free**: no mutex, no complicated atomic primitives, just barriers.
- **Expressive**: a process algebra based on *concurrent constraint programming* which supports $\mathbb{Z}, \mathbb{R}, \mathcal{P}(\mathbb{Z}), \ldots$ domains.

Implementation

TURBO: a constraint solver fully executing on GPU.

- **General**: Support MiniZinc and XCSP3 constraint models (currently only $\mathbb{Z}$ variables).
- **Open-source**: Publicly available on `https://github.com/ptal/turbo`.
- **Efficient?**: In progress! $\times 10$ in 1 year (`https://lattice-land.github.io/`).

[6] P. Talbot et al., *A Variant of Concurrent Constraint Programming on GPU*, AAAI, 2022.

What is in this presentation?

On the menu:

1. Constraint Programming (CP)
2. Graphical Processing Units (GPU)
3. Towards a **correct** GPU constraint solver.
4. Towards an **efficient** GPU constraint solver.

Constraint Programming

Constraint Programming

Constraint satisfaction problem (CSP)

A CSP is a pair $\langle d, C\rangle$, example:

$$\langle\{x \mapsto [1,10], y \mapsto [1,10], z \mapsto [1,10]\}, \{x+3 \leq y, x+6 \leq z\}\rangle$$

A solution is $\{x \mapsto 1, y \mapsto 4, z \mapsto 9\}$.

Notation: given $d(x) = [a, b]$, we write $d(x)_\ell = a$ and $d(x)_u = b$.

Propagators

Propagator for the constraint $x + k \leq y$ with x, y variables and $k \in \mathbb{Z}$ a constant:

$$\begin{aligned}
&\mathit{propagate}(d, x + k \leq y) \triangleq \\
&\qquad \text{update } d(x)_u \text{ to } d(y)_u - k \\
&\qquad \text{update } d(y)_\ell \text{ to } d(x)_\ell + k
\end{aligned}$$

Definition

A propagator is a *reductive* ($f(x) \leq x$) and *monotone* ($x \leq y \Rightarrow f(x) \leq f(y)$) function over d.

Example

Memory:

$x = [1, 10]$
$y = [1, 10]$
$z = [1, 10]$

Propagators:

$propagate(d, x + 3 \leq y) =$
update $d(x)_u$ to $d(y)_u - 3$
update $d(y)_\ell$ to $d(x)_\ell + 3$

$propagate(d, x + 6 \leq z) =$
update $d(x)_u$ to $d(z)_u - 6$
update $d(z)_\ell$ to $d(x)_\ell + 6$

Example

Memory:

$x = [1, 10]$
$y = [1, 10]$
$z = [1, 10]$

Propagators:

$propagate(d, x + 3 \leq y) =$

update $d(x)_u$ to $d(y)_u - 3$

update $d(y)_\ell$ to $d(x)_\ell + 3$

$propagate(d, x + 6 \leq z) =$

update $d(x)_u$ to $d(z)_u - 6$

update $d(z)_\ell$ to $d(x)_\ell + 6$

Example

Memory:

$x = [1, 7]$
$y = [1, 10]$
$z = [1, 10]$

Propagators:

$propagate(d, x + 3 \leq y) =$
    update $d(x)_u$ to $d(y)_u - 3$
    update $d(y)_\ell$ to $d(x)_\ell + 3$

$propagate(d, x + 6 \leq z) =$
    update $d(x)_u$ to $d(z)_u - 6$
    update $d(z)_\ell$ to $d(x)_\ell + 6$

Example

Propagators:

Memory:

$x = [1, 7]$
$y = [4, 10]$
$z = [1, 10]$

$propagate(d, x + 3 \leq y) =$

update $d(x)_u$ to $d(y)_u - 3$

update $d(y)_\ell$ to $d(x)_\ell + 3$

$propagate(d, x + 6 \leq z) =$

update $d(x)_u$ to $d(z)_u - 6$

update $d(z)_\ell$ to $d(x)_\ell + 6$

Example

Memory:

$x = [1, 4]$
$y = [4, 10]$
$z = [1, 10]$

Propagators:

$propagate(d, x + 3 \leq y) =$
  update $d(x)_u$ to $d(y)_u - 3$
  update $d(y)_\ell$ to $d(x)_\ell + 3$

$propagate(d, x + 6 \leq z) =$
  update $d(x)_u$ to $d(z)_u - 6$
  update $d(z)_\ell$ to $d(x)_\ell + 6$

Example

Memory:

$x = [1, 4]$
$y = [4, 10]$
$z = [7, 10]$

Propagators:

$propagate(d, x + 3 \leq y) =$

update $d(x)_u$ to $d(y)_u - 3$

update $d(y)_\ell$ to $d(x)_\ell + 3$

$propagate(d, x + 6 \leq z) =$

update $d(x)_u$ to $d(z)_u - 6$

update $d(z)_\ell$ to $d(x)_\ell + 6$

Example

Memory:

$x = [1, 4]$
$y = [4, 10]$
$z = [7, 10]$

Propagators:

$propagate(d, x + 3 \leq y) =$
update $d(x)_u$ to $d(y)_u - 3$
update $d(y)_\ell$ to $d(x)_\ell + 3$

$propagate(d, x + 6 \leq z) =$
update $d(x)_u$ to $d(z)_u - 6$
update $d(z)_\ell$ to $d(x)_\ell + 6$

Example

Memory:

$x = [1, 4]$
$y = [4, 10]$
$z = [7, 10]$

Propagators:

$propagate(d, x + 3 \leq y) =$

update $d(x)_u$ to $d(y)_u - 3$

update $d(y)_\ell$ to $d(x)_\ell + 3$

$propagate(d, x + 6 \leq z) =$

update $d(x)_u$ to $d(z)_u - 6$

update $d(z)_\ell$ to $d(x)_\ell + 6$

Example

Memory:

$x = [1, 4]$
$y = [4, 10]$
$z = [7, 10]$

Propagators:

$propagate(d, x + 3 \leq y) =$

update $d(x)_u$ to $d(y)_u - 3$

update $d(y)_\ell$ to $d(x)_\ell + 3$

$propagate(d, x + 6 \leq z) =$

update $d(x)_u$ to $d(z)_u - 6$

update $d(z)_\ell$ to $d(x)_\ell + 6$

Fixpoint

Memory:

$x = [1, 4]$
$y = [4, 10]$
$z = [7, 10]$

Propagators:

$propagate(d, x + 3 \leq y) =$

update $d(x)_u$ to $d(y)_u - 3$

update $d(y)_\ell$ to $d(x)_\ell + 3$

$propagate(d, x + 6 \leq z) =$

update $d(x)_u$ to $d(z)_u - 6$

update $d(z)_\ell$ to $d(x)_\ell + 6$

Fixpoint

- There is no propagator that can update the domains!
- We have reached a **fixpoint**: $f(x) = x$.

But... the fixpoint of propagation is not necessarily a solution

Memory:

$x = [1, 4]$
$y = [4, 10]$
$z = [7, 10]$

Propagators:

$propagate(d, x + 3 \leq y) =$
update $d(x)_u$ to $d(y)_u - 3$
update $d(y)_\ell$ to $d(x)_\ell + 3$

$propagate(d, x + 6 \leq z) =$
update $d(x)_u$ to $d(z)_u - 6$
update $d(z)_\ell$ to $d(x)_\ell + 6$

But... the fixpoint of propagation is not necessarily a solution

Memory:

$x = [1, 4]$
$y = [4, 10]$
$z = [7, 10]$

The result is not a solution yet!

$$\{x \mapsto 4, y \mapsto 4, z \mapsto 7\}$$

$$x + 3 \leq y$$

Initial problem

$x = [1, 4]$
$y = [4, 10]$
$z = [7, 10]$

Subproblem 1 $(x = 1)$

$x = [1, 1]$
$y = [4, 10]$
$z = [7, 10]$

Solution reached!

$\{x \mapsto [1, 1], y \mapsto [4, 10], z \mapsto [7, 10]\}$

Subproblem 2 $(x > 1)$

$x = [2, 4]$
$y = [4, 10]$
$z = [7, 10]$

Not a solution, need to be further split into more subproblems.

A constraint solving algorithm: propagate and search

Let $\langle d, \{c_1, \ldots, c_n\}\rangle$ be a CSP.

- **Propagate**: Remove inconsistent values from the variables' domain: $\mathit{propagate}(d, \{c_1, \ldots, c_n\}) \triangleq \mathbf{gfp}\ ((\lambda x.x \sqcap d) \circ p_1 \circ \ldots \circ p_n)$ with $p_i \triangleq \mathit{propagate}(d, c_i)$.
- **Search**: Divide the problem into (complementary) subproblems explored using *backtracking*.

```
solve($\langle d, C\rangle$) =
    $d' \leftarrow$ propagate($\langle d, C\rangle$)
    if $d'$ is a solution then
        return $\{d'\}$
    else if $d'$ has an empty domain then
        return $\{\}$
    else
        $\langle d_1, \ldots, d_n\rangle \leftarrow$ branch($d'$)
        return $\bigcup_{i=0}^{n}$ solve($\langle d_i, C\rangle$)
    end if
```

A constraint solving algorithm: propagate and search

Let $\langle d, \{c_1, \ldots, c_n\}\rangle$ be a CSP.

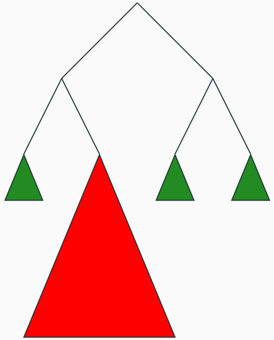
- **Propagate**: Remove inconsistent values from the variables' domain: $\mathit{propagate}(d, \{c_1, \ldots, c_n\}) \triangleq \mathbf{gfp}\ ((\lambda x.x \sqcap d) \circ p_1 \circ \ldots \circ p_n)$ with $p_i \triangleq \mathit{propagate}(d, c_i)$.
- **Search**: Divide the problem into (complementary) subproblems explored using *backtracking*.

```
$\texttt{solve}(\langle d, C\rangle) =$
  $d' \leftarrow \texttt{propagate}(\langle d, C\rangle)$
  **if** $d'$ is a solution **then**
    **return** $\{d'\}$
  **else if** $d'$ has an empty domain **then**
    **return** $\{\}$
  **else**
    $\langle d_1, \ldots, d_n\rangle \leftarrow \texttt{branch}(d')$
    **return** $\bigcup_{i=0}^{n} \texttt{solve}(\langle d_i, C\rangle)$
  **end if**
```

How to parallelize propagate and search?

Parallel CP (on CPU)

Parallel Search Made Simple[7]

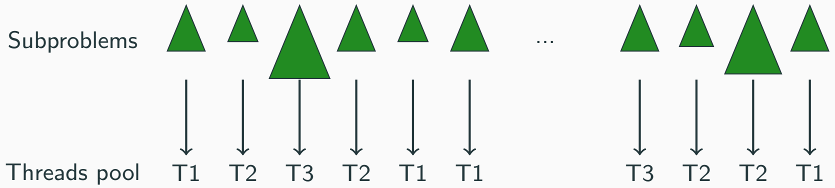


Unbalanced tree: use work-stealing but then threads need to communicate (complicated data structure, efficiency loss, ...).

[7] C. Schulte, ‘Parallel search made simple’, in Proceedings of TRICS, 2000

Embarrasingly Parallel Search (EPS)[8]

- **Idea**: Divide the problem into many subproblems beforehand (e.g. $N \times 30$ with N the number of threads).
- **Intuition**: Statistically, there is little chance a subproblem takes longer than the sum of the other subproblems.



[8]A. Malapert, J.-C. Régin, and M. Rezgui, 'Embarrassingly Parallel Search in Constraint Programming', JAIR, 2016

What About Parallel Propagation?

State of the art

- Parallelization of propagation has never worked well on CPU[9].
- It was not (and still not) very successful in Prolog (AND-Parallelism)[10].

$\Rightarrow$ There is no solver using parallel propagation on CPU!

WHY ?

[9] I. P. Gent et al., *A review of literature on parallel constraint solving*, Theory and Practice of Logic Programming, 2018.
[10] A. Dovier et al., *Parallel logic programming: A sequel*, Theory and Practice of Logic Programming, 2022.

What About Parallel Propagation?

State of the art

- Parallelization of propagation has never worked well on CPU[9].
- It was not (and still not) very successful in Prolog (AND-Parallelism)[10].

$\Rightarrow$ There is no solver using parallel propagation on CPU!

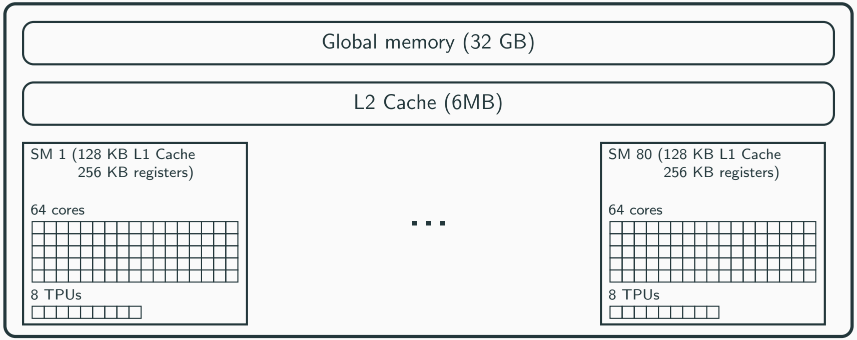
WHY ?

- OR-parallelism is easy: independent subproblems per core (**almost linear scaling**).
- AND-parallelism is hard: propagators works on the same data.

[9] I. P. Gent et al., *A review of literature on parallel constraint solving*, Theory and Practice of Logic Programming, 2018.
[10] A. Dovier et al., *Parallel logic programming: A sequel*, Theory and Practice of Logic Programming, 2022.

GPU Architecture

(Simplified) Architecture of the GPU Nvidia V100


Global memory (32 GB)
L2 Cache (6MB)
SM 1 (128 KB L1 Cache 256 KB registers)
64 cores
8 TPUs
...
SM 80 (128 KB L1 Cache 256 KB registers)
64 cores
8 TPUs

5120 cores on a single V100 GPU @ 1290MHz

640 Tensor Processing Units (TPUs)

Whitepaper: `https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf`

Programming Challenges

- **Memory coalescence**: the way to access the data is important (factor 10).
- **Thread divergence**: each thread within a warp (group of 32 threads) should execute the same instructions.
- **Memory allocation** (dynamic data structures): costly on GPU, everything is generally pre-allocated.
- **Other limitations**: small cache, limited number of lines of code, ...

Example: Find the Minimum in an Array

Each thread computes its local min (*map*), then we compute the min of all local min (*reduce*).

- Map:

$$\underbrace{\boxed{3}\boxed{22}\boxed{10}\boxed{23}}_{\text{Thread 0, } m_0 = 3}\underbrace{\boxed{21}\boxed{7}\boxed{91}\boxed{1}}_{\text{Thread 1, } m_1 = 1}\underbrace{\boxed{3}\boxed{10}\boxed{42}\boxed{11}}_{\text{Thread 2, } m_2 = 3}\underbrace{\boxed{8}\boxed{7}\boxed{32}}_{\text{Thread 3, } m_3 = 7}$$

- Reduce: $min([3, 1, 3, 7]) = 1$.

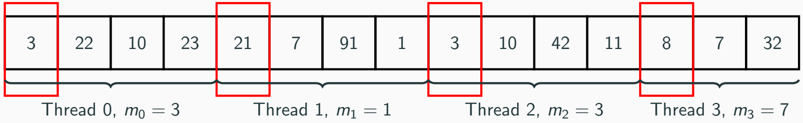
Example: Find the Minimum in an Array

Intuitive implementation

```
__global__ void parallel_min(int* v, size_t n, int* local_min) {
  local_min[threadIdx.x] = INT_MAX;
  size_t m = n / blockDim.x + (n % blockDim.x != 0);
  size_t from = threadIdx.x * m;
  size_t to = min(n, from + m);
  for(size_t i = from; i < to; ++i) {
    local_min[threadIdx.x] = min(local_min[threadIdx.x], v[i]);
  }
}
```

Iteration 1:

- Map:



- Reduce: $min([3, 1, 3, 7]) = 1$.

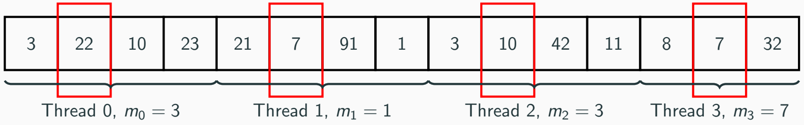
Example: Find the Minimum in an Array

Intuitive implementation

```
__global__ void parallel_min(int* v, size_t n, int* local_min) {
  local_min[threadIdx.x] = INT_MAX;
  size_t m = n / blockDim.x + (n % blockDim.x != 0);
  size_t from = threadIdx.x * m;
  size_t to = min(n, from + m);
  for(size_t i = from; i < to; ++i) {
    local_min[threadIdx.x] = min(local_min[threadIdx.x], v[i]);
  }
}
```

Iteration 2:

- Map:



- Reduce: $min([3, 1, 3, 7]) = 1$.

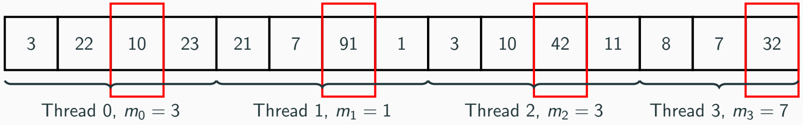
Example: Find the Minimum in an Array

Intuitive implementation

```
__global__ void parallel_min(int* v, size_t n, int* local_min) {
  local_min[threadIdx.x] = INT_MAX;
  size_t m = n / blockDim.x + (n % blockDim.x != 0);
  size_t from = threadIdx.x * m;
  size_t to = min(n, from + m);
  for(size_t i = from; i < to; ++i) {
    local_min[threadIdx.x] = min(local_min[threadIdx.x], v[i]);
  }
}
```

Iteration 3:

- Map:



- Reduce: $min([3, 1, 3, 7]) = 1$.

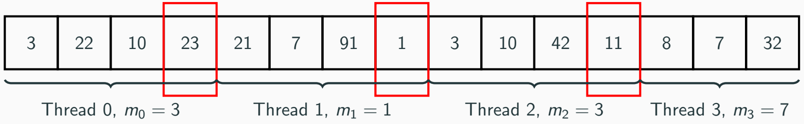
Example: Find the Minimum in an Array

Intuitive implementation

```
__global__ void parallel_min(int* v, size_t n, int* local_min) {
  local_min[threadIdx.x] = INT_MAX;
  size_t m = n / blockDim.x + (n % blockDim.x != 0);
  size_t from = threadIdx.x * m;
  size_t to = min(n, from + m);
  for(size_t i = from; i < to; ++i) {
    local_min[threadIdx.x] = min(local_min[threadIdx.x], v[i]);
  }
}
```

Iteration 4:

- Map:

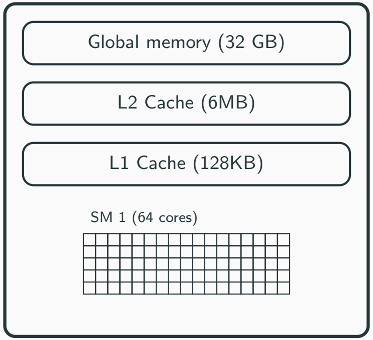


- Reduce: $min([3, 1, 3, 7]) = 1$.

Optimization: Coalesced Memory Accesses

Knowing about the hardware is crucial for efficiency.

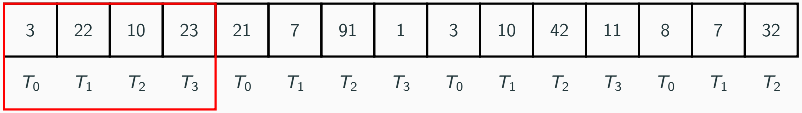
- Previous implementation can work well on CPU since each core has its own cache.
- On GPU, it is better to access the memory contiguously—it allows the GPU to move data from global memory to cache faster.



Optimization: Coalesced Memory Accesses

Strided implementation

```
__global__ void parallel_min_stride(int* v, size_t n, int* local_min) {
  local_min[threadIdx.x] = INT_MAX;
  for(size_t i = threadIdx.x; i < n; i += blockDim.x) {
    local_min[threadIdx.x] = min(local_min[threadIdx.x], v[i]);
  }
}
```



Very important: up to an order of magnitude faster (10x).

Optimization: Coalesced Memory Accesses

Strided implementation

```
__global__ void parallel_min_stride(int* v, size_t n, int* local_min) {
  local_min[threadIdx.x] = INT_MAX;
  for(size_t i = threadIdx.x; i < n; i += blockDim.x) {
    local_min[threadIdx.x] = min(local_min[threadIdx.x], v[i]);
  }
}
```

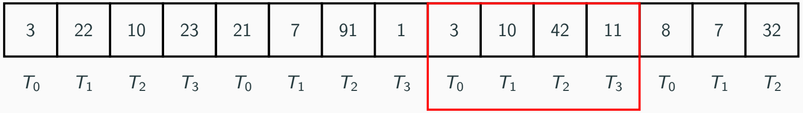
3	22	10	23	21	7	91	1	3	10	42	11	8	7	32
T_0	T_1	T_2	T_3	T_0	T_1	T_2	T_3	T_0	T_1	T_2	T_3	T_0	T_1	T_2

Very important: up to an order of magnitude faster (10x).

Optimization: Coalesced Memory Accesses

Strided implementation

```
__global__ void parallel_min_stride(int* v, size_t n, int* local_min) {
  local_min[threadIdx.x] = INT_MAX;
  for(size_t i = threadIdx.x; i < n; i += blockDim.x) {
    local_min[threadIdx.x] = min(local_min[threadIdx.x], v[i]);
  }
}
```


3 22 10 23 21 7 91 1 3 10 42 11 8 7 32
T_0 T_1 T_2 T_3 T_0 T_1 T_2 T_3 T_0 T_1 T_2 T_3 T_0 T_1 T_2

Very important: up to an order of magnitude faster (10x).

Optimization: Coalesced Memory Accesses

Strided implementation

```
__global__ void parallel_min_stride(int* v, size_t n, int* local_min) {
  local_min[threadIdx.x] = INT_MAX;
  for(size_t i = threadIdx.x; i < n; i += blockDim.x) {
    local_min[threadIdx.x] = min(local_min[threadIdx.x], v[i]);
  }
}
```

3	22	10	23	21	7	91	1	3	10	42	11	8	7	32
T_0	T_1	T_2	T_3	T_0	T_1	T_2	T_3	T_0	T_1	T_2	T_3	T_0	T_1	T_2

Very important: up to an order of magnitude faster (10x).

Towards a Correct GPU Constraint Solver

Architecture Overview

- **Each GPU block solves an independent subproblem**: similar to embarassingly parallel search on CPU.
- Why not 1 subproblem per GPU thread? $\Rightarrow$ not enough cache per thread.

How to parallelize the propagation inside a GPU block?

Parallel Concurrent Constraint Programming (PCCP)

History

Concurrent constraint programming (CCP) is a *process calculus* introduced in the eighties[11].
Two main operations: ask and tell (example on whiteboard).

But CCP lacked a proper connection to parallel architecture, and we worked on that by simplifying the language (no recursion)[12].

[11] V. A. Saraswat and M. Rinard, *Concurrent constraint programming* (POPL-89)
[12] P. Talbot et al., *A Variant of Concurrent Constraint Programming on GPU* (AAAI 2022)

Parallel Concurrent Constraint Programming (PCCP)

History

Concurrent constraint programming (CCP) is a *process calculus* introduced in the eighties[11].
Two main operations: ask and tell (example on whiteboard).

But CCP lacked a proper connection to parallel architecture, and we worked on that by simplifying the language (no recursion)[12].

Syntax of PCCP

Let $x, y, y_1, \dots, y_n \in Vars$ be variables, L a lattice, f a monotone function, and b a Boolean variable of type $\langle\{true, false\}, \Leftarrow\rangle$:

$$\begin{array}{lll} \langle P, Q\rangle ::= & \texttt{if}\ \ b\ \ \texttt{then}\ \ P & \textit{ask statement} \\ \mid & x \leftarrow f(y_1, \dots, y_n) & \textit{tell statement} \\ \mid & \exists x{:}\mathrm{L},\ P & \textit{local statement} \\ \mid & P \parallel Q & \textit{parallel composition} \end{array}$$

[11] V. A. Saraswat and M. Rinard, *Concurrent constraint programming* (POPL-89)
[12] P. Talbot et al., *A Variant of Concurrent Constraint Programming on GPU* (AAAI 2022)

Denotational Semantics

- A PCCP process is a reductive and monotone function over a Cartesian product $\mathit{Store} = L_1 \times \ldots \times L_n$ storing the values of all local variables.
- Since we do not have recursion, we know at compile-time the number of variables.
- Let *Proc* be the set of all PCCP processes.

Denotational Semantics

- A PCCP process is a reductive and monotone function over a Cartesian product $\mathit{Store} = L_1 \times \ldots \times L_n$ storing the values of all local variables.
- Since we do not have recursion, we know at compile-time the number of variables.
- Let *Proc* be the set of all PCCP processes.

Denotational Semantics

We define a function $\mathcal{D} : \mathit{Proc} \to (\mathit{Store} \to \mathit{Store})$:

$$\begin{aligned}
&\mathcal{D}(x \leftarrow f(y_1, .., y_n)) \triangleq \lambda s.s[x \mapsto s(x) \sqcap f(s(y_1), .., s(y_n))] \\
&\mathcal{D}(\texttt{if } b \texttt{ then } P) \triangleq \lambda s.(\!|\ s(b) \,?\, \mathcal{D}(P)(s) \,\text{:}\, s\ |\!) \\
&\mathcal{D}(P \parallel Q) \triangleq \mathcal{D}(P) \sqcap \mathcal{D}(Q)
\end{aligned}$$

Executing the program: **gfp** $\mathcal{D}(P)$.

Sequential Computation = Parallel Computation

We obtain the same result if we execute P in parallel or if we replace all parallel $\|$ by a sequential operator ; (a transformation we write $\mathtt{seq}\ P$) defined as follows:

$$\mathcal{D}(P\ ;\ Q) \triangleq \mathcal{D}(Q) \circ \mathcal{D}(P)$$

Let **fix** f be the set of fixpoints of a function f.

Theorem (Equivalence Between Sequential and Parallel Operators)

$\textbf{fix}\ \mathcal{D}(\mathtt{seq}\ P) = \textbf{fix}\ \mathcal{D}(P)$

Very weak requirements on the parallel hardware

The semantics make the following assumptions on the memory consistency model and cache coherency protocol:

(ATOM) Load and store instructions must be atomic.

(EC) The caches must eventually become coherent.

(OTA) Values cannot appear *out-of-thin-air*.

Parallel Propagation

Memory:

$x = [1..10]$
$y = [1..10]$
$z = [1..10]$

Propagators:

$propagate(d, x + 3 \leq y) =$
  update $d(x)_u$ to $d(y)_u - 3$
  update $d(y)_\ell$ to $d(x)_\ell + 3$

$propagate(d, x + 6 \leq z) =$
  update $d(x)_u$ to $d(z)_u - 6$
  update $d(z)_\ell$ to $d(x)_\ell + 6$

$$propagate(d, x + 3 \leq y) \,||\, propagate(d, x + 6 \leq z)$$

Issues of Parallel Propagation

Memory:

$x = [1..7]$
$y = [1..10]$
$z = [1..10]$

Propagators:

$$\begin{array}{ll} & \text{update } d(x)_u \text{ to } d(y)_u - 3 \\ \| & \text{update } d(y)_\ell \text{ to } d(x)_\ell + 3 \\ \| & \text{update } d(x)_u \text{ to } d(z)_u - 6 \\ \| & \text{update } d(z)_\ell \text{ to } d(x)_\ell + 6 \end{array}$$

Issue 1: data race? Parallel update of the same integer $d(x)_u$.

Issues of Parallel Propagation

Memory:

$x = [1..7]$
$y = [1..10]$
$z = [1..10]$

Propagators:

$$\begin{array}{ll} & \text{update } d(x)_u \text{ to } d(y)_u - 3 \\ \| & \text{update } d(y)_\ell \text{ to } d(x)_\ell + 3 \\ \| & \text{update } d(x)_u \text{ to } d(z)_u - 6 \\ \| & \text{update } d(z)_\ell \text{ to } d(x)_\ell + 6 \end{array}$$

Issue 1: data race? Parallel update of the same integer $d(x)_u$.

$\Rightarrow$ **Solution**: Use atomic load and store!

Issues of Parallel Propagation

Memory:

$x = [1..7]$
$y = [1..10]$
$z = [1..10]$

Propagators:

$$\begin{array}{ll} & \text{update } d(x)_u \text{ to } d(y)_u - 3 \\ \| & \text{update } d(y)_\ell \text{ to } d(x)_\ell + 3 \\ \| & \text{update } d(x)_u \text{ to } d(z)_u - 6 \\ \| & \text{update } d(z)_\ell \text{ to } d(x)_\ell + 6 \end{array}$$

Issue 1: data race? Parallel update of the same integer $d(x)_u$.
$\Rightarrow$ **Solution**: Use atomic load and store!
$\Rightarrow$ **In CUDA**: Integer load and store are atomic by default!

Issues of Parallel Propagation

Memory:

$x = [1..7]$
$y = [1..10]$
$z = [1..10]$

Propagators:

$$
\begin{array}{ll}
 & \text{update } d(x)_u \text{ to } d(y)_u - 3 \\
\| & \text{update } d(y)_\ell \text{ to } d(x)_\ell + 3 \\
\| & \text{update } d(x)_u \text{ to } d(z)_u - 6 \\
\| & \text{update } d(z)_\ell \text{ to } d(x)_\ell + 6
\end{array}
$$

Issue 1: data race? Parallel update of the same integer $d(x)_u$.
$\Rightarrow$ **Solution**: Use atomic load and store!
$\Rightarrow$ **In CUDA**: Integer load and store are atomic by default!

Issue 2: nondeterminism? $d(x)_u$ can be equal to 4 or 7 depending on the order of execution.

Solution: Use a fixpoint loop

While something is changing we reexecute all the propagators!

Memory:

$x = [1..4]$
$y = [1..10]$
$z = [1..10]$

Propagators:

```
fp (
        update $d(x)_u$ to $d(y)_u - 3$
     || update $d(y)_\ell$ to $d(x)_\ell + 3$
     || update $d(x)_u$ to $d(z)_u - 6$
     || update $d(z)_\ell$ to $d(x)_\ell + 6$   )
```

Solution: Use a fixpoint loop

While something is changing we reexecute all the propagators!

Memory:

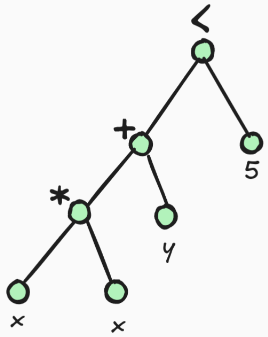
$x = [1..4]$
$y = [1..10]$
$z = [1..10]$

Need to add a condition for progress!

update $d(x)_\ell$ to v iff $v > d(x)_\ell$
update $d(x)_u$ to v iff $v < d(x)_u$

Towards an Efficient GPU Constraint Solver

Representation of Propagators



- Represented using `shared_ptr` and `variant` data structures.

  ⇒ Uncoalesced memory accesses.

- Code similar to an interpreter:

```
switch(term.index()) {
  case IVar:
  case INeg:
  case IAdd:
  case IMul:
  // ...
```

  ⇒ Thread divergence.

Ternary Normal Form (TNF)

We simplify the representation of constraints to ternary constraints of the form:

- `x = y <op> z` where `x,y,z` are variables.
- The operators are $\{+, -, /, *, \%, \mathit{min}, \mathit{max}, \leq, =\}$.

Example

The constraint $x + y \neq 2$ is represented by:

```
t1 = x + y
ZERO = (t1 = TWO)
```

where `ZERO` and `TWO` are two variables with constant values.

Bytecode Representation

The ternary form of a propagator holds on 16 bytes:

```
struct bytecode_type {
  Sig op;
  AVar x;
  AVar y;
  AVar z;
};
```

- **Uniform representation** of propagators in memory $\Rightarrow$ coalesced memory accesses.
- Limited number of operators + sorting $\Rightarrow$ reduced thread divergence.

Problems in Ternary Normal Form

Drawback 1 of TNF: increase in number of propagators and variables.

Problem	**#Variables**	**#Constraints**
team-assignment	35445 (x2.2)	45197 (x1.8)
generalized-peacable-queens	20186 (x6.9)	25519 (x3.1)
spot5	22053 (x19.8)	29065 (x3.6)
accap	2319 (x5.2)	2782 (x3)
wordpress	92695 (x139)	122921 (x4)

Observation: The increase in constraints is relatively stable across problems (between 1x and 6x), but not the increase in variables (between 1.5x and 139x).

Benchmarks

Average gain of TNF/bytecode representation instead of tree-based propagators (on 16 MiniZinc instances):

- **Fixpoint iterations per second**: +553% (845k)
- $\Rightarrow$ Huge gain: x5 more iterations.

Benchmarks

Average gain of TNF/bytecode representation instead of tree-based propagators (on 16 MiniZinc instances):

- **Fixpoint iterations per second**: +553% (845k)
- ⇒ Huge gain: x5 more iterations.
- **Nodes per second**: +18% (21808)
- ⇒ More modest increase in nodes per second. Why?

Benchmarks

Average gain of TNF/bytecode representation instead of tree-based propagators (on 16 MiniZinc instances):

- **Fixpoint iterations per second**: +553% (845k)
- $\Rightarrow$ Huge gain: x5 more iterations.
- **Nodes per second**: +18% (21808)
- $\Rightarrow$ More modest increase in nodes per second. Why?
- **Fixpoint iterations per node**: +138% (40 instead of 17 before).
- $\Rightarrow$ Slower convergence of the fixpoint loop.

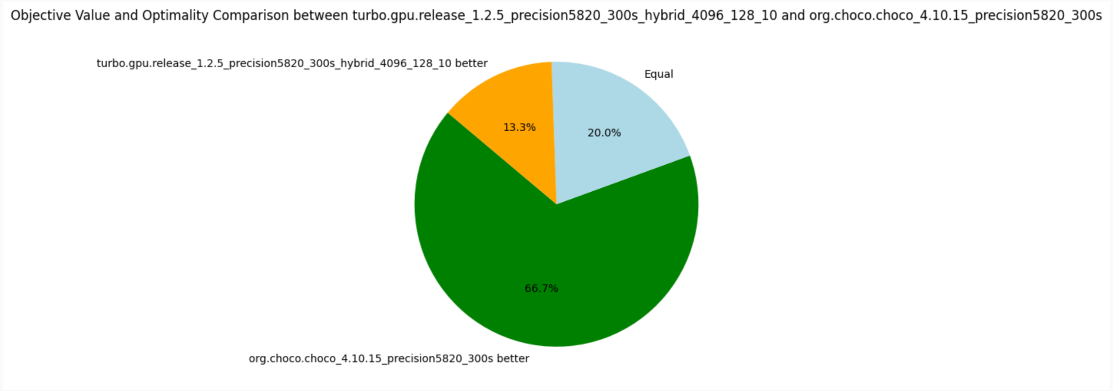
Benchmarks

Average gain of TNF/bytecode representation instead of tree-based propagators (on 16 MiniZinc instances):

- **Fixpoint iterations per second**: +553% (845k)
- ⇒ Huge gain: x5 more iterations.
- **Nodes per second**: +18% (21808)
- ⇒ More modest increase in nodes per second. Why?
- **Fixpoint iterations per node**: +138% (40 instead of 17 before).
- ⇒ Slower convergence of the fixpoint loop.
- **Memory footprint of propagators**: -90% (0.85MB instead of 7MB).
- **Memory footprint of variables**: +718% (402.75KB instead of 4.98KB).

Benchmarks: Turbo vs Choco v4.15

Comparison of the best objective values found.



What’s Next?

In-Progress: Parallel Consistency Algorithm

- **Goal**: Avoid executing all propagators in each iteration of the fixpoint loop.
- Constraint network as a sparse graph.
- Waking up propagators using GPU-based algorithms (SpMV and scan).

In-Progress: Table Abstract Domain

- **Context:** In constraint programming, *global constraints* are propagators with dedicated inference algorithms for subproblems, e.g., `alldifferent(`$[x_1,\ldots,x_n]$`)`.
- Research question: Which global constraints can be generalized into abstract domains?

Collaboration with Éric Monfroy

We are working on the *Table abstract domain* generalizing the well-known `table` constraint:

$$\begin{aligned} &(x \geq 4 \land y > 1 \land z < 3)\\ \lor&(x = 1 \land y = 2 \land z = 3)\\ \lor&(x > 1 \land y > 1 \land z > 3)\end{aligned}$$

Perspective: Towards automatic creation of the abstract domain

Research question: Given a set of abstract domains and reduced products, how to build the most efficient one to solve a given formula?

$$\begin{tikzcd}
 & \varphi \arrow[dl, "\llbracket\rrbracket"'] \arrow[dr, "\llbracket\rrbracket^\flat"] & \\
A_1^\sharp \times \ldots \times A_n^\sharp \arrow[rr, shift left, "\gamma"] & & C^\flat \arrow[ll, shift left, "\alpha"]
\end{tikzcd}$$

- How to create an appropriate combination of abstract domains for a particular formula?
- "Type inference": In which abstract domain goes each subformula $\varphi_i \in \varphi$?

Conclusion

C++ Abstraction: Lattice Land Project

`lattice-land` is a collection of libraries abstracting our parallel model.

It provides various data types and fixpoint engine:

- `ZLB`, `ZUB`: increasing/decreasing integers.
- `B`: Boolean lattices.
- `VStore`: Array (of lattice elements).
- `IPC`: Arithmetic constraints.
- `GaussSeidelIteration`: Sequential CPU fixed point loop.
- `AsynchronousIteration`: GPU-accelerated fixed point loop.
- ...

```
void max(int tid, const int* data, ZLB& m) {
  m.tell(data[tid]);
}
AsynchronousIteration::fixpoint(max);
```

https://github.com/lattice-land

Conclusion

Data races occur rarely, so we should avoid working so much to avoid them.

Properties of the model

A Variant of Concurrent Constraint Programming on GPU (AAAI 2022)[13].

- Correct: Proofs that $P; Q \equiv P||Q$, parallel and sequential versions produce the same results.
- Restartable: Stop the program at any time, and restart on partial data.
- Modular: Add more threads without fear of breaking existing code.
- Weak memory consistency: Very few requirements on the underlying memory model $\Rightarrow$ wide compatibility across hardware, unlock optimization.

[13] `https://ptal.github.io/papers/aaai2022.pdf`