Exploiting Unexploited Computing Resources for Computational Logics

Alessandro Dal Palù[1] Agostino Dovier[2]
Andrea Formisano[3] Enrico Pontelli[4]

1. Università di Parma

2. Università di Udine

3. Università di Perugia

4. New Mexico State University

ROMA, June 2012

Introduction

- Every new desktop/laptop comes equipped with a powerful graphic processor unit (GPU)

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)

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)

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 computational logics?

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 computational logics?
- We present here a preliminary investigation, focusing on SAT

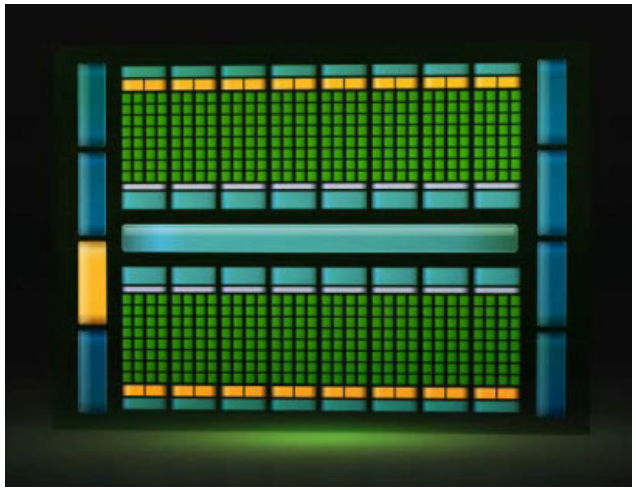
GPUs, in few minutes

GPUs, in few minutes

GPUs, in few minutes

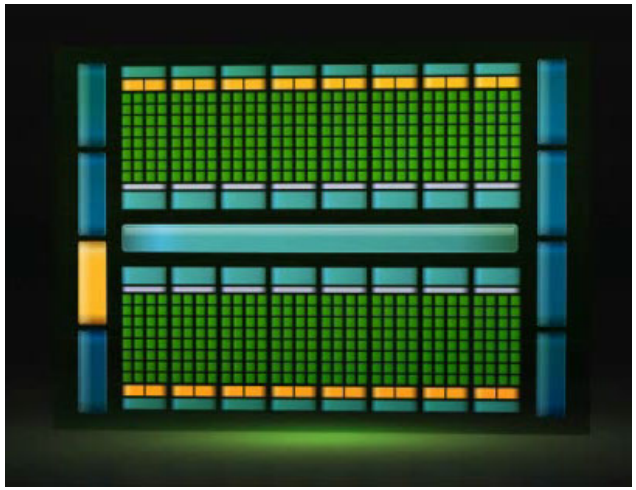
GPUs, in few minutes

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.

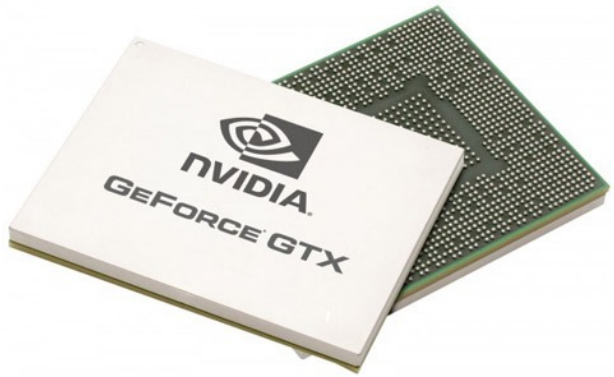


However, things are not that easy. Cores are organized hierarchically, memories has different behaviors, . . . it's not easy to obtain a good speed-up.

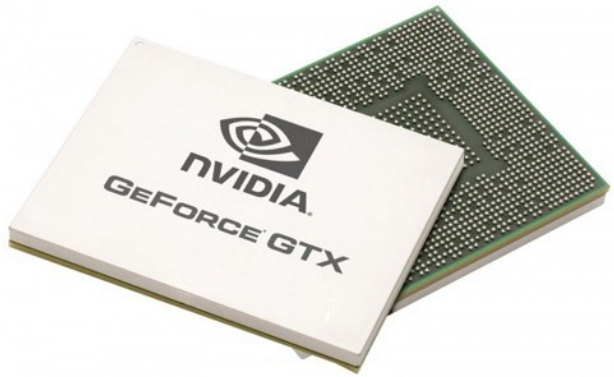
CUDA: Host, Global, Device

- A (C) CUDA program is a high-level program that can partially run on a GPU
- *host* = The computer CPU; *device* = the (general purpose) GPU
- Every function in a CUDA program is labelled as `host` (the default), `global`, or `device`.
- A `host` function runs in the host and it is a standard C function (e.g. `main`)
- A `global` function (or kernel) is called by a `host` function but it runs on the device.
- A `device` function can be called only by a function running on the device and it runs on the device.
- Limitations for device functions exist (and can be different for different GPUs)

CUDA: Host, Global, Device

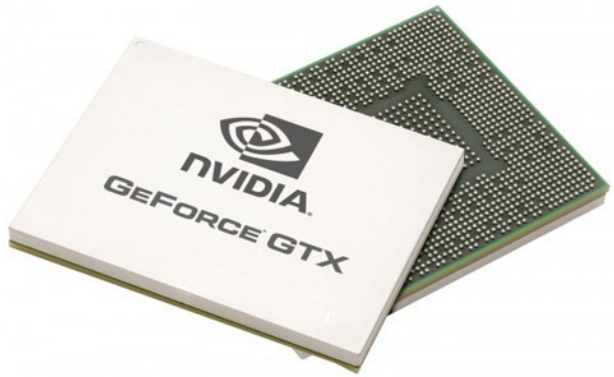


CUDA: Host, Global, Device



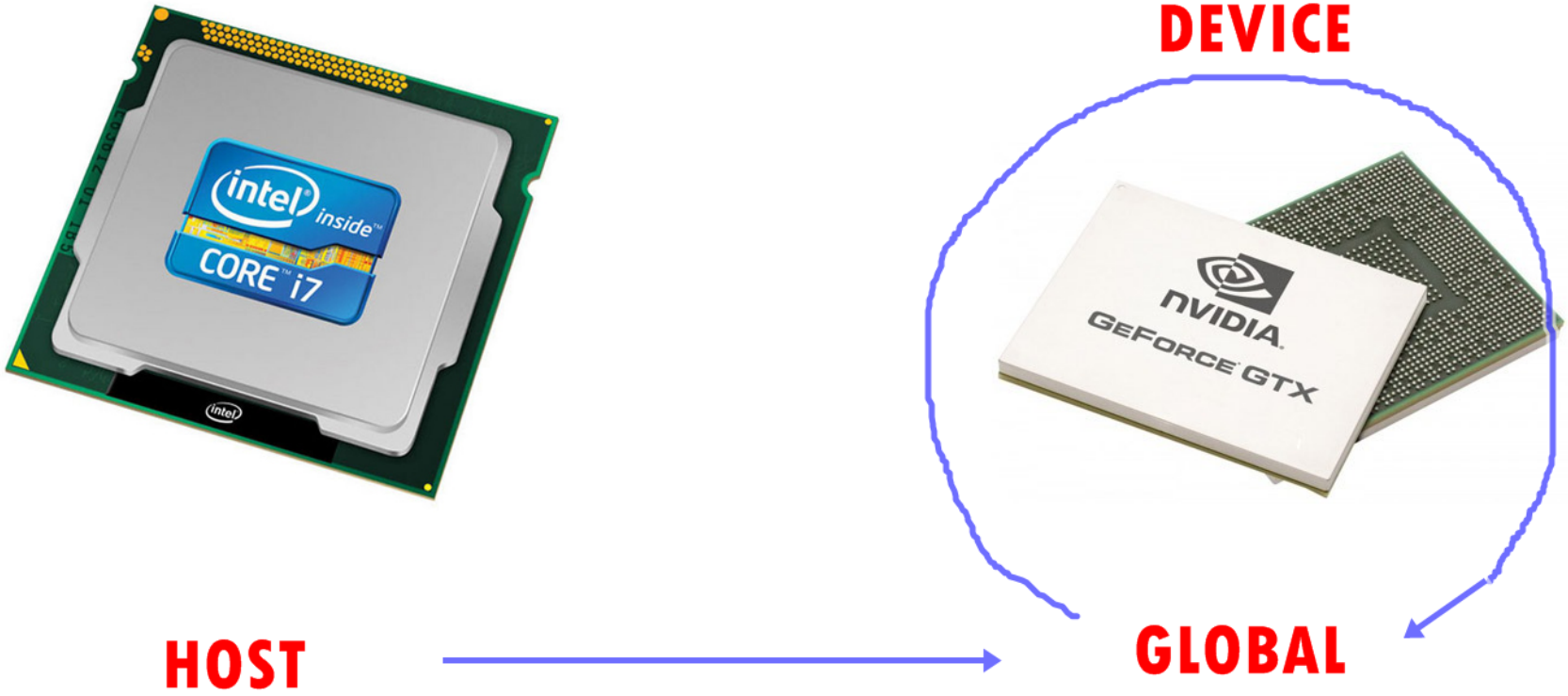
HOST

CUDA: Host, Global, Device

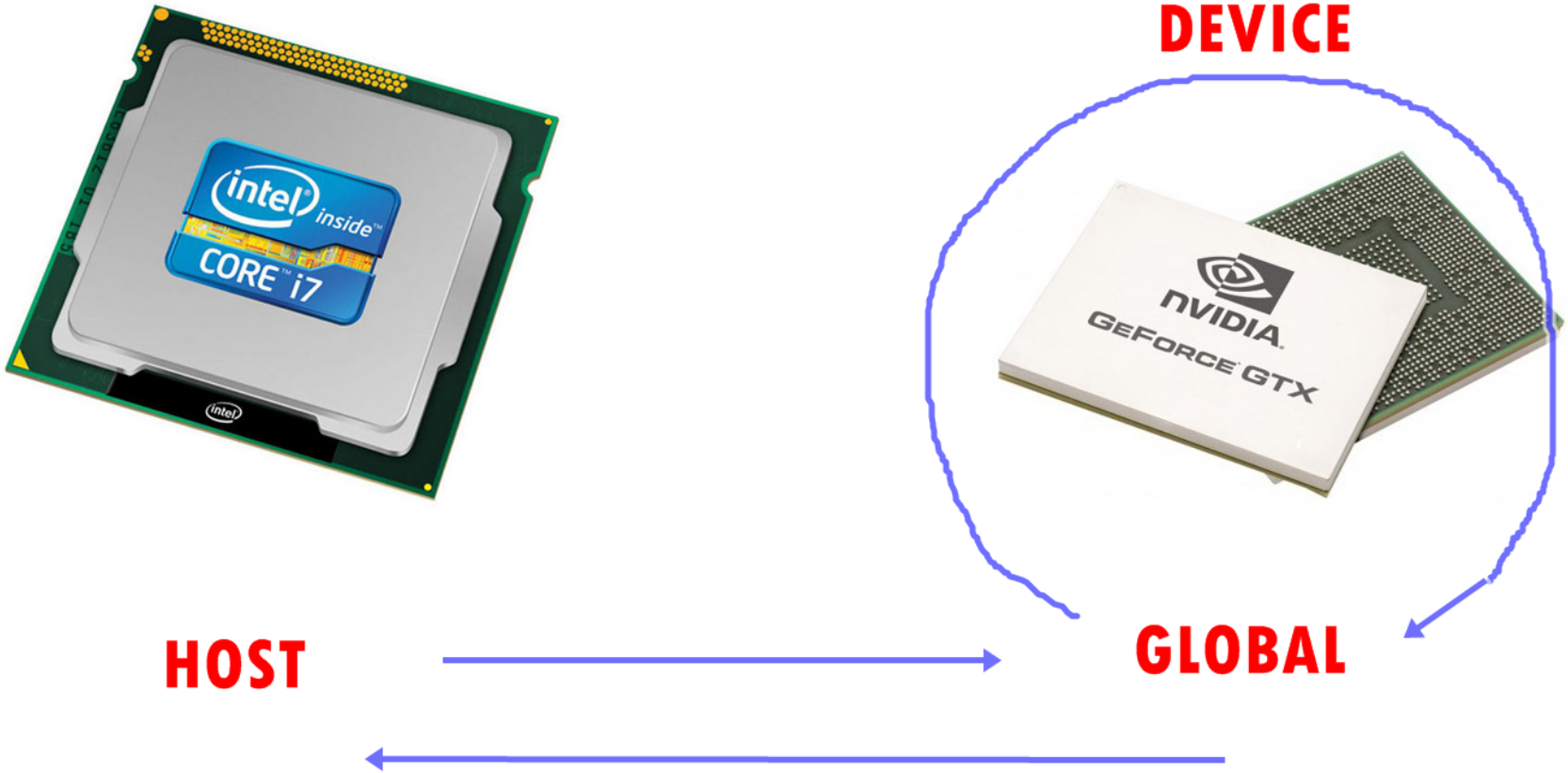


HOST → GLOBAL

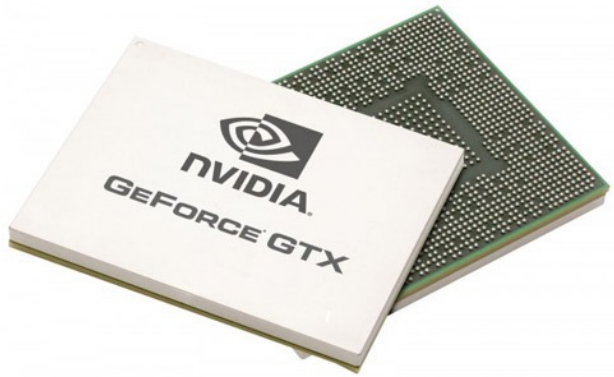
CUDA: Host, Global, Device



CUDA: Host, Global, Device



CUDA: Host, Global, Device

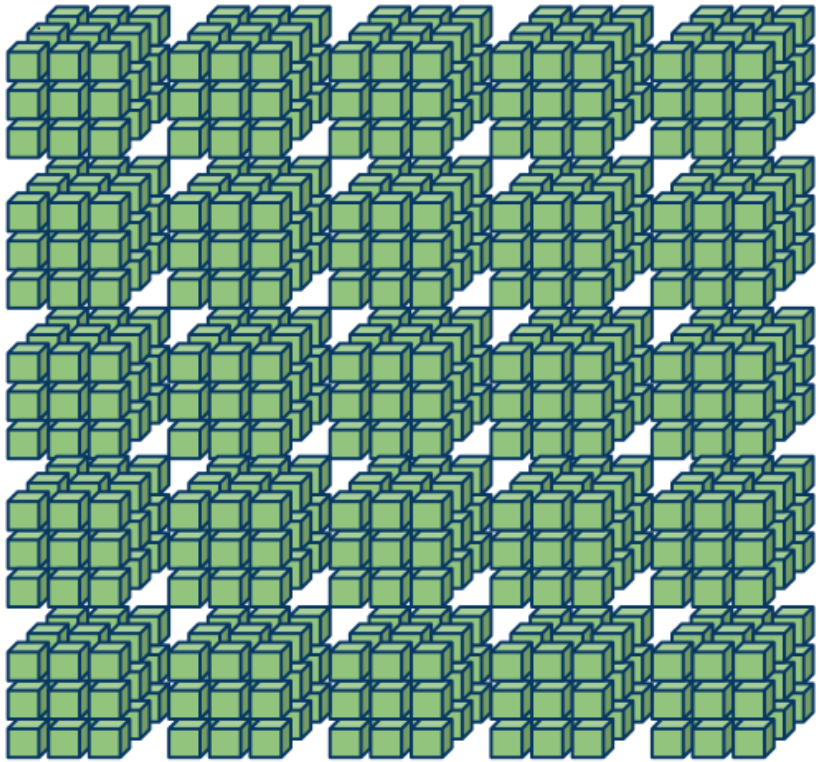


HOST

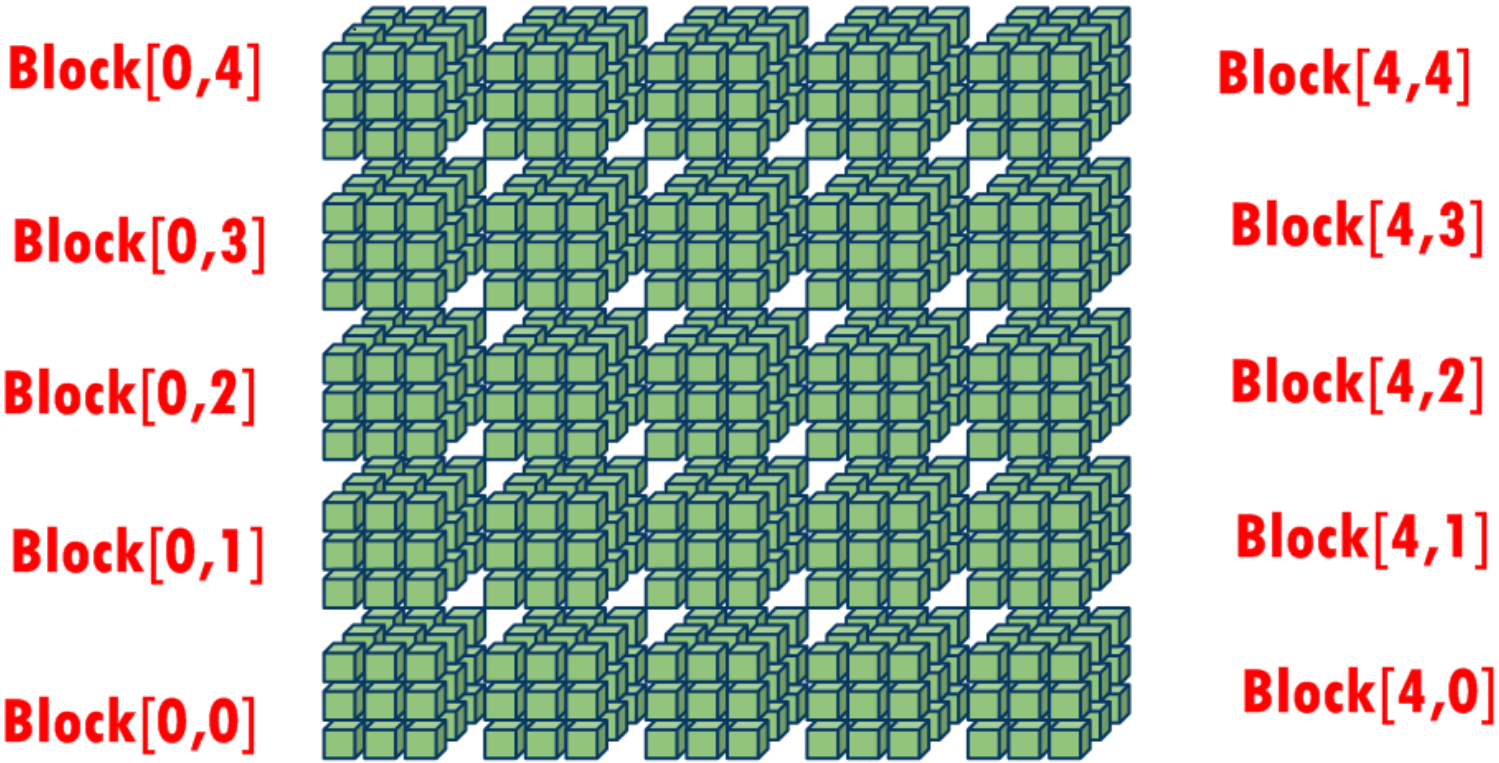
CUDA: Grids, Blocks, threads

- When a kernel function is called, the number of parallel executions is established
- The organization of the threads is hierarchical.
- 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.
- Each thread 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. It is important that there is SIMD (Single Instruction Multiple Data) in a warp to increase real parallelism.

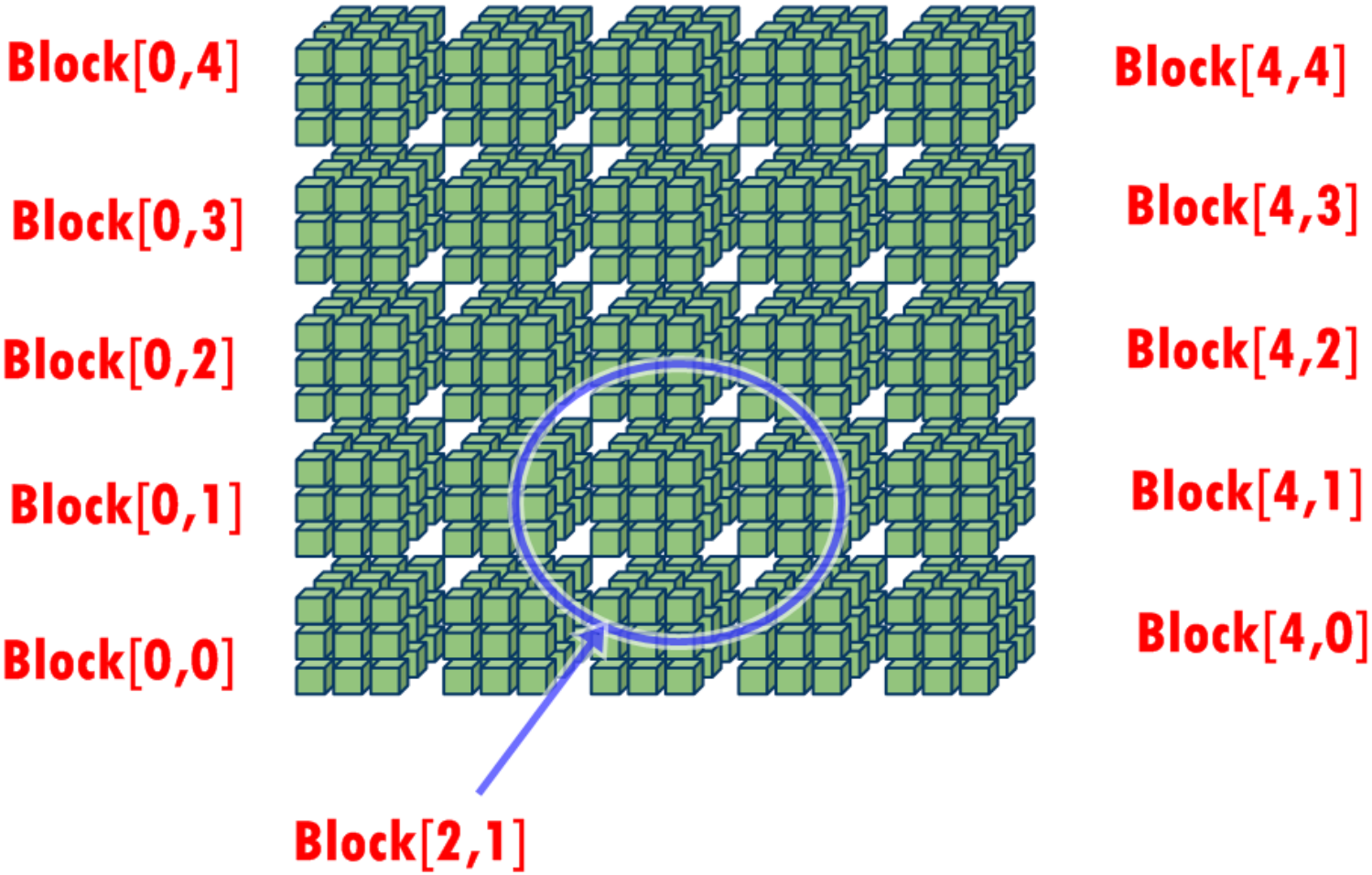
CUDA: Host, Global, Device



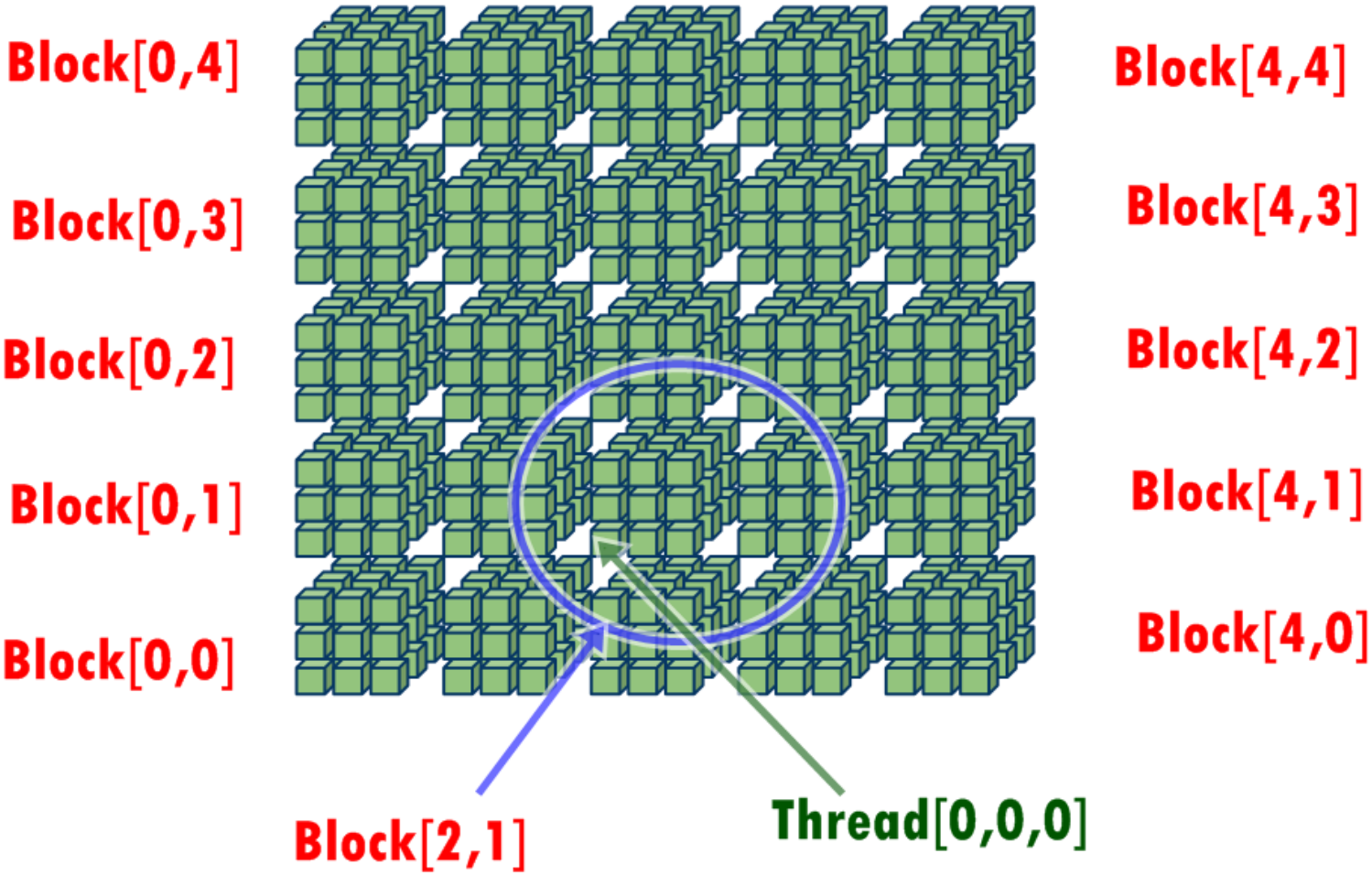
CUDA: Host, Global, Device



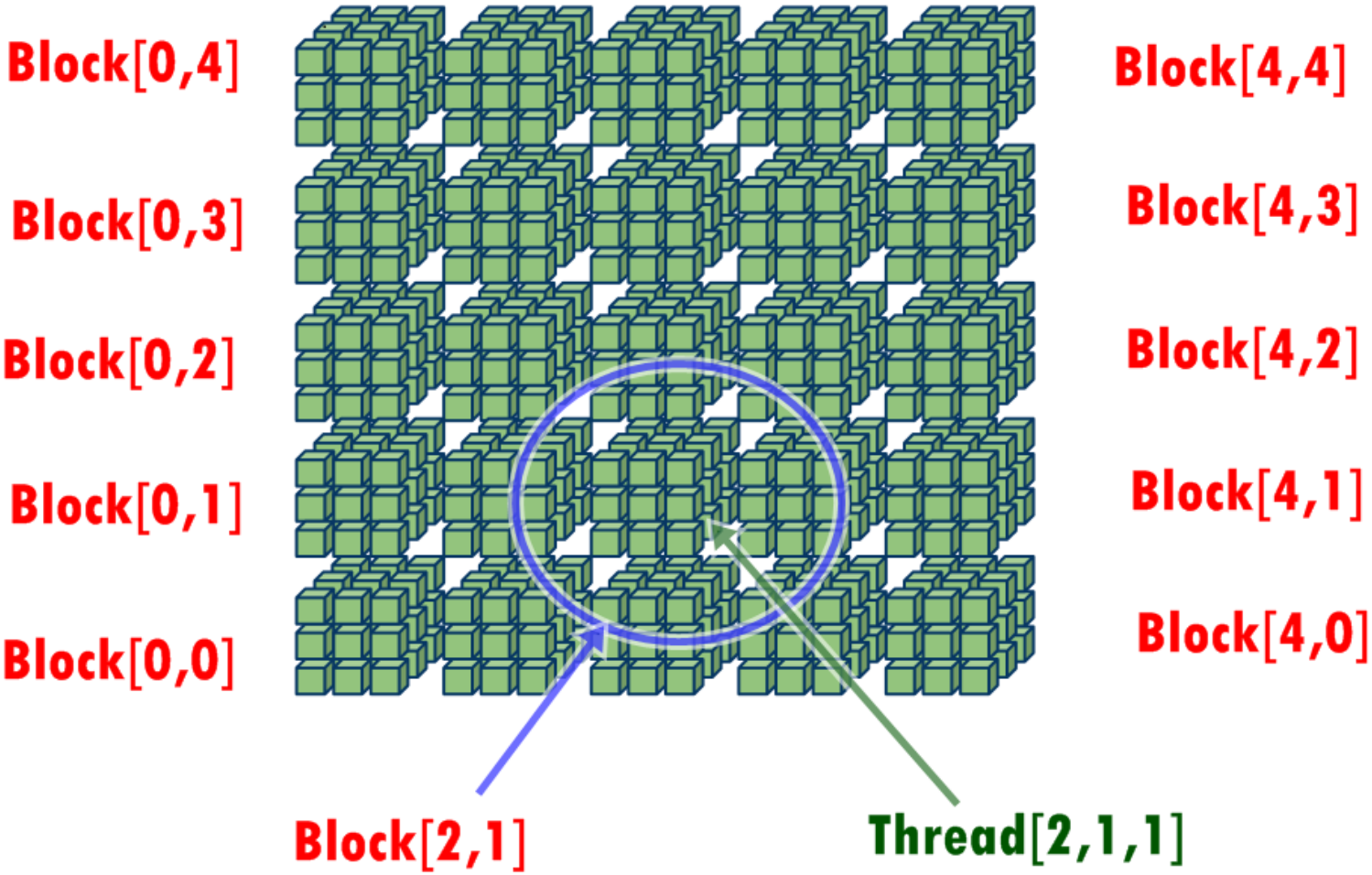
CUDA: Host, Global, Device



CUDA: Host, Global, Device



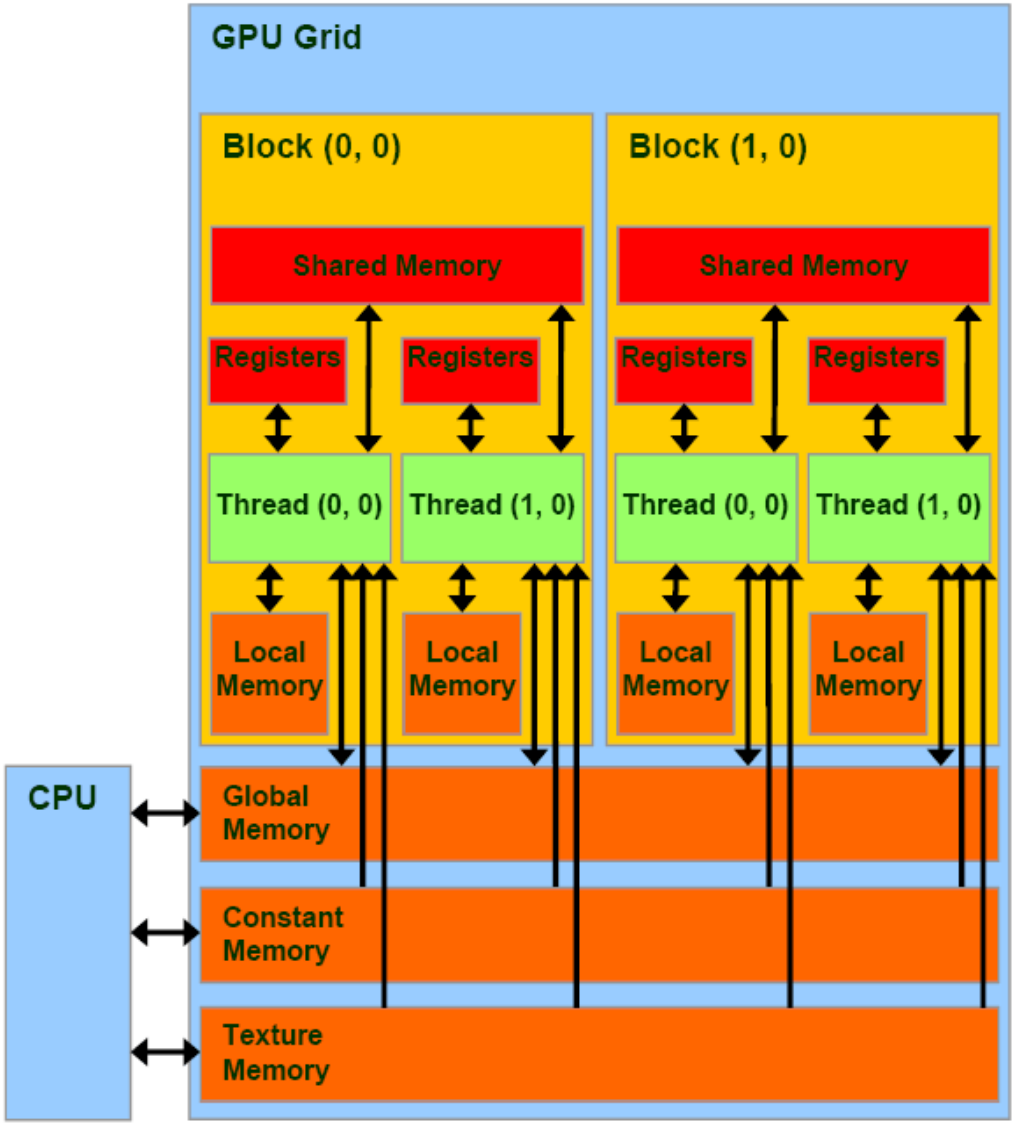
CUDA: Host, Global, Device



CUDA: Memories

- The device memory architecture is rather involved, given the original purpose of the GPU—i.e., a graphical pipeline for image rendering.
- Six different types of memories, with very different properties in terms of location on chip, caching, read/write access, scope and lifetime:
  1. registers (fast)
  2. local (array) memory (slow)
  3. shared memory (fast)
  4. global memory (slow)
  5. constant memory (limited)
  6. texture memory (limited)
- In particular registers and local memory have a thread life span and visibility, while shared memory has a block scope (to facilitate thread cooperation) and the others are permanent and visible from host and every thread on the device. Constant and texture memories are the only memories to be read-only and to be

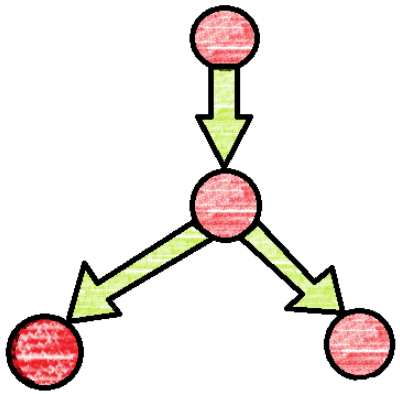
CUDA: Memories



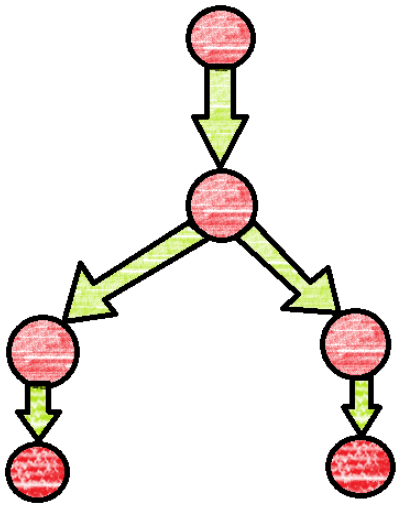
Searching=Propagation+ND assignment

Searching=Propagation+ND assignment

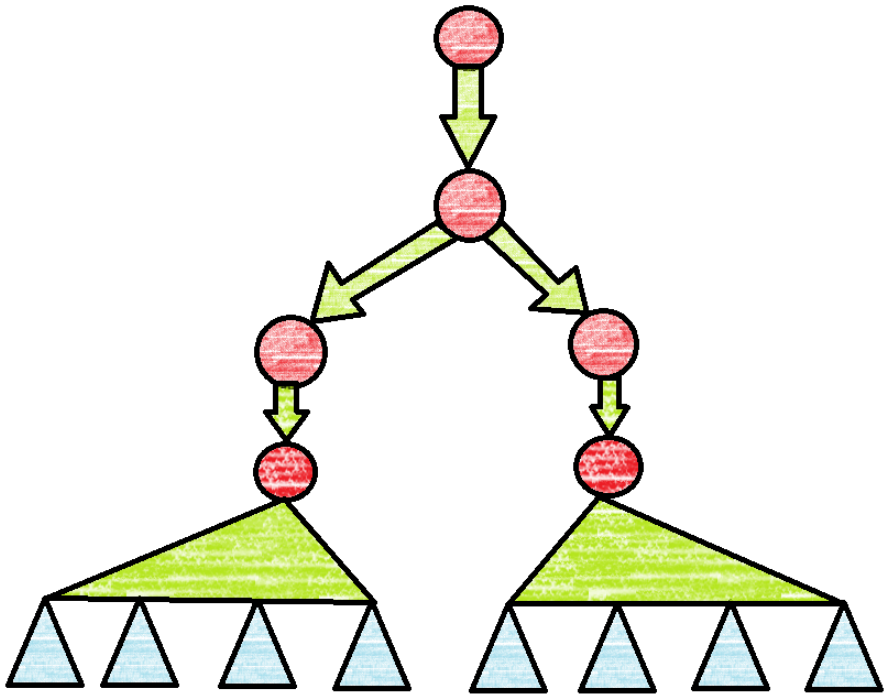
Searching=Propagation+ND assignment



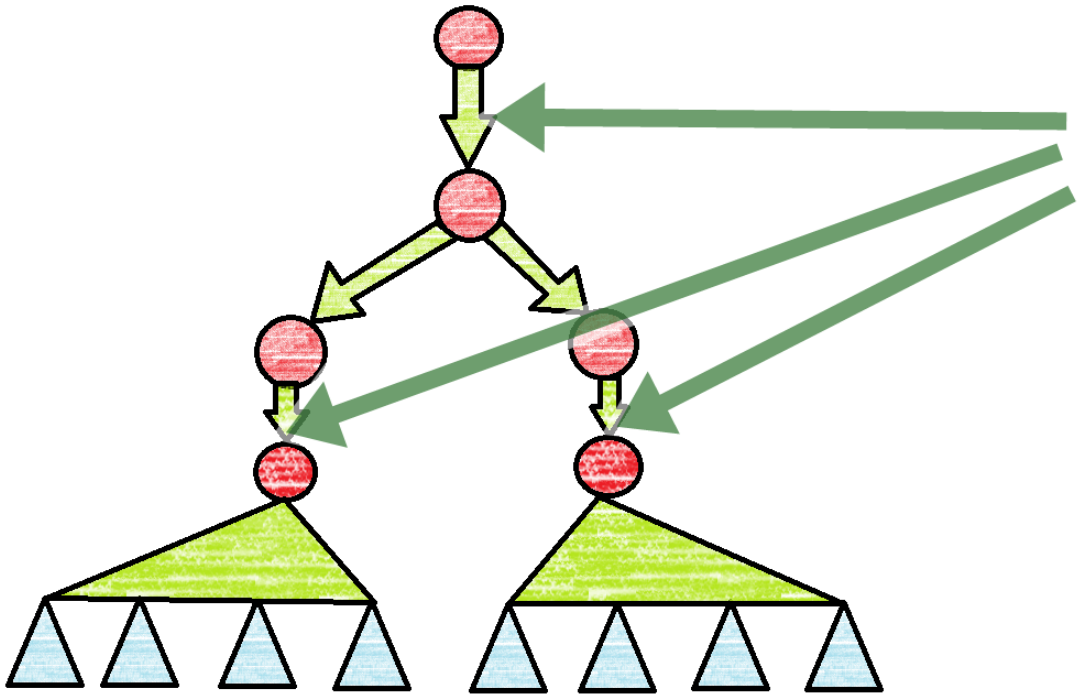
Searching=Propagation+ND assignment



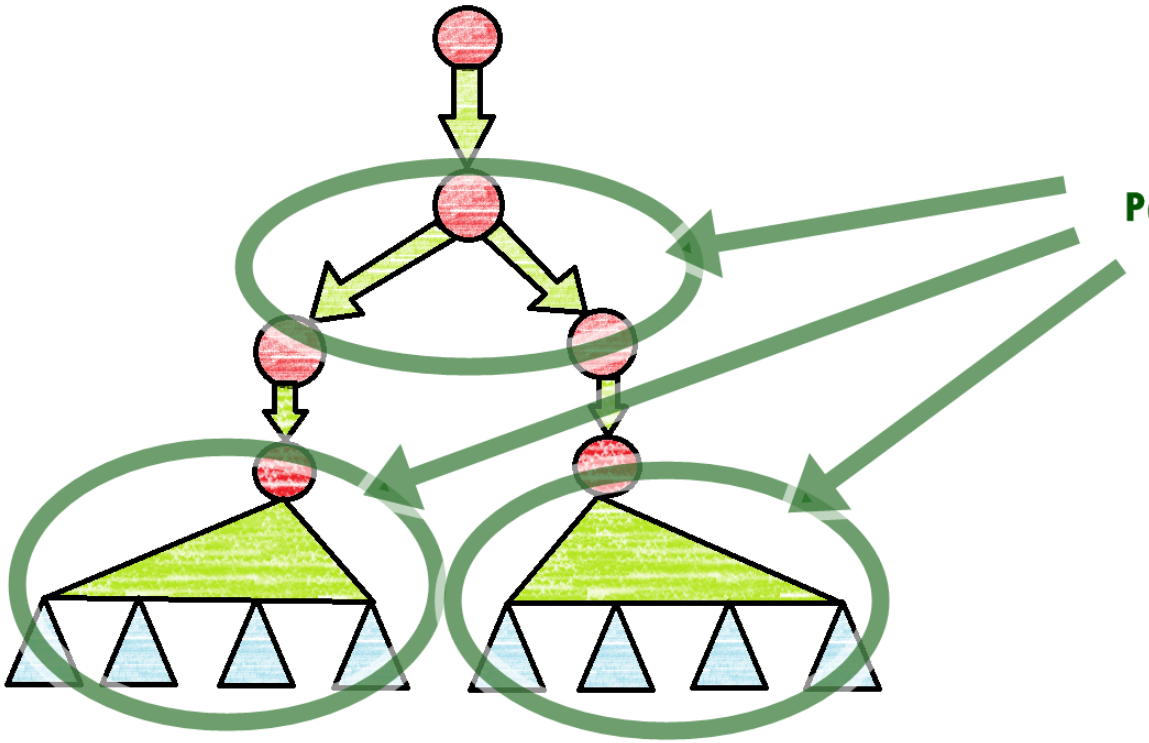
Searching=Propagation+ND assignment



Searching=Propagation+ND assignment



Searching=Propagation+ND assignment



Choices

- Parallelizing an existing state-of-the-art search algorithm (e.g. a constraint solver, an ASP solver, or a SAT solver)

Choices

- Parallelizing an existing state-of-the-art search algorithm (e.g. a constraint solver, an ASP solver, or a SAT solver) while learning CUDA

Choices

- Parallelizing an existing state-of-the-art search algorithm (e.g. a constraint solver, an ASP solver, or a SAT solver) while learning CUDA
- Starting from a simple, well-know algorithm, implementing it in CUDA, and then playing with some parallel versions

Choices

- Parallelizing an existing state-of-the-art search algorithm (e.g. a constraint solver, an ASP solver, or a SAT solver) while learning CUDA
- Starting from a simple, well-know algorithm, implementing it in CUDA, and then playing with some parallel versions
- We choose the second way

SAT and DPLL

Given Φ (CNF), establishing whether exists θ s.t. $\Phi\theta$ is true.

SAT and DPLL

Given Φ (CNF), establishing whether exists θ s.t. $\Phi\theta$ is true.

```
DPLL(Φ,θ)
    θ' ← unit_propagation(Φ,θ)
    if (ok(Φθ')) return θ'
    else if (ko(Φθ')) return false
    else X ← select_variable(Φ,θ')
        θ'' ← DPLL(Φ, θ'[X/true])
        if (θ'' ≠ false) return θ''
        else return DPLL(Φ, θ'[X/false])
```

Parallelizing Unit Propagation

Given a (partial) assignment θ UP computes for each clause i:

- $\texttt{mask}[i] = 0$ if clause i is satisfied by θ;
- $\texttt{mask}[i] = -1$ if all literals of the clause i are falsified by θ;
- $\texttt{mask}[i] = u$ if clause i is not yet satisfied by θ, and there are still $u > 0$ unassigned literals in it.

Parallelizing Unit Propagation

Given a (partial) assignment θ UP computes for each clause i:

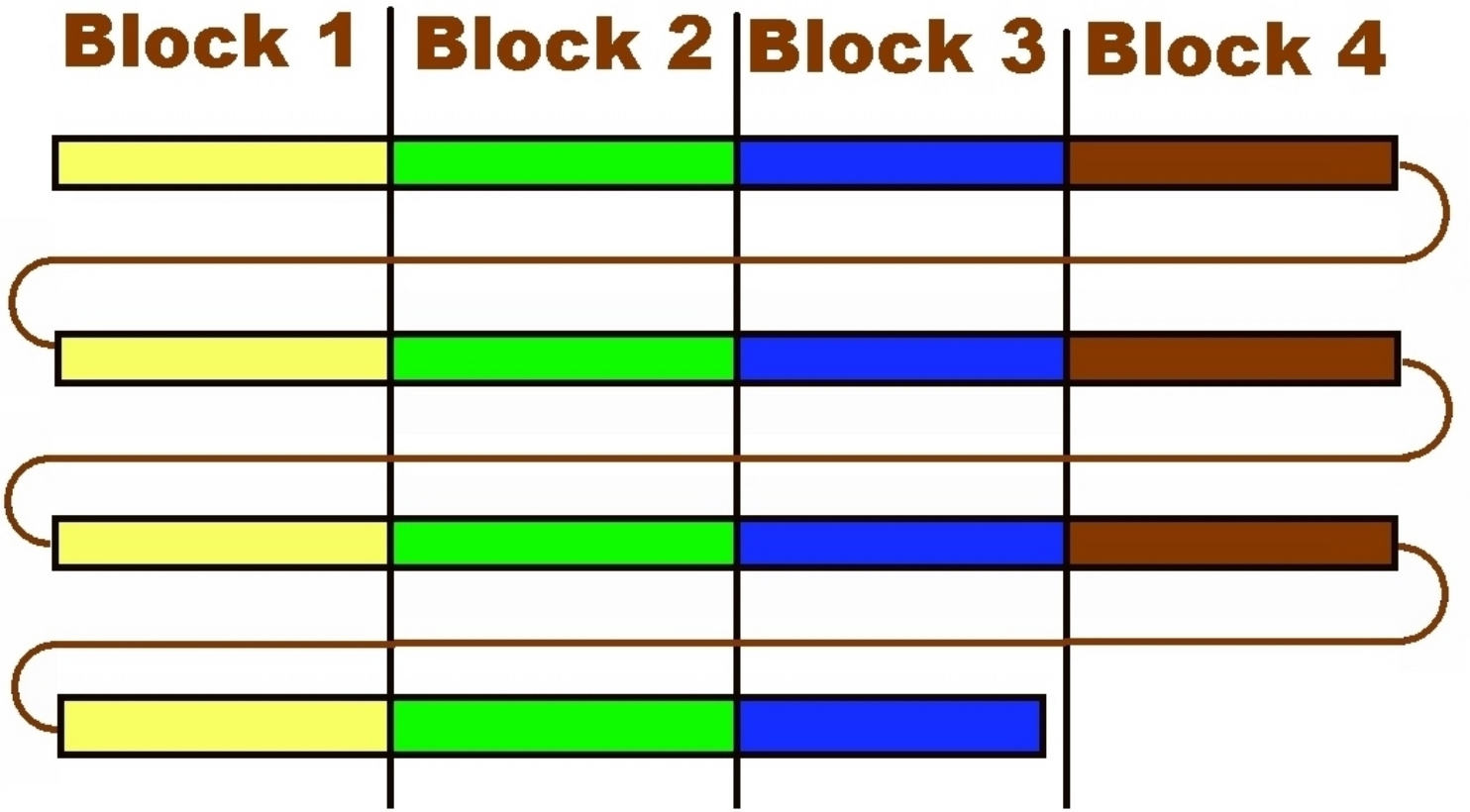
- $\texttt{mask}[i] = 0$ if clause i is satisfied by θ;
- $\texttt{mask}[i] = -1$ if all literals of the clause i are falsified by θ;
- $\texttt{mask}[i] = u$ if clause i is not yet satisfied by θ, and there are still $u > 0$ unassigned literals in it.

The function returns:

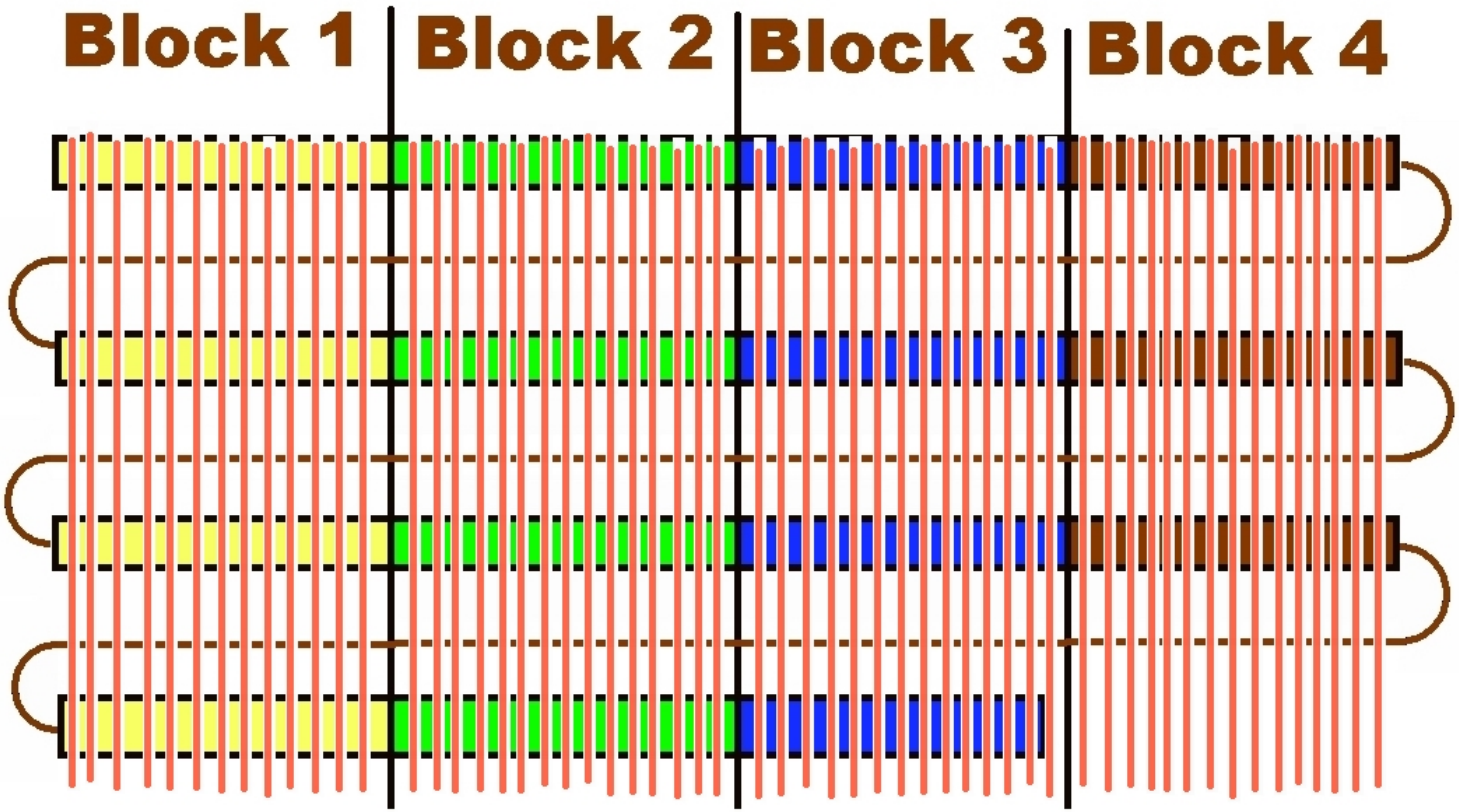
- -1 if there is a value of i such that $\texttt{mask}[i] = -1$,
- 0 if for all i $\texttt{mask}[i] = 0$,
- the pointer to a literal in a clause i with $\texttt{mask}[i] > 0$, otherwise. Different heuristics can be used for this literal selection, but in any case literals in clauses with $\texttt{mask}[i] = 1$ are selected first (unit propagation)

Since we are interested in a unique $\texttt{mask}[i]$ result and a unique pointer, the $\texttt{mask}$ array does not need to be built explicitly.

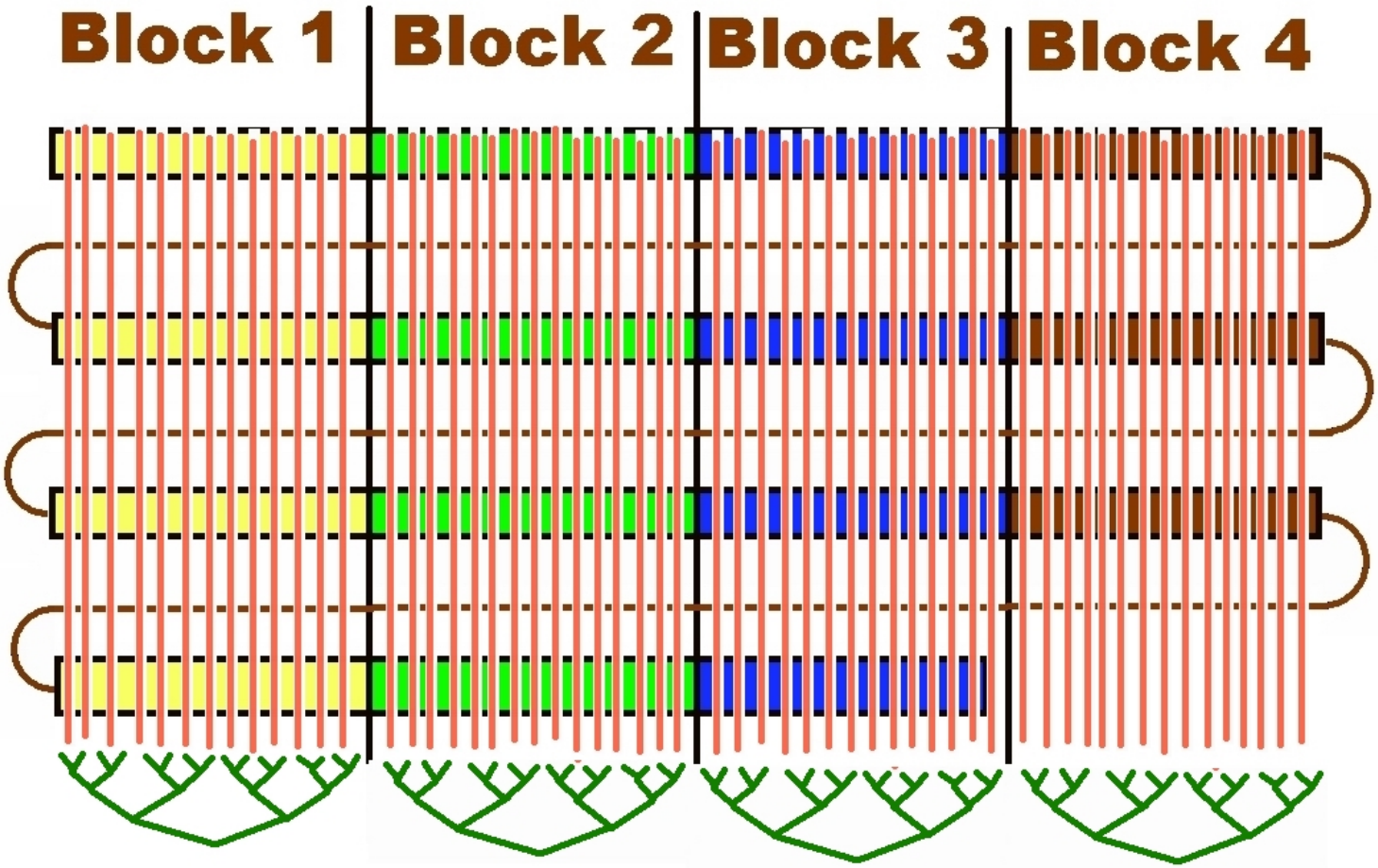
Parallelizing Unit Propagation



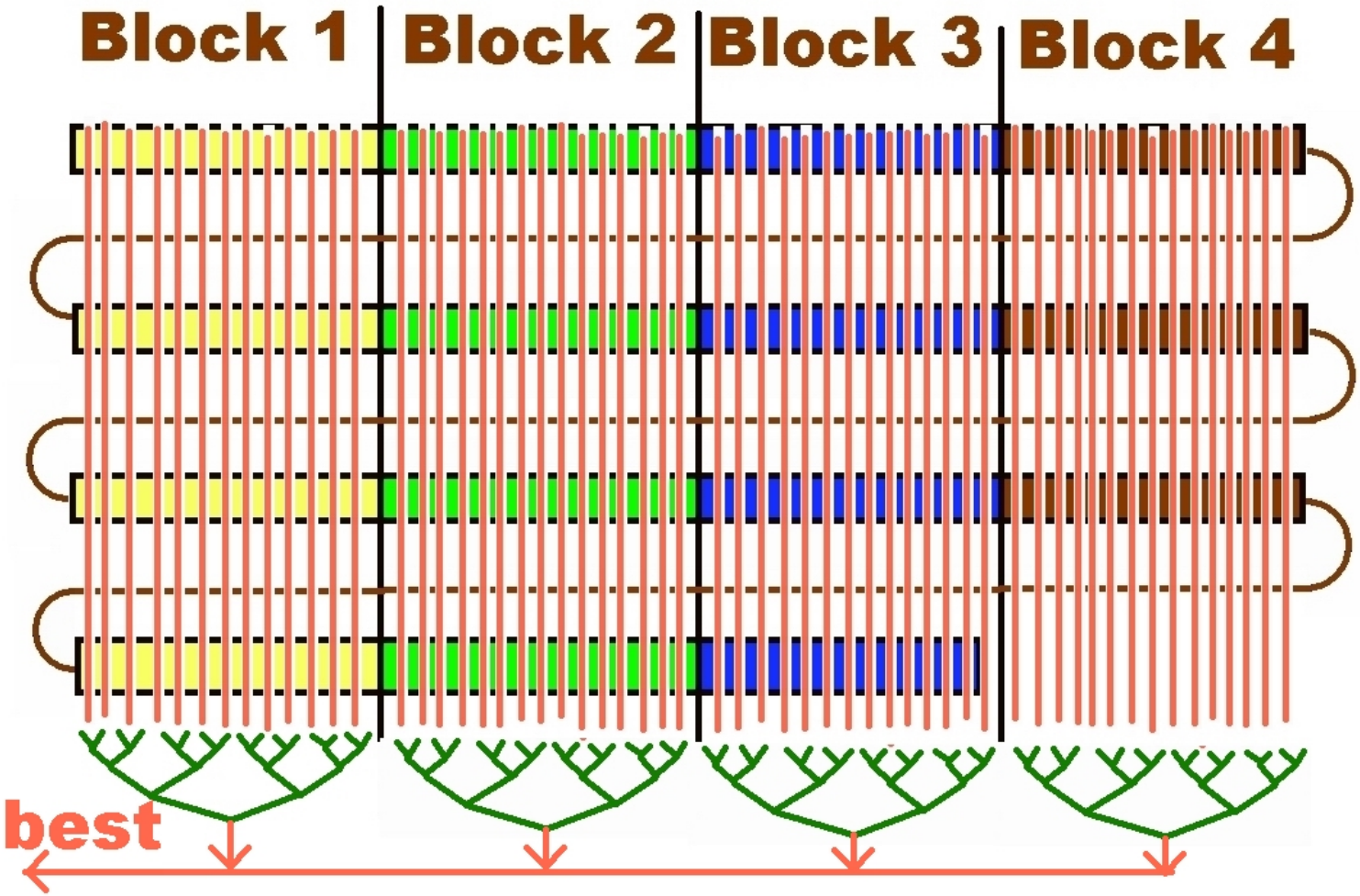
Parallelizing Unit Propagation



Parallelizing Unit Propagation



Parallelizing Unit Propagation



Parallelizing Unit Propagation

- Let SM be the number of multiprocessors (1 MP = 48/32 cores).
- We use 2 SM blocks.
- Each block addresses 512 threads.
- The formula is divided into areas of consecutive 512 clauses that are assigned to each thread of a block.
- A block will process more areas until the whole formula is processed.
- Each thread is in charge of computing the `mask` value for a set of clauses and it stores (in local shared memory) the best among the set.
- A synchronization barrier is enforced before applying a block *reduction*: a logarithmic schema that sorts and extracts the best candidate among threads' results.
- Finally, a single thread in the block is in charge to store the result in the GPU global memory.

Parallelizing Unit Propagation

- Predicting the speed-up is not easy.
- Basically, if we use k blocks, each block deals with $\mathtt{NC}/k$ clauses.
- Each block executes 512 threads.
- Recall that the number of blocks chosen is double than the number of SM and each SM has 48/32 computing cores.
- One has to consider CUDA's peculiar memory access, as well as the fact that the scheduler reasons at the warp level (groups of 32 threads).
- Moreover, each block has to wait for the thread that processes the largest number of literals.

Parallelizing Search

- If/when the formula (reduced by current θ) is large but not huge, we can parallelize the search in it
- We filter the formula using theta (if θ satisfies a clause we remove it; we remove all instantiated literals from other clauses)
- Then we launch parallel execution
- Three parameters: number of blocks `B`, number of threads per block `T`, notion of large (based on the number of free variables, `MaxV`)

Parallelizing Search

Fix the block-variables

```
point=-1;
addr=blockIdx.x
for(i=1;i<NV;i++){
  if (count < log(B))
        { block_vars[i] = addr % 2;
          addr = addr/2; count++; }
   else { block_vars[i] = point;
          point--; }
 }
```

Parallelizing Search

Fix the thread-variables

```
DPLL(Φ,θ)
    θ' ← unit_propagation(Φ,θ)
    if (ok(Φθ')) return θ'
    else if (ko(Φθ')) return false
    else X ← select_variable(Φ,θ')
        θ'' ← DPLL(Φ, θ'[X/true])
        if (θ'' ≠ false) return θ''
        else return DPLL(Φ, θ'[X/false])
```

Parallelizing Search

Fix the thread-variables

addr = threadIdx.x; count = 0;

DPLL(Φ, θ)

$\theta' \leftarrow$ unit_propagation(Φ, θ)

if (ok($\Phi\theta'$)) return θ'

else if (ko($\Phi\theta'$)) return false

else $X \leftarrow$ select_variable(Φ, θ')

if (count $< \log_2(T)$)

t = addr % 2; addr = addr/2;

return DPLL(Φ, $\theta'[X/t]$);

else

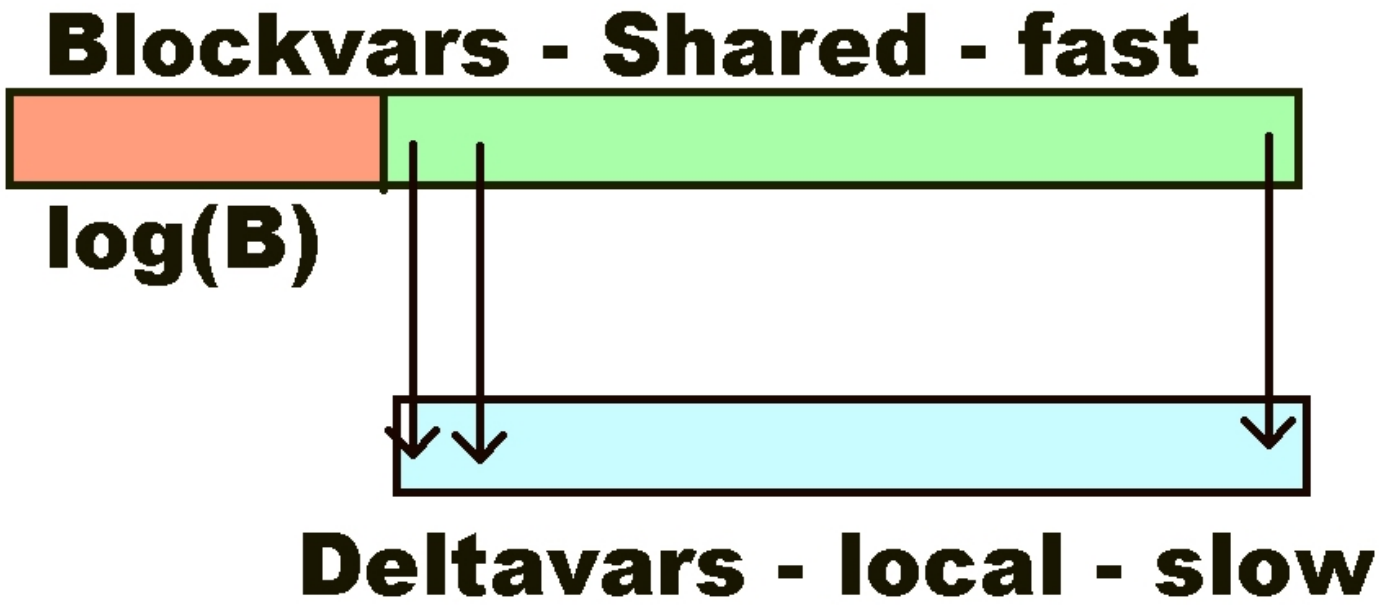
$\theta'' \leftarrow$ DPLL(Φ, $\theta'[X/\text{true}]$)

if ($\theta'' \neq$ false) return θ''

else return DPLL(Φ, $\theta'[X/\text{false}]$)

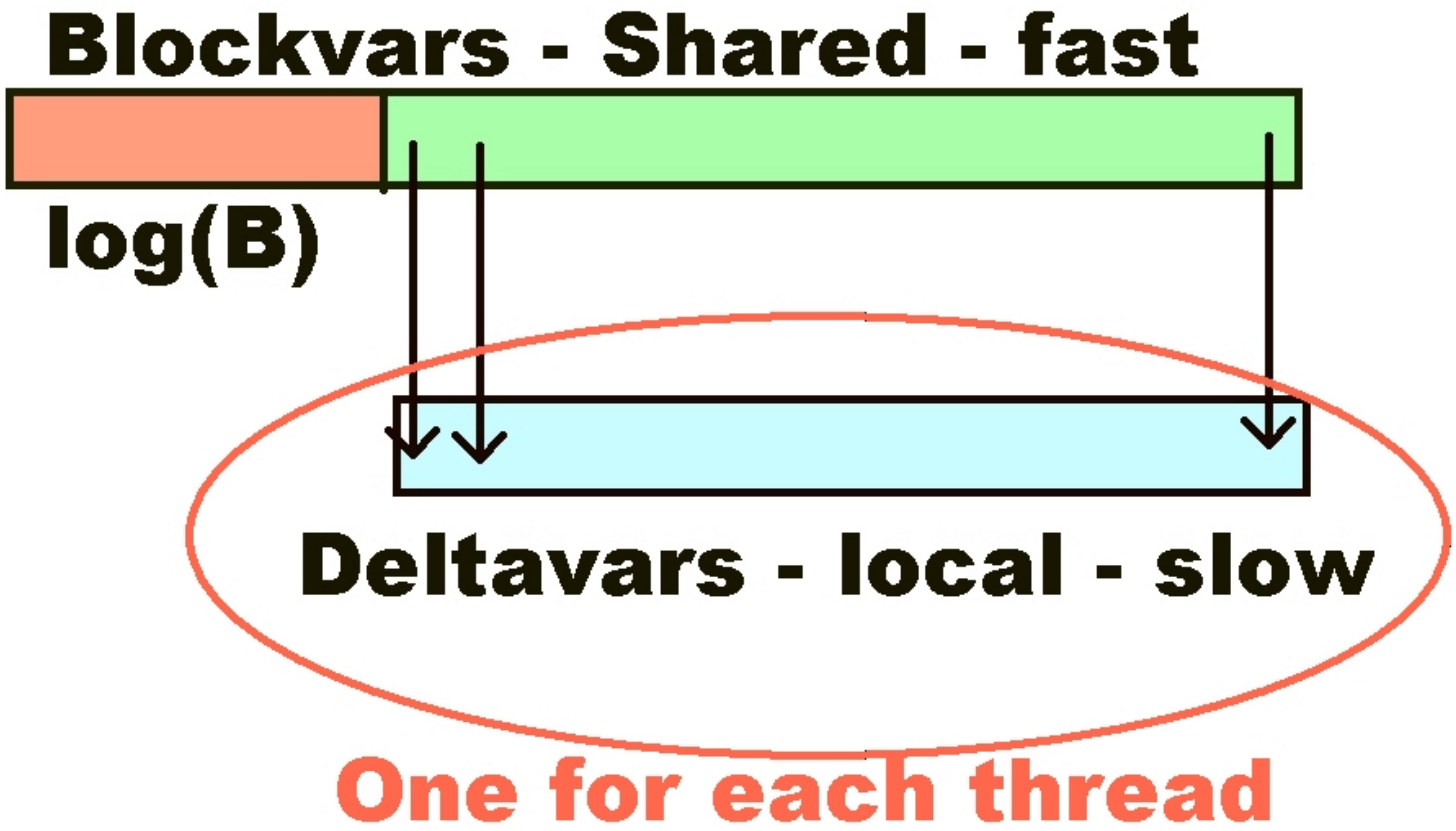
Parallelizing Search

Some remarks on memories



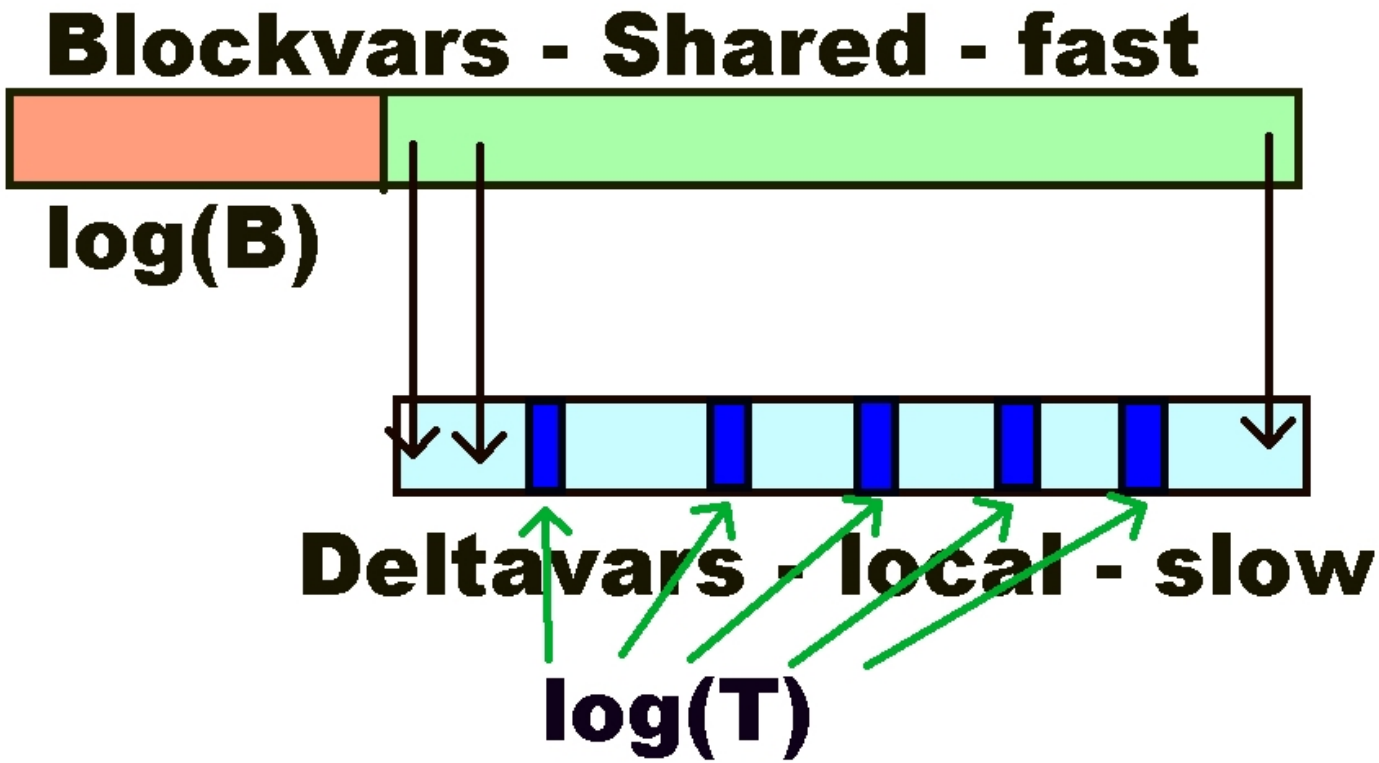
Parallelizing Search

Some remarks on memories

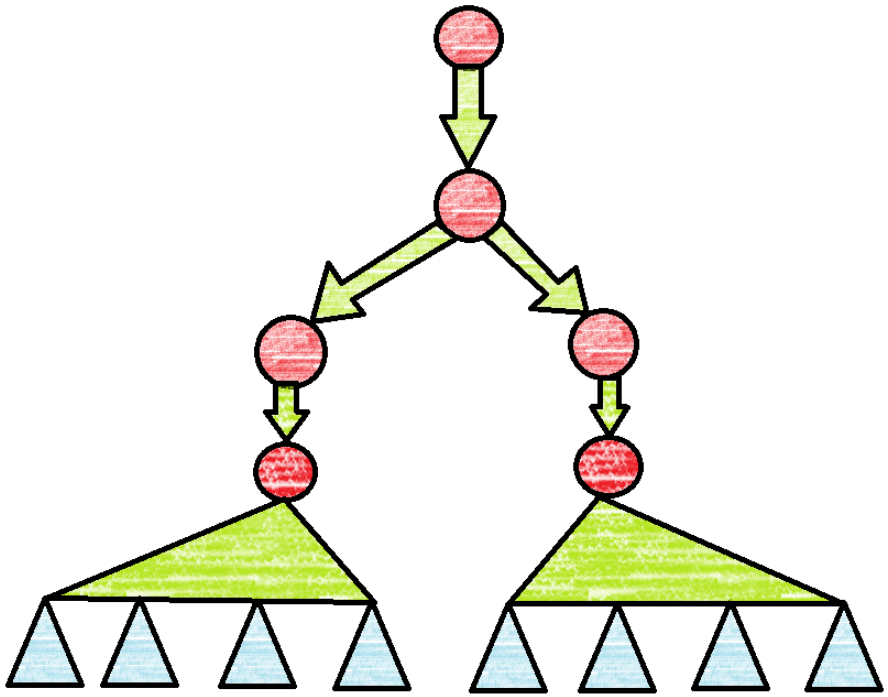


Parallelizing Search

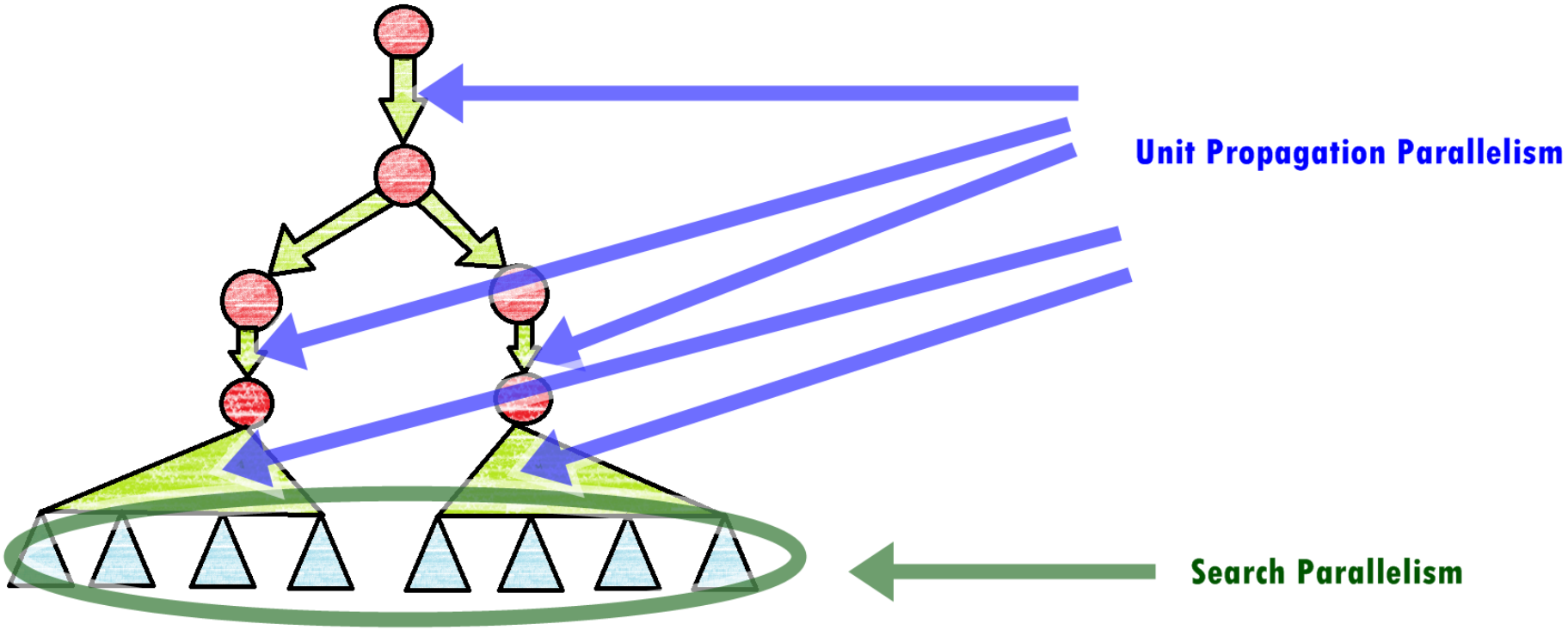
Some remarks on memories



Summary



Summary



HW used

U: (Udine)

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

F: (Fermi)

- *Host*: (12 core) Xeon e55645 at 2.4GHZ, 32GB RAM.
- *Device*: NVIDIA Tesla C2075, 448 cores (14MP). Processor Clock 1.15GHz.

A: (Agostino's desktop)

- *Host:* Intel core i7, Windows 7, 64 bits, 3.4–3.7GHz
- *Device:* NVIDIA GeForce GTX 560, 336 cores (7MP). Processor Clock 1.62–1.9GHz.

All Host CPUs are used as single core in tests.

Results

Unit Propagation

	2	4	8	16	32	64	128	256	512	1024	2048	4096
F0	0.06	0.12	0.23	0.63	0.88	2.02	3.5	6.98	13.97	28.01	55.99	111.96
F1	2.67	2.68	2.74	2.73	2.73	2.82	3.04	3.22	3.78	4.8	6.99	11.44
U0	0.12	0.22	0.43	0.86	1.71	3.46	6.91	13.81	27.59	57.61	116	232.42
U1	0.92	0.95	0.94	0.97	1.1	1.2	1.42	1.94	2.9	4.77	8.37	15.82
A0	0.05	0.09	0.18	0.36	0.71	1.41	2.81	5.63	11.27	19.77	39.16	80.49
A1	1.75	1.72	1.73	1.69	1.79	1.83	1.95	2.21	2.62	3.52	5.24	9.02

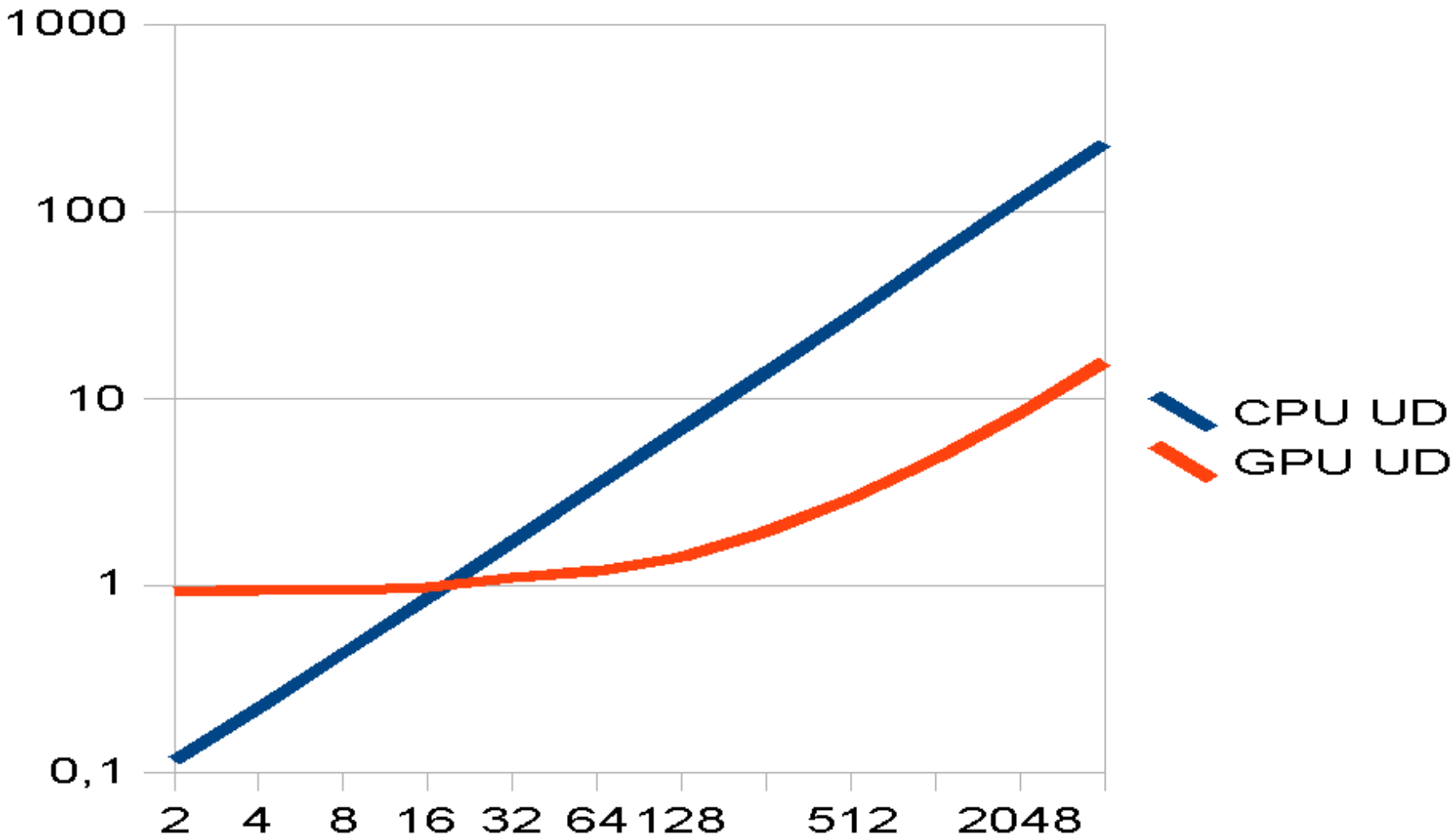
Behaviour of the options:

- 0 (only host) and
- 1 (CUDA U.P.)

on multiple copies (2, . . . , 4096) of the unsatisfiable instance `hole6`.
Hardware: F (Fermi), U (Udine), and A (Ago).

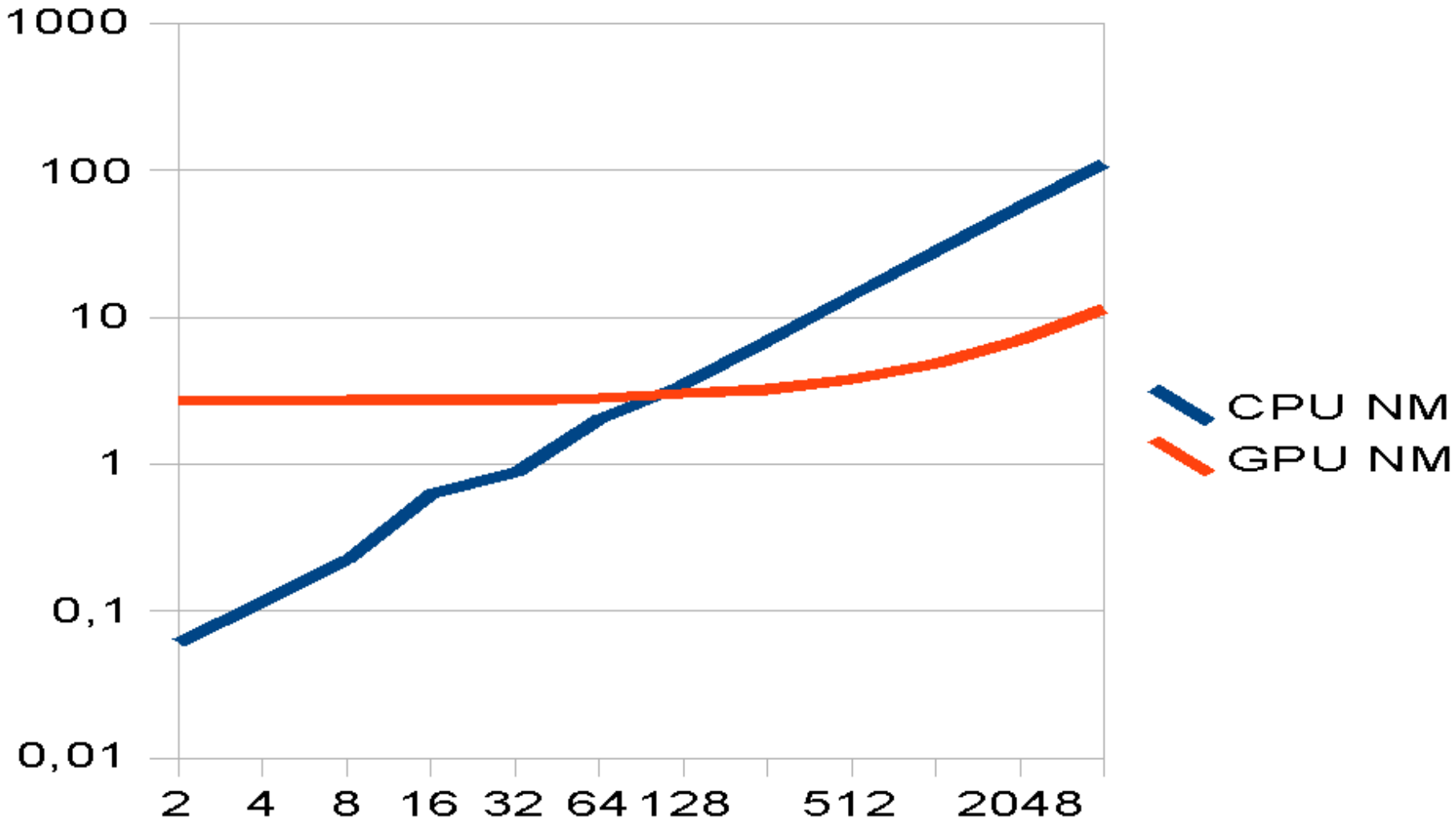
Results

Unit Propagation



Results

Unit Propagation



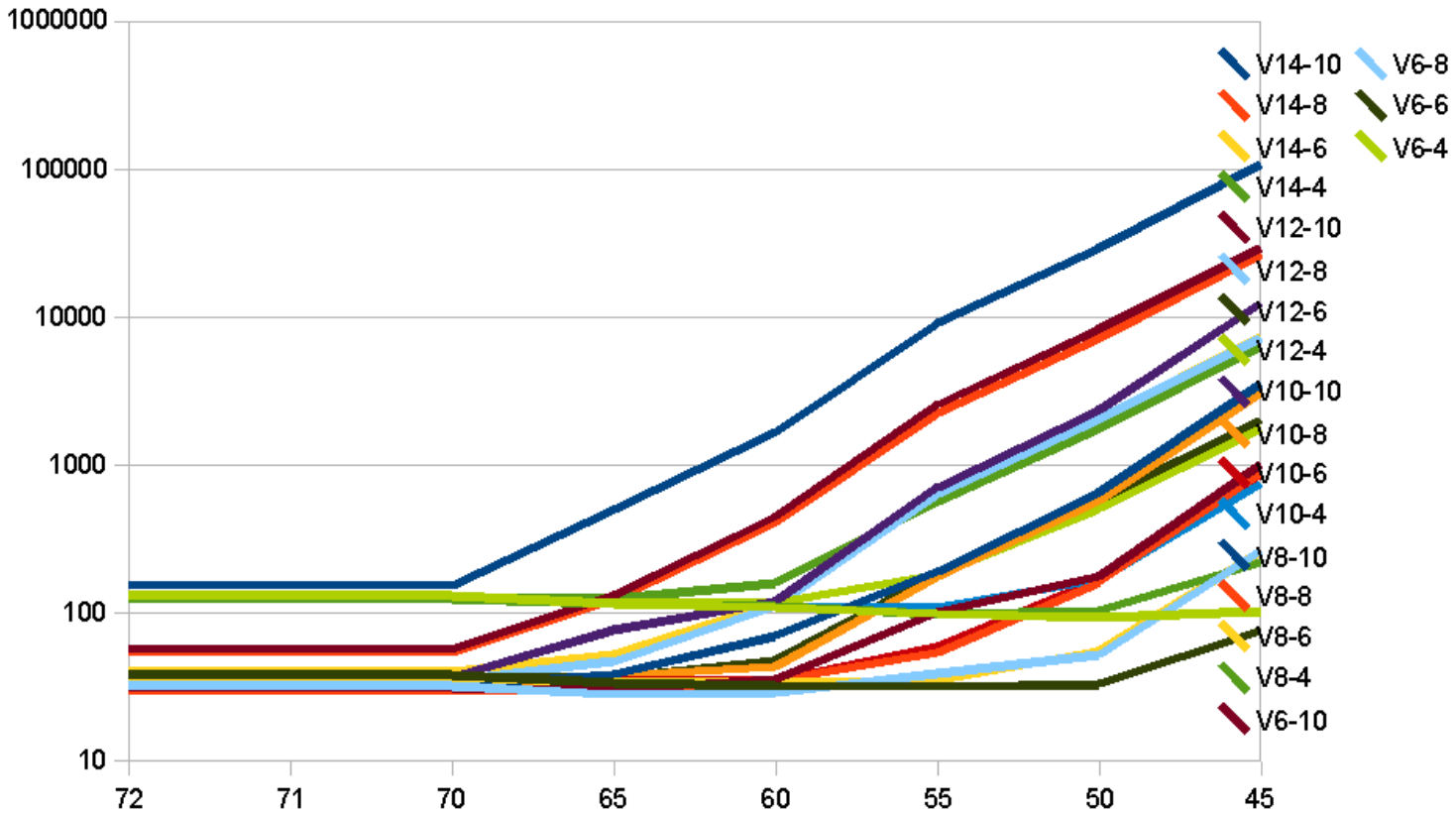
Results (Search)

x1_24.shuffled07 (SAT Competition 2002)

	U (192)	A (336)	F (448)
CPU	18m36s	6m44s	14m14s
GPU (6,6,72)	37.74	29.95	22.61
GPU (6,8,60)	28.34	15.61	13.76
Speed up	40	29	62

Results (Search)

x1_24.shuffled07 (SAT Competition 2002)



HW: U. On CPU: 1116s

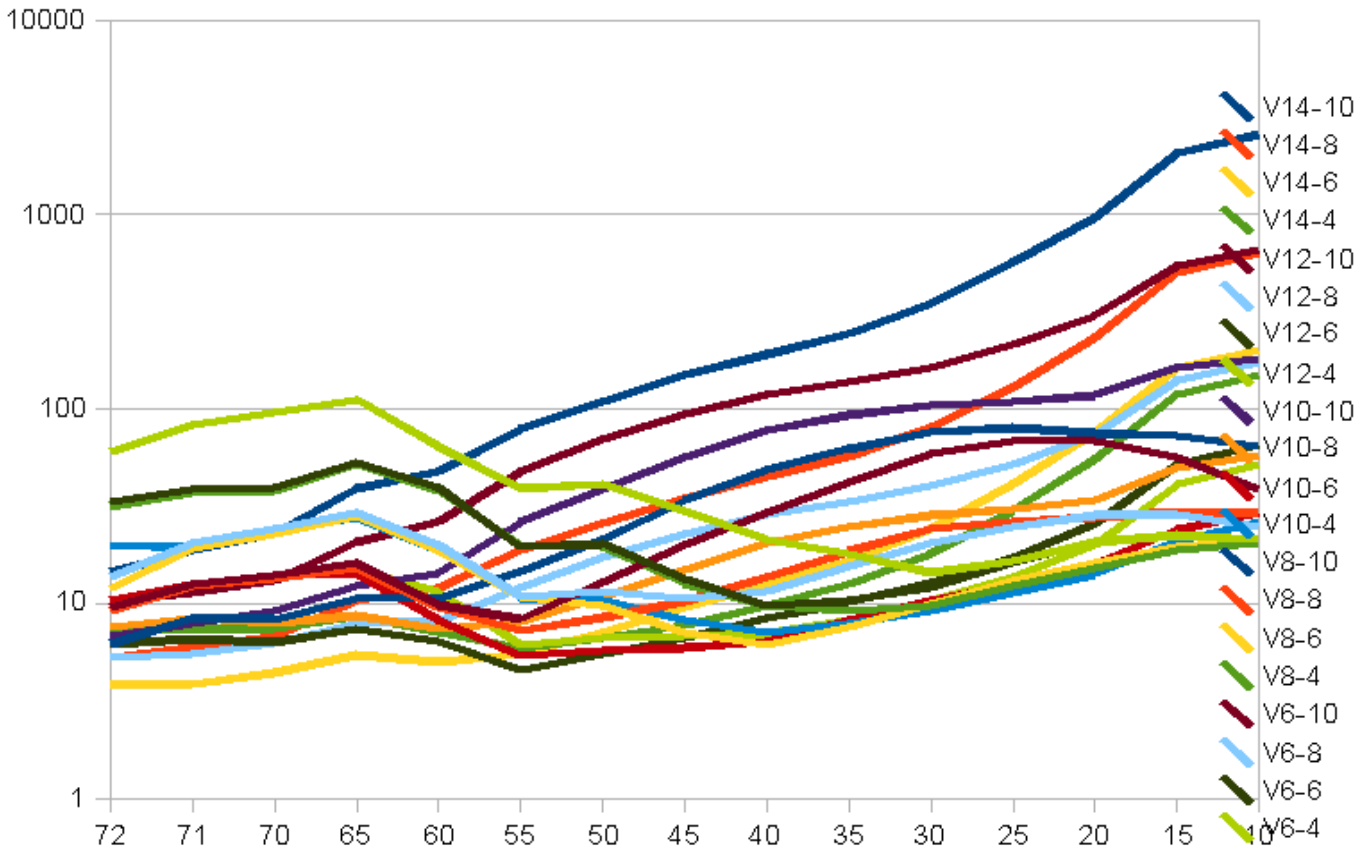
Results (Search)

hole8

	U	A	F
CPU	13.76	5.70	6.94
GPU (6,6,72)	26.86	22.10	33.28
GPU (14,6,72)	6.47	3.67	3.82
Speed up	2.1	1.5	1.8

Results (Search)

hole8



HW: U. On CPU: 13.7s

Results (Search)

Instances “BIG” — 0/1 GPU call

$$\text{V=2: } \underbrace{(\neg x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \wedge) \wedge (x_1 \vee \neg x_2)}_{2^{|V|}-1}$$

Results (Search)

Instances “BIG” — 0/1 GPU call

$$\text{V=2: } \underbrace{(\neg x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \wedge) \wedge (x_1 \vee \neg x_2)}_{2^{|V|}-1} \wedge (x_1 \vee x_2)$$

Results (Search)

Instances "BIG" — 0/1 GPU call

V=2: $\underbrace{(\neg x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \wedge) \wedge (x_1 \vee \neg x_2)}_{2^{|V|}-1} \wedge (x_1 \vee x_2)$

	U (192)			A (336)			F (448)		
V	CPU	(6,6)	S-up	CPU	(6,6)	S-up	CPU	(6,6)	S-up
14 (N)	8.41	0.46	18.30	2.83	0.32	8.84	4.31	0.28	15.35
14 (Y)	4.16	0.46	9.04	1.36	0.32	4.25	2.27	0.28	8.1
15 (N)	58.28	1.41	42.04	10.18	1.00	10.18	17.18	0.86	19.91
15 (Y)	29.21	1.41	20.71	5.44	1.00	5.44	9.14	0.87	10.54
16 (N)	265.51	4.68	56.73	41.30	3.58	11.53	68.85	2.95	23.32
16 (Y)	142.56	4.68	30.46	21.35	3.58	5.96	36.51	2.95	12.38

Conclusions (or Start?)

- The question was: can we exploit this computation power for computational logics?

Conclusions (or Start?)

- The question was: can we exploit this computation power for computational logics?
- The answer is maybe

Conclusions (or Start?)

- The question was: can we exploit this computation power for computational logics?
- The answer is maybe
- First results are encouraging.

Conclusions (or Start?)

- The question was: can we exploit this computation power for computational logics?
- The answer is maybe
- First results are encouraging.
- However, if calling parameters are not tuned we don't have the expected results

Conclusions (or Start?)

- The question was: can we exploit this computation power for computational logics?
- The answer is maybe
- First results are encouraging.
- However, if calling parameters are not tuned we don't have the expected results
- Next stage: learning of parameters depending on instance and HW

Conclusions (or Start?)

- The question was: can we exploit this computation power for computational logics?
- The answer is maybe
- First results are encouraging.
- However, if calling parameters are not tuned we don't have the expected results
- Next stage: learning of parameters depending on instance and HW
- Then, porting the ideas on a state-of-the art SAT solver

Conclusions (or Start?)

Meanwhile, using GPU in the correct way: