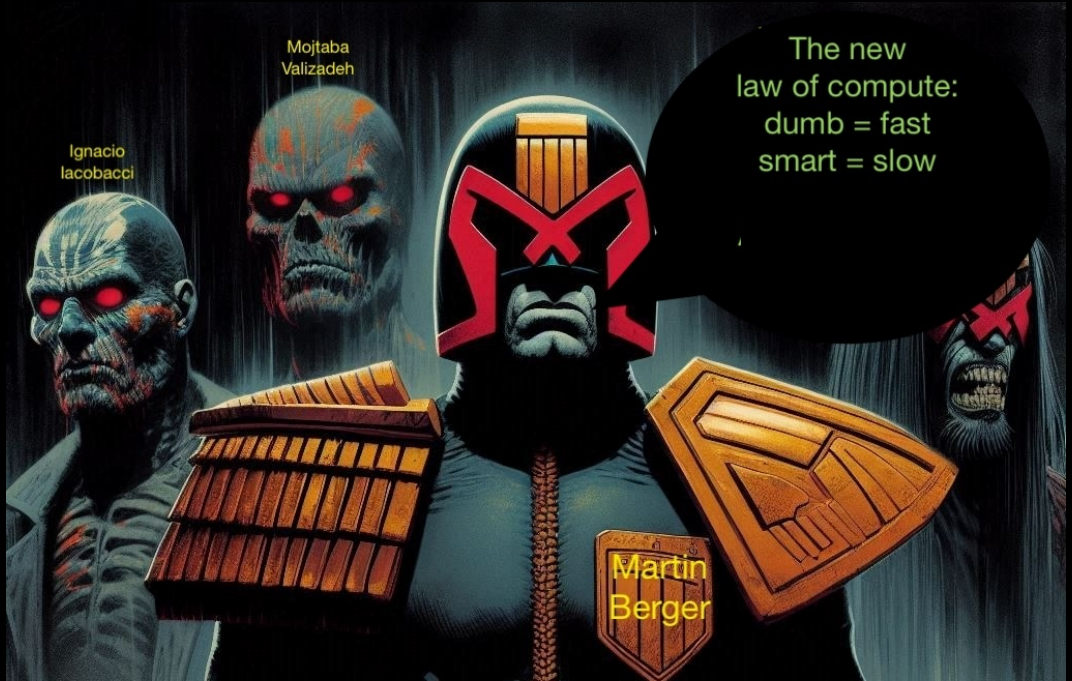
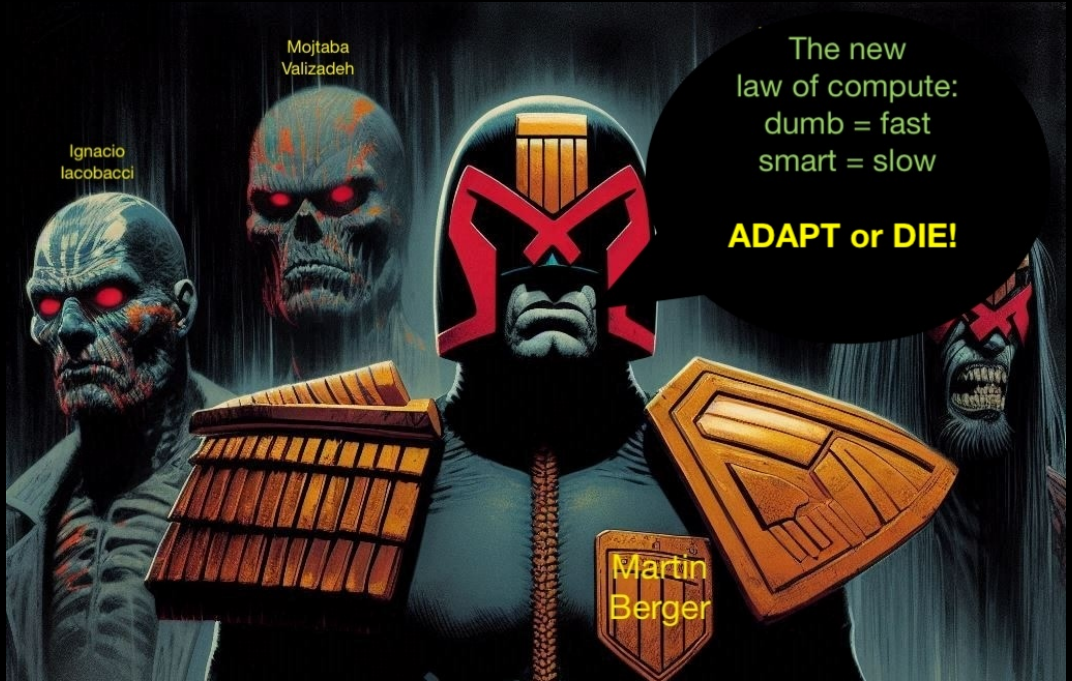
Somewhere in a cold land ...


Mojtaba
Valizadeh
Nathaniel
Fijalkow
Ignacio
Iacobacci
Philip
Gorinski
Martin
Berger


Mojtaba
Valizadeh
Ignacio
Iacobacci
The new
law of compute:
Martin
Berger


Mojtaba
Valizadeh
Ignacio
Iacobacci
The new
law of compute:
dumb = fast
smart = slow
Martin
Berger


Mojtaba
Valizadeh
Ignacio
Iacobacci
The new
law of compute:
dumb = fast
smart = slow
ADAPT or DIE!
Martin
Berger

Towards GPU-accelerated automated reasoning

Martin Berger

University of Sussex Montanarius Ltd

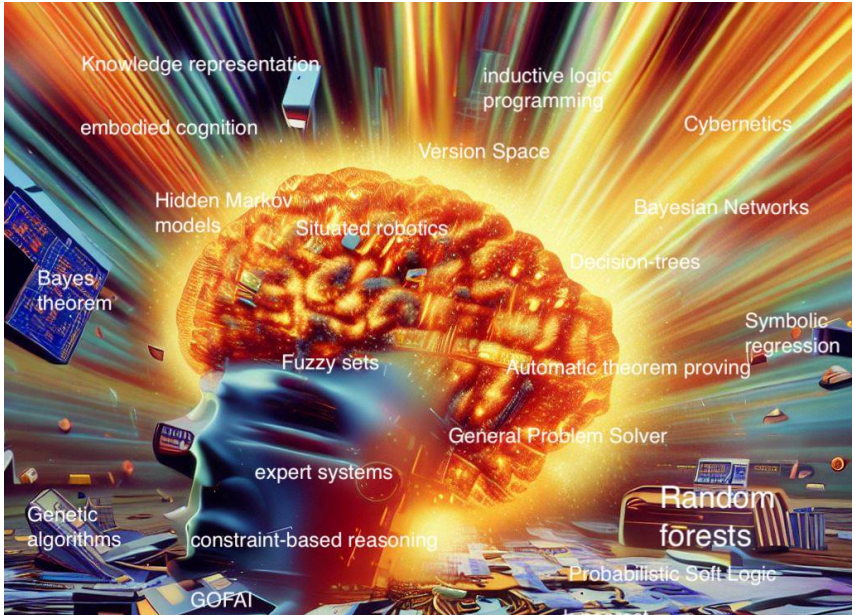
Praha, 15 November 2024

I  NP-hard algorithms

or: How I Learned to Stop Worrying and Love the Exponential!

Before 2004ish

Before 2004ish



Today

Today

Why?

The Bitter Lesson

Rich Sutton

March 13, 2019

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. The ultimate reason for this is Moore's law, or rather its generalization of continued exponentially falling cost per unit of computation. Most AI research has been conducted as if the computation available to the agent were constant (in which case leveraging human knowledge would be one of the only ways to improve performance) but, over a slightly longer time

Wait a minute ...

The Bitter Lesson

Rich Sutton

March 13, 2019

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. The ultimate reason for this is Moore's law, or rather its generalization of continued exponentially falling cost per unit of computation. Most AI research has been conducted as if the computation available to the agent were constant (in which case leveraging human knowledge would be one of the only ways to improve performance) but, over a slightly longer time

Conundrum

When we started this research, November 2022, there was **NO** GPU acceleration of ML technique other than neural nets. And (almost) no GPU acceleration of verification workloads.

Conundrum

When we started this research, November 2022, there was **NO** GPU acceleration of ML technique other than neural nets. And (almost) no GPU acceleration of verification workloads.

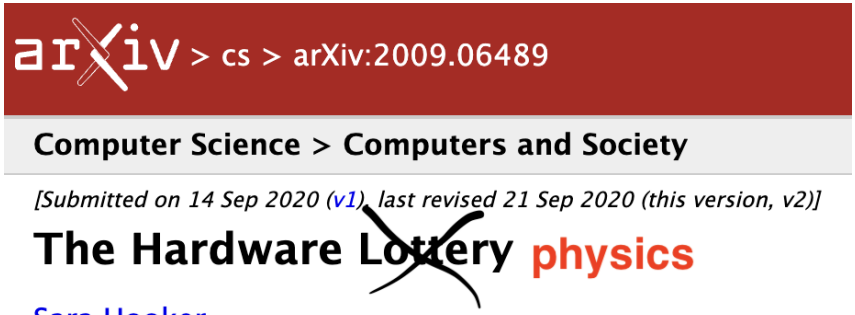
Computer Science > Computers and Society

[Submitted on 14 Sep 2020 (v1), last revised 21 Sep 2020 (this version, v2)]

The Hardware Lottery

Sara Hooker

Hardware, systems and algorithms research communities have h
motivation to engage with each other explicitly. This historical tr
frequently determined which research ideas succeed (and fail). T



Can we de-randomise the hardware lottery?

All models are wrong but some are useful. (G. Box)

All models are wrong but some are useful. (G. Box)

$$15\,048 = \infty,$$

All models are wrong but some are useful. (G. Box)

$15\,048 = \infty, 900\,000 = \infty$

All models are wrong but some are useful. (G. Box)

$15\,048 = \infty$, $900\,000 = \infty$

Compute is free, data-movement is expensive

My motivation for this talk

Today, Moore's law is delivered by GPUs, the work-horses of high-performance computation. The acceleration they provide to applications compatible with their programming paradigm can surpass CPU performance by several orders of magnitude.

Many applications related to logic and verification cannot, today, be accelerated on GPUs, holding back scientific progress.

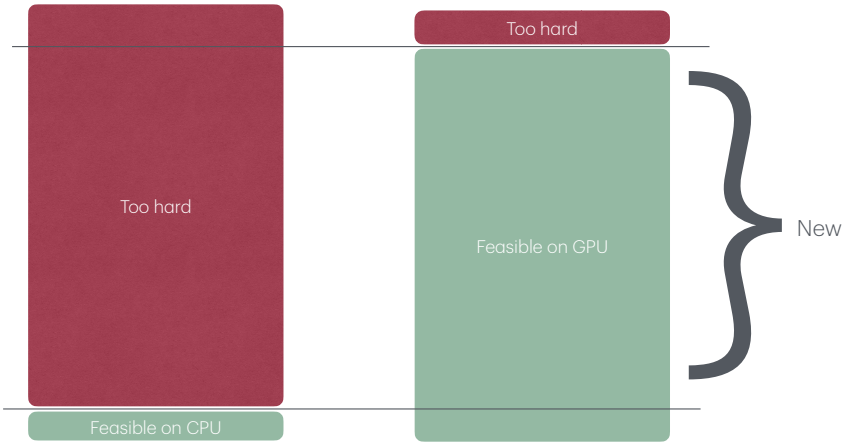
I want to convince you that there is now strong evidence that at least some workloads arising in formal methods and automated reasoning benefit massively from being ported to GPUs, and that the time is ripe for sustained research and engineering in this subject.

Note

The G in GPU stands for “good” in my talk, and includes accelerators like TPUs, Cerebras, Groq ...

Note

Hit from exponential scaling not avoided, just delayed ...



Problem with GPU programming

Problem with GPU programming

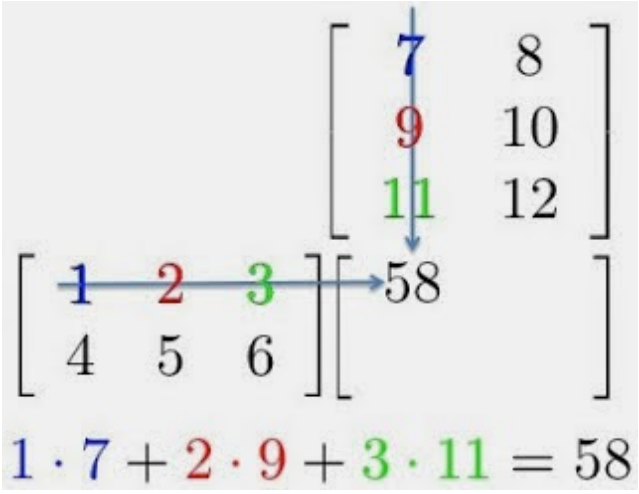
Hard

Essence of GPU-friendly programming

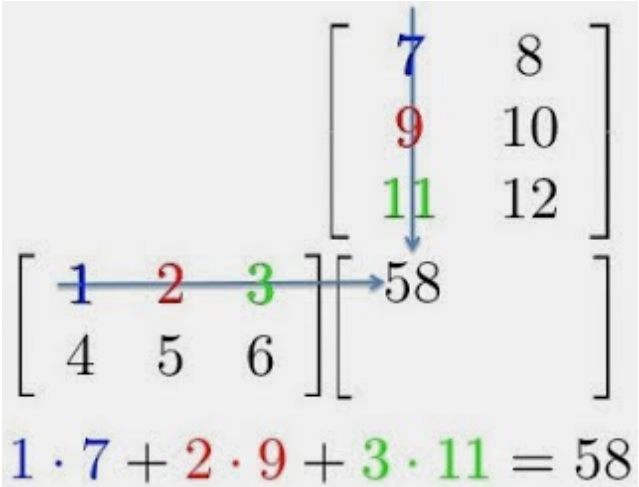
Essence of GPU-friendly programming

- Minimise data movement
- Make data movement predictable (ideally linear scan)
- Minimise data-dependent branching
- Maximise parallelism (and avoid synchronisation between threads)
- Maximise lock-step parallelism (SIMD)

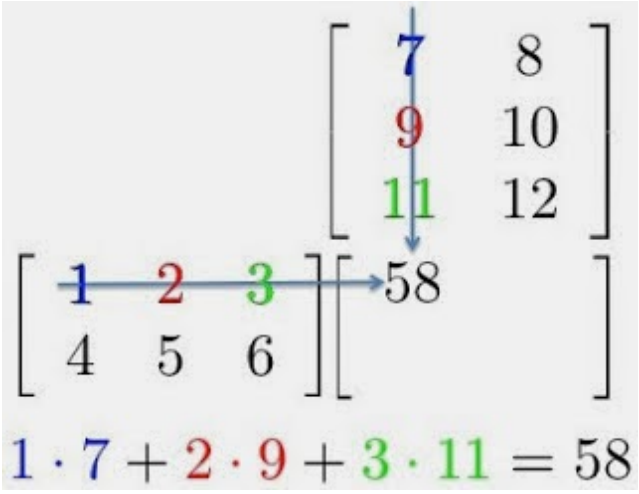
NNs ≈ generalised matrix ops



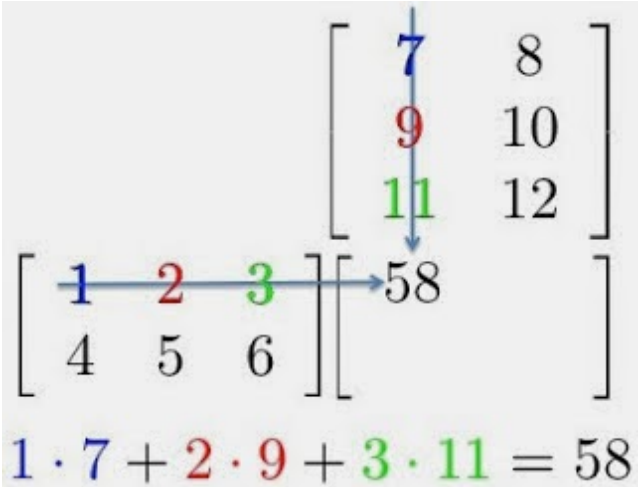
Predictable data movement



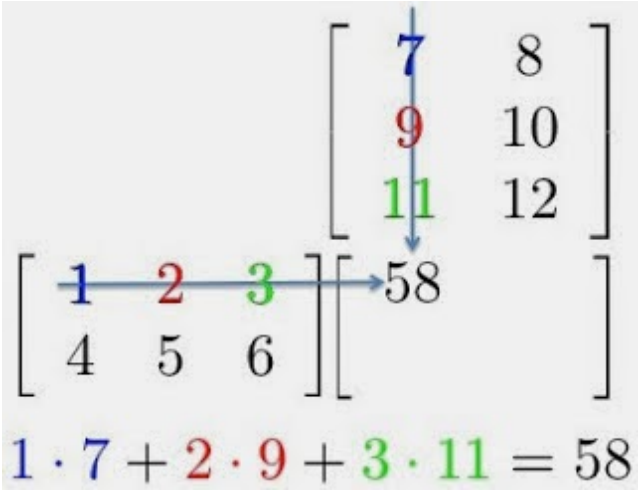
Minimise data-dependent branching



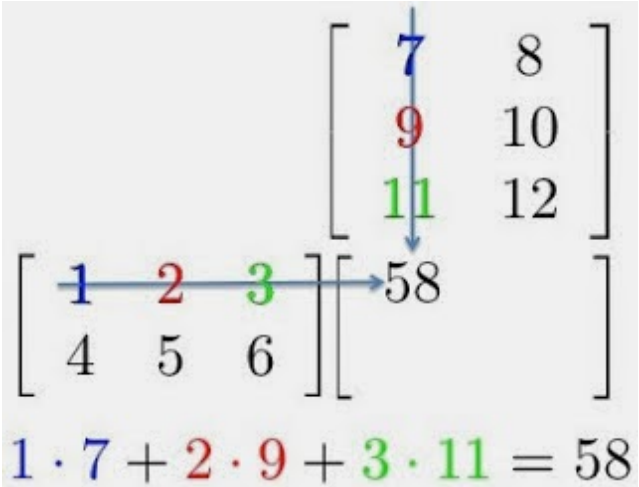
Maximise parallelism



Lock-step parallelism (SIMD)



Minimise synchronisation



Essence of GPU-friendly programming

Essence of GPU-friendly programming

- Minimise data movement
- Make data movement predictable (ideally linear scan)
- Minimise data-dependent branching
- Maximise parallelism (and avoid synchronisation between threads)
- Maximise lock-step parallelism (SIMD)

Essence of GPU-friendly programming

- Minimise data movement
-
-
-
-

Memory wall

Memory wall

Progress in hardware is unevenly distributed!

Memory wall

Progress in hardware is unevenly distributed!

Memory wall = growing disparity of speed between a processor reading from (off-chip) memory, and the response time of memory

Memory wall

Progress in hardware is unevenly distributed!

Memory wall = growing disparity of speed between a processor reading from (off-chip) memory, and the response time of memory

Processor speed has been increasing faster than DRAM speeds

Memory wall

Progress in hardware is unevenly distributed!

Memory wall = growing disparity of speed between a processor reading from (off-chip) memory, and the response time of memory

Processor speed has been increasing faster than DRAM speeds

When the processor's speed outpaces the rate at which data can be transferred to and from the memory system, a **sequential** processor must wait for the data to be fetched from memory, which slows down its performance and limits its speed.

Compute intensity (CI)

$$CI = \frac{\#\text{machine ops per second}}{\#\text{bytes per second from DRAM to processor}}$$

Compute intensity (CI)

$$CI = \frac{\#\text{machine ops per second}}{\#\text{bytes per second from DRAM to processor}}$$

CI $\approx$ number of (sequential) computations performed on word from DRAM, before next load arrives

Compute intensity (CI)

$$CI = \frac{\#\text{machine ops per second}}{\#\text{bytes per second from DRAM to processor}}$$

CI $\approx$ number of (sequential) computations performed on word from DRAM, before next load arrives

CI is the amount of compute device needs to do to avoid being stalled by DRAM.

Compute intensity (CI) of modern 2021 CPU

Compute intensity (CI) of modern 2021 CPU

- DRAM to CPU: 200 GBytes/second (= 25 Giga-FP64)
- CPU can compute 2000 GFLOPS FP64 per second (2 TFlops).

Compute intensity (CI) of modern 2021 CPU

- DRAM to CPU: 200 GBytes/second (= 25 Giga-FP64)
- CPU can compute 2000 GFLOPS FP64 per second (2 TFlops).

So to max out the CPU: required compute intensity

$$\frac{FLOPs}{Data\ rate} = \frac{2000}{25} = 80 \text{ compute instructions}$$

Compute intensity (CI)

```
/* gcc -Wall -Wextra -std=c99 -c daxpy.c */

void daxpy(int n, double alpha, double* x, double* y) {
  for(int i = 0; i < n; i++) {
    y[i] = alpha * x[i] + y[i];
  }
}
```

https://godbolt.org/ https://godbolt.org/z/ab737hKqq

Compute intensity (CI)

	NVIDIA A100	Intel Xeon 8280	AMD Rome 7742
Peak FP64 GigaFLOPs	19500	2190	2300
Memory B/W (GB/sec)	1555	131	204
Compute Intensity	100	134	90

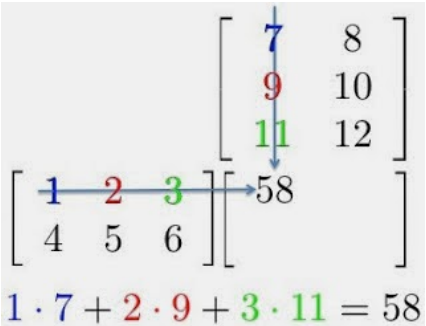
Nvidia A100 is 2020’s micro-architecture

Aside: compute intensity of matrix ops

What happens to compute intensity of matmul as matrix gets bigger?

Aside: compute intensity of matrix ops

What happens to compute intensity of matmul as matrix gets bigger?



Reason for memory wall (highly simplified)

Reason for memory wall (highly simplified)

Processor: 3 Ghz

Reason for memory wall (highly simplified)

Processor: 3 Ghz

Light speed: 300 000 km/s

Reason for memory wall (highly simplified)

Processor: 3 Ghz

Light speed: 300 000 km/s

Electron speed in silicon: 60 000 km/s

Reason for memory wall (highly simplified)

Processor: 3 Ghz

Light speed: 300 000 km/s

Electron speed in silicon: 60 000 km/s

Best case: electrons can move $\approx$ 2cm in a clock cycle!

Reason for memory wall (highly simplified)

Processor: 3 Ghz

Light speed: 300 000 km/s

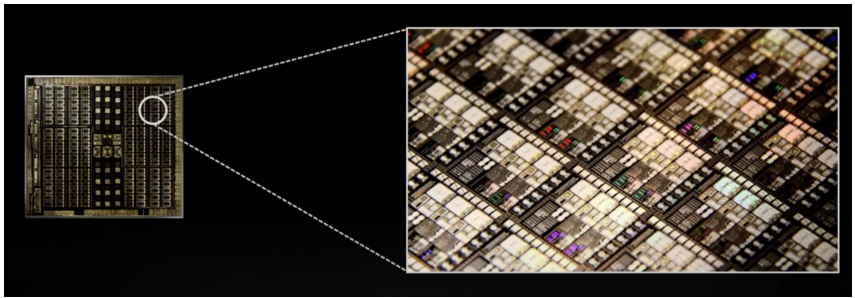
Electron speed in silicon: 60 000 km/s

Best case: electrons can move $\approx$ 2cm in a clock cycle!

But it's worse ...

Reason for memory wall (highly simplified)

Memory read is not in vacuum, but across a lot of transistors (100s - 1000s), which pass on information at each clock-switch ... so much slower.



Note: clock speeds are unlikely to go up much in future. (Energy / heat)

Memory bandwidth usage example

Another problem with naive programming: leaving memory bandwidth unused

Memory bandwidth usage example

Another problem with naive programming: leaving memory bandwidth unused

```
/* gcc -Wall -Wextra -std=c99 -c daxpy.c */

void daxpy(int n, double alpha, double* x, double* y) {
  for(int i = 0; i < n; i++) {
    y[i] = alpha * x[i] + y[i];
  }
}
```

	NVIDIA A100	AMD Rome 7742	Intel Xeon 8280
Memory B/W (GB/sec)	1555	204	131
DRAM Latency (ns)	404	122	89
Peak bytes per latency	628,220	24,888	11,659
Memory efficiency	0.0025%	0.064%	0.14%

Classic approach to masking memory latency

Classic approach to masking memory latency

Old problem:

1. INTRODUCTION

The year is 1991. Researchers at NASA Ames Research Center and at the University of Virginia's Institute for Parallel Computation (also supported by NASA) are trying to understand the performance of scientific computations on Intel's iPSC/860 hypercube, the state of the art in message-passing multicomputer. Lee [6] at NASA and Moyer [14] at UVa find that performance of even highly optimized, hand-coded scientific kernels can be up to orders of magnitude below peak. The reason: an imbalance between memory bandwidth and processor speed in the iPSC/860 node architecture.

---

*This work is based on an earlier work: *Hitting the Memory Wall: Implications of the Obvious*, in ACM SIGARCH Computer Architecture News, 23, ISSN:0163-5964 , March 1995 ACM, 1995. http://doi.acm.org/10.1145/216585.216588

Classic approach to masking memory latency

Parallelism (more precisely concurrency):

Classic approach to masking memory latency

Parallelism (more precisely concurrency): whenever thread waits for memory load, swap it out for a thread that is ready to compute. Swap back in when memory read arrives.

Classic approach to masking memory latency

Parallelism (more precisely concurrency): whenever thread waits for memory load, swap it out for a thread that is ready to compute. Swap back in when memory read arrives.

Several problems:

1. Cost of context switch
2. Number of threads the system can support
3. Number of 'cores'
4. Does the application have enough parallelism?

Modern GPUs solve (1, 2, 3).

Classic approach to masking memory latency

Parallelism (more precisely concurrency): whenever thread waits for memory load, swap it out for a thread that is ready to compute. Swap back in when memory read arrives.

Several problems:

1. Cost of context switch
2. Number of threads the system can support
3. Number of 'cores'
4. Does the application have enough parallelism?

Modern GPUs solve (1, 2, 3). Research question boils down to:

Classic approach to masking memory latency

Parallelism (more precisely concurrency): whenever thread waits for memory load, swap it out for a thread that is ready to compute. Swap back in when memory read arrives.

Several problems:

1. Cost of context switch
2. Number of threads the system can support
3. Number of 'cores'
4. Does the application have enough parallelism?

Modern GPUs solve (1, 2, 3). Research question boils down to: **do formal verification workloads have enough parallelism?**

Cost of context switch

On CPUs, context switches are extremely expensive, because thread state needs to be saved (e.g. 32 registers) to, and read from DRAM.

Cost of context switch

On CPUs, context switches are extremely expensive, because thread state needs to be saved (e.g. 32 registers) to, and read from DRAM.

Nvidia: 65536 registers per SM. Maximum registers per thread is 255. Cost of context switch:

Cost of context switch

On CPUs, context switches are extremely expensive, because thread state needs to be saved (e.g. 32 registers) to, and read from DRAM.

Nvidia: 65536 registers per SM. Maximum registers per thread is 255. Cost of context switch: **1 cycle**.

Number of threads the system can support

Nvidia A100 108 SMs ($\approx$ cores) each of which can carry 2048 threads. So 2048x108 = 221 184 threads.

Number of ’cores’

Number of ’cores’

What’s a core?

Number of ’cores’

What’s a core?

NVIDIA H100

The **NVIDIA H100 GPU** includes the following units:

- 7 or 8 GPCs, 57 TPCs, 2 SMs/TPC, 114 SMs per GPU
- 128 FP32 CUDA Cores/SM, 14592 FP32 CUDA Cores per GPU
- 4 4th-generation Tensor Cores per SM, 456 per GPU

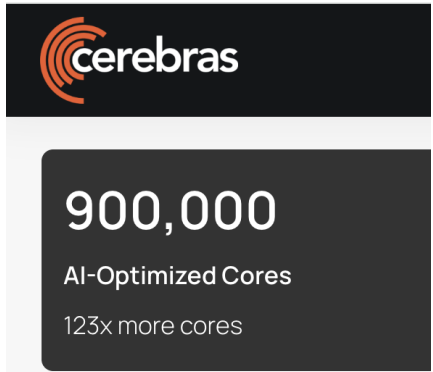
Number of ’cores’

What’s a core?

NVIDIA H100

The **NVIDIA H100 GPU** includes the following units:

- 7 or 8 GPCs, 57 TPCs, 2 SMs/TPC, 114 SMs per GPU
- 128 FP32 CUDA Cores/SM, 14592 FP32 CUDA Cores per GPU
- 4 4th-generation Tensor Cores per SM, 456 per GPU



Does the application have enough parallelism?

Does the application have enough parallelism?

Does the application have enough good parallelism? (lock-step / SIMD)

Problem with GPU programming

Problem with GPU programming

Hard

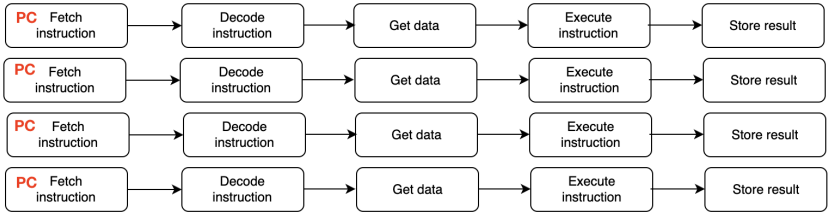
Summary: what makes program GPU-friendly?

- Minimise data movement
- Make data movement predictable (ideally linear scan)
- Minimise data-dependent branching
- Maximise parallelism (and avoid synchronisation between threads)
- Maximise lock-step parallelism (SIMD)

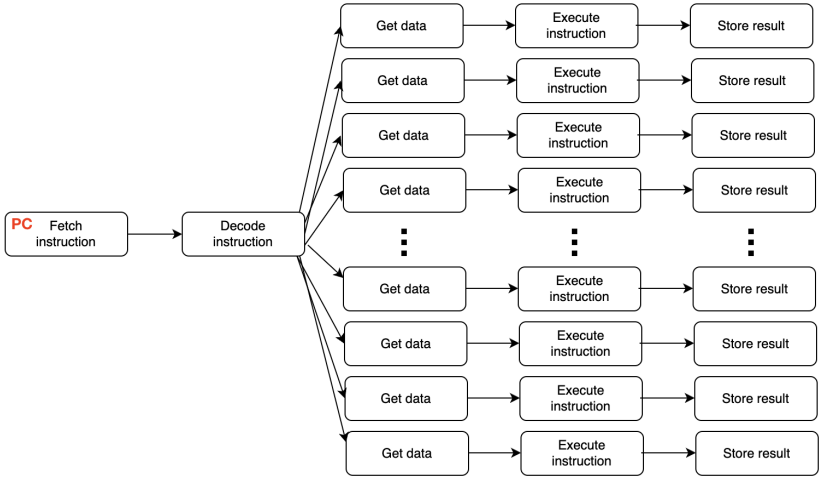
Summary: what makes program GPU-friendly?

- Maximise lock-step parallelism (SIMD)

CPU (highly idealised)



GPU (highly idealised)



Why no verification workloads on GPUs today?

Widely held belief that verification related workloads are intrinsically sequential, and branching intensive

Why no verification workloads on GPUs today?

Many attempts (e.g. annual SAT competition has / had “parallel” track)

Experience that taking a branch-heavy algorithm, optimised for CPUs and slap it on a GPU will probably make it slower.

Why no verification workloads on GPUs today?

Many attempts (e.g. annual SAT competition has / had “parallel” track)

Experience that taking a branch-heavy algorithm, optimised for CPUs and slap it on a GPU will probably make it slower.

Philosophical problem

Why no verification workloads on GPUs today? Wrong cost model

WIKIPEDIA
The Free Encyclopedia

Search Wikipedia | Search

Time complexity

ontents hide

Top)

able of common time omplexities

onstant time

ogarithmic time

olylogarithmic time

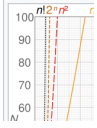
ub-linear time

Article Talk Read Edit

From Wikipedia, the free encyclopedia

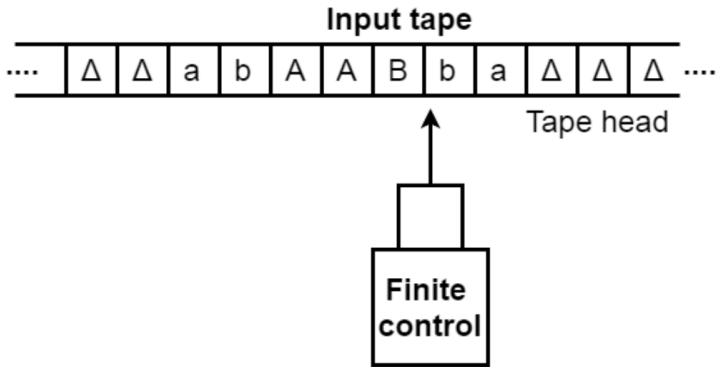
"Running time" redirects here. For the film, see Running Time (film).

In theoretical computer science, the **time complexity** is the computational complexity that describes the amount of computer time it takes to run an algorithm. Time complexity is commonly estimated by counting the number of



Why no verification workloads on GPUs today? Wrong cost mode

I



Remember the mental model: compute is free, data-movement is expensive

Why no verification workloads on GPUs today? Wrong cost model

Many interlocking reasons, the most important is **cost model** for GPUs is quite different from that for CPUs.

Why no verification workloads on GPUs today? Wrong cost model

Many interlocking reasons, the most important is **cost model** for GPUs is quite different from that for CPUs.

	Old cost model (CPUs)	**New cost model (GPUs)**
Cheap/free	data movement, data-dependent branching	compute steps, lock-step (SIMD) parallelism
Expensive	compute steps, ~~parallelism~~, ~~thread-synchronisation~~	data movement, data-dependent branching, thread-synchronisation

To this day, the verification community has been optimising against the old cost model.

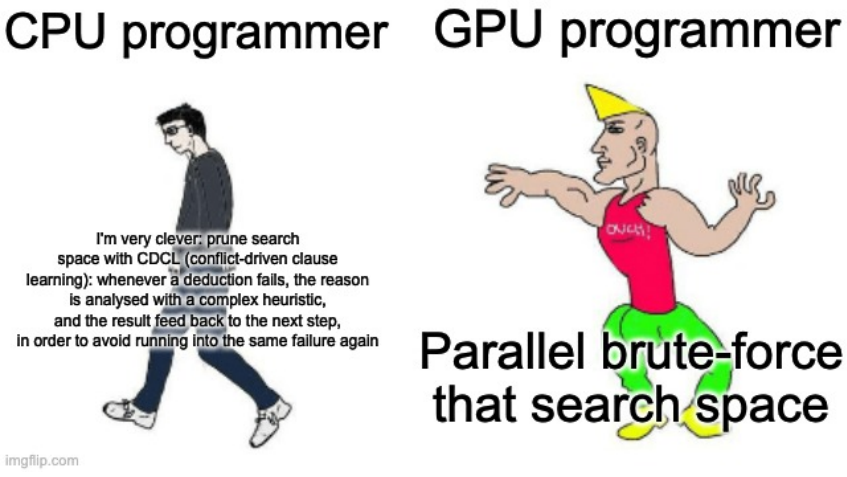
The positive side of combinatorial exposion

The positive side of combinatorial exposion

Example: SAT solving, n variables, 2^n possible models ...

The positive side of combinatorial exposion

Example: SAT solving, n variables, 2^n possible models ...



The positive side of combinatorial exposion. Example: SAT solving

NP-complete. Probably $O(2^n)$

The positive side of combinatorial exposion. Example: SAT solving

NP-complete. Probably $O(2^n)$

In the old cost model: prune the search space as hard and early as possible.

The positive side of combinatorial exposion. Example: SAT solving

NP-complete. Probably $O(2^n)$

In the old cost model: prune the search space as hard and early as possible.

CDCL (conflict-driven clause learning):

The positive side of combinatorial exposion. Example: SAT solving

NP-complete. Probably $O(2^n)$

In the old cost model: prune the search space as hard and early as possible.

CDCL (conflict-driven clause learning): whenever a deduction fails, the reason is analysed with a complex heuristic, and the result feed back to the next step, in order to avoid running into the same failure again. This is extremely effective at minimising the cost of SAT-solving under the old cost model: solving industrial benchmarks with millions of variables.

The positive side of combinatorial exposion. Example: SAT solving

NP-complete. Probably $O(2^n)$

In the old cost model: prune the search space as hard and early as possible.

CDCL (conflict-driven clause learning): whenever a deduction fails, the reason is analysed with a complex heuristic, and the result feed back to the next step, in order to avoid running into the same failure again. This is extremely effective at minimising the cost of SAT-solving under the old cost model: solving industrial benchmarks with millions of variables.

But ... sequentialises search, i.e., very expensive in the new cost model.

The positive side of combinatorial exposion. Example: SAT solving

NP-complete. Probably $O(2^n)$

In the old cost model: prune the search space as hard and early as possible.

CDCL (conflict-driven clause learning): whenever a deduction fails, the reason is analysed with a complex heuristic, and the result feed back to the next step, in order to avoid running into the same failure again. This is extremely effective at minimising the cost of SAT-solving under the old cost model: solving industrial benchmarks with millions of variables.

But ... sequentialises search, i.e., very expensive in the new cost model.

What would GPU do?

The positive side of combinatorial exposion. Example: SAT solving

NP-complete. Probably $O(2^n)$

In the old cost model: prune the search space as hard and early as possible.

CDCL (conflict-driven clause learning): whenever a deduction fails, the reason is analysed with a complex heuristic, and the result feed back to the next step, in order to avoid running into the same failure again. This is extremely effective at minimising the cost of SAT-solving under the old cost model: solving industrial benchmarks with millions of variables.

But ... sequentialises search, i.e., very expensive in the new cost model.

What would GPU do? **Speculate and let it fail**

The positive side of combinatorial exposion. Example: SAT solving

NP-complete. Probably $O(2^n)$

In the old cost model: prune the search space as hard and early as possible.

CDCL (conflict-driven clause learning): whenever a deduction fails, the reason is analysed with a complex heuristic, and the result feed back to the next step, in order to avoid running into the same failure again. This is extremely effective at minimising the cost of SAT-solving under the old cost model: solving industrial benchmarks with millions of variables.

But ... sequentialises search, i.e., very expensive in the new cost model.

What would GPU do? **Speculate and let it fail**

It might be faster to run into the same failure again millions of times in parallel, just to avoid this sequentialisation.

The positive side of combinatorial exposion. Example: SAT solving

Summary: if we want to make SAT solving fast on GPUs, we probably have to rethink the algorithm from scratch!

Example: union-find (UF), aka disjoint-sets

UF is example why even the simplest of algorithms cannot easily be ported from CPUs to GPUs without loosing performance and needs reconceptualisation, especially if the algorithm is highly optimised against the old cost model.

Example: union-find (UF), aka disjoint-sets

UF is example why even the simplest of algorithms cannot easily be ported from CPUs to GPUs without loosing performance and needs reconceptualisation, especially if the algorithm is highly optimised against the old cost model.

Equality over terms is one of the most fundamental relations in formal reasoning, and any automated reasoning tool needs to account for reasoning techniques involving equality. UF is a data structure that is often used for handling equality in formal reasoning because of its excellent run-time cost (in the old cost model for CPUs).

Example: union-find

```
MAKE-SET(x)
1  x.p = x
2  x.rank = 0

UNION(x, y)
1  LINK(FIND-SET(x), FIND-SET(y))

LINK(x, y)
1  if x.rank > y.rank
2       y.p = x
3  else x.p = y
4       if x.rank == y.rank
5            y.rank = y.rank + 1

FIND-SET(x)
1  if x ≠ x.p                    // not the root?
2       x.p = FIND-SET(x.p)      // the root becomes the parent
3  return x.p                    // return the root
```

Path compression

Example: union-find

```
MAKE-SET(x)
1  x.p = x
2  x.rank = 0

UNION(x, y)
1  LINK(FIND-SET(x), FIND-SET(y))

LINK(x, y)
1  if x.rank > y.rank
2        y.p = x
3  else x.p = y
4        if x.rank == y.rank
5              y.rank = y.rank + 1

FIND-SET(x)
1  if x ≠ x.p
2     return FIND-SET(x.p)
3  return x.p
```

No path compression

Example: union-find

fast scalable and machine-verified multicore disjoint set union data structures and their wide deployment in parallel algorithms – jayanti tarjan.pdf

Where to start?

Has my talk so far been all speculation about could be done?

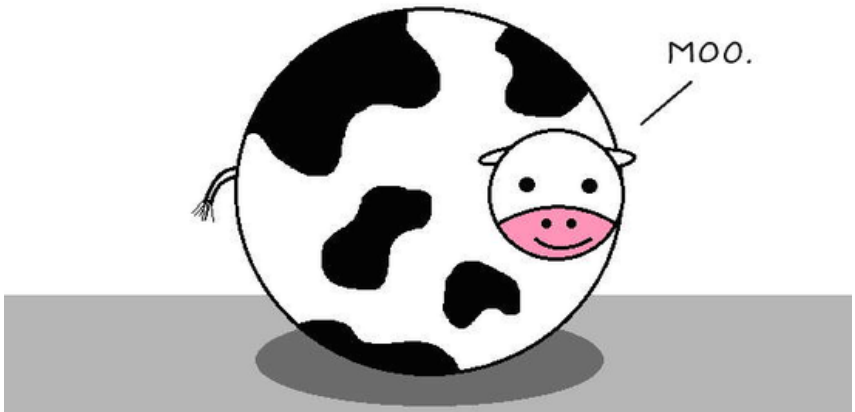
Where to start?

Has my talk so far been all speculation about could be done?

Lots of research on SAT-solving on GPUs. Not very successful, as far as I am aware.

Maybe verification workloads are intrinsically sequential

We need a simple success story to get started. We need ...


MOO.

Program synthesis

Program synthesis

Algorithmic generation of syntactic entities from specifications.

Dominant flavours:

- PBE (= programming by example), where the input is a set of examples
- PBF (= programming by formula), where the input is a logical formula

Program synthesis

INFORMATION AND CONTROL 10, 447–474 (1967)

Language Identification in the Limit

E MARK GOLD*

The RAND Corporation

Language learnability has been investigated. This refers to the following situation: A class of possible languages is specified, together with a method of presenting information to the learner about an unknown language, which is to be chosen from the class. The question is now asked, "Is the information sufficient to determine which of the

Unsolved with neural networks

Program synthesis

LOGIC, ARITHMETIC, AND AUTOMATA(1)

By ALONZO CHURCH

This paper is a summary of recent work in the application of mathematical logic to finite automata, and especially of mathematical logic beyond propositional calculus.

To begin with a sketch of the history of the matter, let us recall that application of the "algebra of logic", i.e., elementary Boolean algebra, to the analysis of switching circuits was first suggested by Ehrenfest [A]. Nothing came of Ehrenfest's remark for many years, and it seems to have remained

Unsolved with neural networks

Program synthesis

Naive algorithm: bottom-up enumeration

```
def enumerate(spec):
    cost = 0
    while true:
        phi = next_formula(cost)
        if phi satisfies spec:
            return with phi
        cost += 1
```

aka generate-and-filter, guess-and-check, diagonalisation, ...!

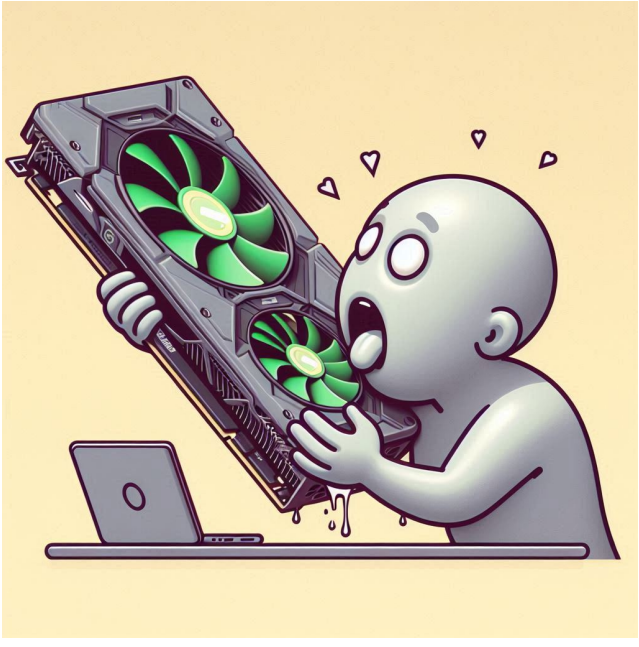
```
if enum(0) satisfies spec then return enum(0);

if enum(1) satisfies spec then return enum(1);

if enum(2) satisfies spec then return enum(2);

if enum(3) satisfies spec then return enum(3);

if enum(4) satisfies spec then return enum(4);

if enum(5) satisfies spec then return enum(5);

...
```

```
if enum(0) satisfies spec then return enum(0)
   PAR
if enum(1) satisfies spec then return enum(1)
   PAR
if enum(2) satisfies spec then return enum(2)
   PAR
if enum(3) satisfies spec then return enum(3)
   PAR
if enum(4) satisfies spec then return enum(4)
   PAR
if enum(5) satisfies spec then return enum(5)
   PAR
...
```

Embarrassingly parallel!

Embarrassingly parallel!

Remember $15048 = \infty, 900000 = \infty$



LTL_f -learning by example with program synthesis

(based on PLDI'23 and CAV'24 work)

LTL = linear temporal logic

Linear temporal logic (LTL) is widely used in industrial verification.

LTL is a modal logic for specifying properties of finite or infinite traces / strings.

LTL over finite traces (aka LTL_f) is (semantically) a strict subsystem of regular expressions ($\approx$ "aperiodic" regular expressions)

LTL, linear temporal logic

LTL formulae over $\Sigma = \{p_1, ..., p_n\}$ are given by the following grammar.

$$\phi \quad ::= \quad p_i \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \mathsf{X}\phi \mid \mathsf{F}\phi \mid \mathsf{G}\phi \mid \phi \,\mathsf{U}\, \phi$$

We assume a simple cost$(\cdot)$ function that gives the cost of each formula. (E.g. size)

LTL_f semantics: satisfaction relation

Let *tr* be a **finite** trace over powerset alphabet $\mathfrak{P}(\Sigma)$ and ϕ a formula.

$$tr, i \vDash \phi$$

LTL_f semantics: satisfaction relation

Let *tr* be a **finite** trace over powerset alphabet $\mathfrak{P}(\Sigma)$ and ϕ a formula.

$$tr, i \vDash \phi$$

- $tr, i \vDash p$ if $p \in tr(i)$
- ...
- $tr, i \vDash \mathsf{X}\phi$, if $tr, i+1 \vDash \phi$,
- $tr, i \vDash \mathsf{F}\phi$, if there is $i \leq j < \mathrm{len}(tr)$ with $tr, j \vDash \phi$,

LTL_f semantics: language of a formula

Each ϕ induces languages:

$$\mathrm{Lang}(\phi, i) = \{tr \mid tr, i \vDash \phi\} \qquad \mathrm{Lang}(\phi) = \mathrm{Lang}(\phi, 0)$$

Branch-free algorithms and data-structures: search space

We are going a (refined variant of) bottom-up enumeration, so each step needs to be **fast**!

Branch-free algorithms and data-structures: search space

Search space	**Representation**	**Data Structure**	**Issue**
Formula	Tree	Pointers	Slow, redundant
Language	$\Sigma^* \to \mathbb{B}$	–	Infinite
Language up to $P \cup N$	$(P \cup N) \to \mathbb{B}$	Bitvector	Non-compositional
Language up to $cl(P \cup N)$	$cl(P \cup N)) \to \mathbb{B}$	Bitvector	More space

Branch-free algorithms and data-structures: search space

Need to prove $\phi \vDash (P, N)$ so quotient formulae by equality up to $\mathrm{Lang}(\phi) \cap (P \cup N)$

Branch-free algorithms and data-structures: search space

Need to prove $\phi \vDash (P, N)$ so quotient formulae by equality up to $\mathrm{Lang}(\phi) \cap (P \cup N)$

$(P \cup N) \to \mathbb{B}$ is (isomorphic to) bitvector, assuming a fixed **total order** on $P \cup N$.

$$\mathbf{1}_L : \quad P \cup N \quad \to \quad \mathbb{B}$$

Branch-free algorithms and data-structures: search space

Need to prove $\phi \vDash (P, N)$ so quotient formulae by equality up to $\text{Lang}(\phi) \cap (P \cup N)$

$(P \cup N) \to \mathbb{B}$ is (isomorphic to) bitvector, assuming a fixed **total order** on $P \cup N$.

$$\mathbf{1}_L : \quad P \cup N \quad \to \quad \mathbb{B}$$

Positive = {1, 011, 1011, 11011}
Negative = {ϵ, 10, 101, 0011}

Bitvector representation of ϕ

For trace tr of length n satisfaction is isomorphic to bitvector bv of same length.

$$bv(i) = \begin{cases} 1 & tr, i \vDash \phi \\ 0 & \text{else} \end{cases}$$

Bitvector representation of ϕ

For trace *tr* of length *n* satisfaction is isomorphic to bitvector *bv* of same length.

$$bv(i) = \begin{cases} 1 & tr, i \vDash \phi \\ 0 & \text{else} \end{cases}$$

Example for *tr* = “squeegee” and atomic proposition *g*:

ϕ	*bv*
g	00000100
Xg	00001000
XXg	00010000
$XXXg$	00100000
$XXXXg$	01000000
$XXXXXg$	10000000
$XXXXXXg$	00000000

Bitvector representation of ϕ

For trace *tr* of length *n* satisfaction is isomorphic to bitvector *bv* of same length.

$$bv(i) = \begin{cases} 1 & tr, i \vDash \phi \\ 0 & \text{else} \end{cases}$$

Example for *tr* = “squeegee” and atomic proposition *g*:

ϕ	*bv*
g	00000100
Xg	00001000
XXg	00010000
$XXXg$	00100000
$XXXXg$	01000000
$XXXXXg$	10000000
$XXXXXXg$	00000000

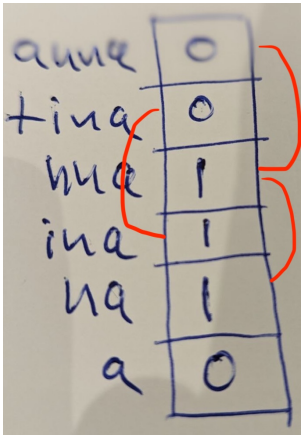
Note:

- Bitvector is suffix-closed
- Bitshifts only implements X branch-free
- Bitshifts are machine instructions
- Bitshifts assume locality

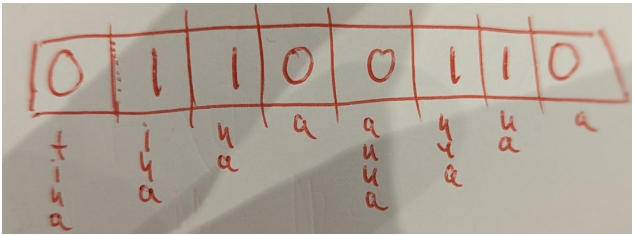
Bitvector representation of ϕ

Suffix closure of $P \cup N = \{anna, tina\}$ is $\{tina, ina, na, a, anna, nna\}$

Irredundant



Redundant



- Preserves locality
- Allows compositional construction of (representations of) LTL_f formulae
- Space/time trade-off

Branch-free X

Let *bv* represent formula ϕ. Want bitvector for Xϕ.

Branch-free X

Let *bv* represent formula ϕ. Want bitvector for Xϕ.

```
def branchfree_X(bv):
    return bv << 1
```

Branch-free X

Let *bv* represent formula ϕ. Want bitvector for Xϕ.

```
def branchfree_X(bv):
    return bv << 1
```

ϕ	*bv*
g	00000100
X*g*	00001000
XX*g*	00010000
XXX*g*	00100000
XXXX*g*	01000000
XXXXX*g*	10000000
XXXXXX*g*	00000000

Branch-free F

Let *bv* represent formula ϕ. Want bitvector for Fϕ.

Branch-free F

Let *bv* represent formula ϕ. Want bitvector for Fϕ.

```
def branchfree_F(bv):
    L = len(bv)
    for i in range(log(L)+1):
        bv |= bv << 2**i
    return bv
```

Branch-free F

Let *bv* represent formula ϕ. Want bitvector for Fϕ.

```
def branchfree_F(bv):
    L = len(bv)
    for i in range(log(L)+1):
        bv |= bv << 2**i
    return bv
```

```
def branchfree_F(bv):  // Assume length(bv) == 64
        bv |= bv << 1
        bv |= bv << 2
        bv |= bv << 4
        bv |= bv << 8
        bv |= bv << 16
        bv |= bv << 32
    return bv
```

Branch-free U

```
def branchfree_U(bv1, bv2):
    L = len(bv1)
    for i in range(log(L)+1):
        bv2 |= bv1 & (bv2 << 2**i)
        bv1 &= bv1 << 2**i
    return bv2
```

```
def branchfree_U(bv1, bv2):  // Assume length(bv1) == 64
       bv2 |= bv1 & (bv2 << 1)
       bv1 &= bv1 << 1
       bv2 |= bv1 & (bv2 << 2)
       bv1 &= bv1 << 2
       bv2 |= bv1 & (bv2 << 4)
       bv1 &= bv1 << 4
       bv2 |= bv1 & (bv2 << 8)
       bv1 &= bv1 << 8
       bv2 |= bv1 & (bv2 << 16)
       bv1 &= bv1 << 16
       bv2 |= bv1 & (bv2 << 32)
   return bv2
```

Complexity

Theorem

Algorithm implements the LTL_f semantics branch-free in $O(\log n)$ time (n trace length), assuming bitwise boolean operations and shifts by powers of 2 have cost 1.

Previous implementations are $O(n^2)$ or worse

Main loop (1)

```
language_cache = []

def enum(p, n, cost):
    if (p, n) can be solved with Atom then return Atom
    language_cache.append([Atom])
    for c in range(cost(Atom)+1, cost(overfit(p, n))):
        language_cache.append([])
        for op in [F, U, G, X, And, Or, Not]:
            handleOp(op, p, n, c, cost)
    return overfit(p, n)
```

Main loop (2)

```
def handleOp(op, p, n, c, cost):
   match op:
      case F:
         for all phi in language_cache(c-cost(F)):
            phi_new = branchfree_F(phi)
            if phi_new |= (p, n): then exit(phi_new)
            if phi_new is unique in language_cache:
               language_cache[c].append(phi_new)
      case U:
            ...
```

CI and combinatorial explosion

Let’s analyse this.

CI and combinatorial explosion

Let's analyse this. Consider the synthesis of logical conjunctions of cost K.

CI and combinatorial explosion

Let's analyse this. Consider the synthesis of logical conjunctions of cost K. Let's say we fetch ϕ_1, ..., ϕ_n and ψ (from the language cache).

CI and combinatorial explosion

Let's analyse this. Consider the synthesis of logical conjunctions of cost K. Let's say we fetch ϕ_1, ..., ϕ_n and ψ (from the language cache).

$$\phi_1 \quad \wedge \quad \psi$$

CI and combinatorial explosion

Let's analyse this. Consider the synthesis of logical conjunctions of cost K. Let's say we fetch ϕ_1, ..., ϕ_n and ψ (from the language cache).

$$\begin{array}{lll} \phi_1 & \wedge & \psi \\ \phi_2 & \wedge & \psi \end{array}$$

CI and combinatorial explosion

Let's analyse this. Consider the synthesis of logical conjunctions of cost K. Let's say we fetch ϕ_1, ..., ϕ_n and ψ (from the language cache).

$$\begin{array}{lll} \phi_1 & \wedge & \psi \\ \phi_2 & \wedge & \psi \\ \phi_3 & \wedge & \psi \end{array}$$

CI and combinatorial explosion

Let's analyse this. Consider the synthesis of logical conjunctions of cost K. Let's say we fetch ϕ_1, ..., ϕ_n and ψ (from the language cache).

$$\begin{array}{lll} \phi_1 & \wedge & \psi \\ \phi_2 & \wedge & \psi \\ \phi_3 & \wedge & \psi \\ & \vdots & \end{array}$$

CI and combinatorial explosion

Let's analyse this. Consider the synthesis of logical conjunctions of cost K. Let's say we fetch ϕ_1, ..., ϕ_n and ψ (from the language cache).

$$
\begin{array}{lcl}
\phi_1 & \wedge & \psi \\
\phi_2 & \wedge & \psi \\
\phi_3 & \wedge & \psi \\
 & \vdots & \\
\phi_n & \wedge & \psi
\end{array}
$$

CI and combinatorial explosion

Let's analyse this. Consider the synthesis of logical conjunctions of cost K. Let's say we fetch ϕ_1, ..., ϕ_n and ψ (from the language cache).

$$\begin{array}{lcl} \phi_1 & \wedge & \psi \\ \phi_2 & \wedge & \psi \\ \phi_3 & \wedge & \psi \\ & \vdots & \\ \phi_n & \wedge & \psi \end{array}$$

Since we need to fetch ψ only once, the compute intensity increases as n grows. Like with matrix multiplication.

CI and combinatorial explosion

Let's analyse this. Consider the synthesis of logical conjunctions of cost K. Let's say we fetch ϕ_1, ..., ϕ_n and ψ (from the language cache).

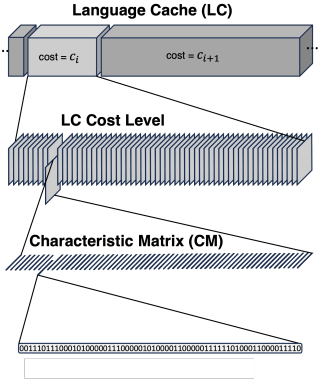
$$\begin{array}{lcl} \phi_1 & \wedge & \psi \\ \phi_2 & \wedge & \psi \\ \phi_3 & \wedge & \psi \\ & \vdots & \\ \phi_n & \wedge & \psi \end{array}$$

Since we need to fetch ψ only once, the compute intensity increases as n grows. Like with matrix multiplication.

As K grows bigger, n grows exponentially ...

This is one of the advantages of combinatorial explostion. (Another being unlimited parallelism.)

Redundancies of syntax



We cache bitvectors (representing formulae) in a (read-only) language cache. Why?

Problem: LTL_f operators don't reserve uniqueness.

If bv_1 represents ϕ and bv_2 represents ψ, both are unique, i.e., have not been seen before, then it is **not** guaranteed that the bitvector representing $\phi \,\mathsf{U}\, \psi$ is unique.

Uniqueness check of newly constructed (representation of) formula. Using fast hashing library. **Most expensive part of search.**

Density conjecture

Conjecture

Density of unique formulae among all formulae is 0

Explosive growth of language cache main scaling limit. Two solutions

- Relaxed uniqueness check
- Divide-and-conquer

Relaxed uniqueness

(Pseudo-)Randomly reject **unique** representations of formulae from language cache

Relaxed uniqueness

(Pseudo-)Randomly reject **unique** representations of formulae from language cache

This is sound: if we find ϕ that satisfies (P, N) we are done

But might increase size of returned formula.

Divide & conquer

Relaxed uniqueness checks, and bitvector representation are not enough to improve on our scalability issue w.r.t. memory.

Divide & conquer

If (P, N) is too big, split (P, N), into disjoint (P_i, N_j) for $i, j = 1, 2$, such that

$$P = P_1 \cup P_2 \qquad N = N_1 \cup N_2$$

Learn recursively:

- $\phi_{11} = \text{synth}(P_1, N_1)$
- $\phi_{12} = \text{synth}(P_1, N_2)$
- $\phi_{21} = \text{synth}(P_2, N_1)$
- $\phi_{22} = \text{synth}(P_2, N_2)$

Combine all into $(\phi_{11} \wedge \phi_{12}) \vee (\phi_{21} \wedge \phi_{22})$

Not guaranteed to be minimal

Divide & conquer

Interesting variant: probabilistic sampling from (P, N) effective.

Two sources of losing minimality

- Divide & conquer
- Relaxed uniqueness checks

For small (P, N) we don't need those and our algorithm learns minimal formula.

Benchmarks

Existing benchmarks are too easy, essentially all solved within measurement threshold.

We made various new benchmarks. Please use them.

Comparison with SOTA (Scarlet)

(# P, # N)	Our impl. Time (Cost)	Scarlet Time (Cost)
$(2^3, 2^3)$	0.31s (12)	1532.85s (19)
$(2^4, 2^4)$	0.32s (12)	1463.67s (17)
$(2^5, 2^5)$	0.36s (12)	2867.47s (17)
$(2^6, 2^6)$	0.34s (12)	5691.98s (17)
$(2^7, 2^7)$	0.63s (20)	OOM
$(2^8, 2^8)$	0.95s (19)	OOM
$(2^9, 2^9)$	0.72s (19)	OOM
$(2^{10}, 2^{10})$	1.09s (19)	OOM
$(2^{11}, 2^{11})$	1.32s (19)	OOM
$(2^{12}, 2^{12})$	1.66s (19)	OOM
$(2^{13}, 2^{13})$	2.46s (19)	OOM
$(2^{14}, 2^{14})$	4.62s (20)	OOM
$(2^{15}, 2^{15})$	8.35s (19)	OOM
$(2^{16}, 2^{16})$	15.52s (19)	OOM
$(2^{17}, 2^{17})$	30.49s (19)	OOM

Future

In the future the performance gap between CPUs and GPUs will grow!

Programming will become more and more GPU-like.

If you can parallelise verification workloads, so they were well on a single GPU, then it **probably** also scales well to clusters of GPUs.

Need new complexity theory that is predictive for GPUs.

I would love to collaborate!

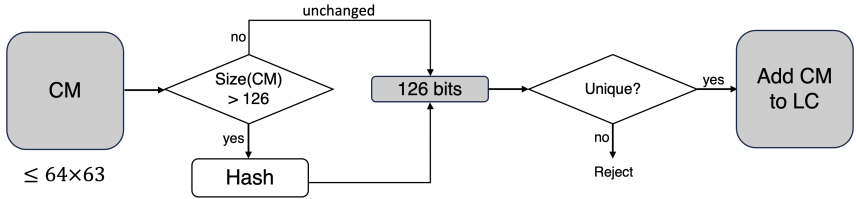
Papers:

- REI: `https://arxiv.org/abs/2305.18575`, code and data: `https://github.com/MojtabaValizadeh/paresy`
- LTLI: `https://arxiv.org/abs/2402.12373`, code and data: `https://github.com/MojtabaValizadeh/ltl-learning-on-gpus`

Please contact me for questions `contact@martinfriedrichberger.net`

Thank you!

Relaxed uniqueness



We implement (pseudo-)random decision by hashing:

We check for uniqueness not using full bitvectors, but bitvectors hashed to k bits.

Choice of $k = 126$ bits is pragmatic, in experiments there is **unexplained** phase transition at around 70 bits.

If size of bitvector is ≤ 126 then our LTL_f -learner is precise: returns minimal formula.

Density conjecture

Fix an enumeration $\#$ of all LTL_f formulae (resp. aperiodic languages) over Σ.

Definition

$L = \#(n)$ is **unique** if for all $i < n$, $L \neq \#(i)$. ϕ is **unique** if ϕ's language is unique.

Conjecture

In the limit the density of unique formulae among all formulae is 0:

$$\lim_{n\to\infty} \frac{\#\{\phi \mid \mathrm{cost}(\phi) \star n, \phi \text{ unique}\}}{\#\{\phi \mid \mathrm{cost}(\phi) < n\}} = 0$$

where $\star$ ranges over $=, <$. (Mutatis mutandis for aperiodic languages)

Density conjecture

Fix an enumeration $\#$ of all LTL_f formulae (resp. aperiodic languages) over Σ.

Definition

$L = \#(n)$ is **unique** if for all $i < n$, $L \neq \#(i)$. ϕ is **unique** if ϕ's language is unique.

Conjecture

In the limit the density of unique formulae among all formulae is 0:

$$\lim_{n\to\infty} \frac{\#\{\phi \mid \mathrm{cost}(\phi) \star n, \phi \text{ unique}\}}{\#\{\phi \mid \mathrm{cost}(\phi) < n\}} = 0$$

where $\star$ ranges over $=, <$. (Mutatis mutandis for aperiodic languages)

I doubt this depends on the chosen notion of cost / enumeration either.

Noisy-learning conjecture

Conjecture

If we learn with an allowed ϵ fraction of misclassified strings from (P, N), then learning becomes easier in a way that is exponential in ϵ.

Here easier means: the size of the bottom-up construction of formulae, before the first solution is hit, shrinks.

- L. Pitt, M. K. Warmuth, The minimum consistent DFA problem cannot be approximated within any polynomial. (1989)
- M. Kearns, L. Valiant, Cryptographic Limitations on Learning Boolean Formulae and Finite Automata. (1994)

Density conjecture

Fix an enumeration $\#$ of all LTL_f formulae (resp. aperiodic languages) over Σ.

Definition

$L = \#(n)$ is **unique** if for all $i < n$, $L \neq \#(i)$. ϕ is **unique** if ϕ's language is unique.

Conjecture

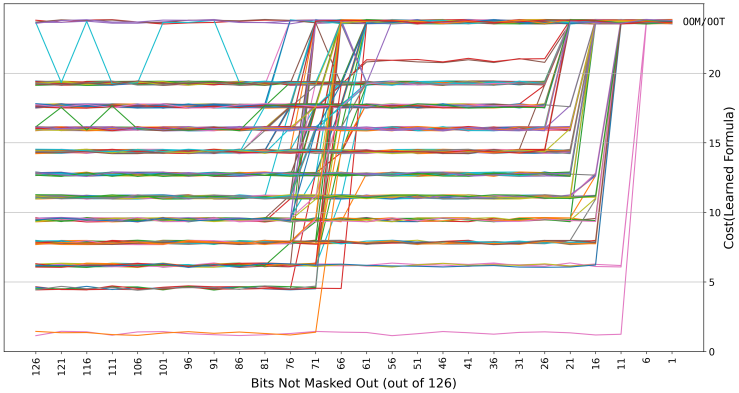
In the limit the density of unique formulae among all formulae is 0:

$$\lim_{n \to \infty} \frac{\#\{\phi \mid \mathrm{cost}(\phi) \star n, \phi \text{ unique}\}}{\#\{\phi \mid \mathrm{cost}(\phi) < n\}} = 0$$

where $\star$ ranges over $=, <$. (Mutatis mutandis for aperiodic languages)

Phase transition conjecture

Recall: relaxed uniqueness checks map candidate to k bits. The smaller the k the bigger the resulting learned formula. Experiment:



Conjecture

This is a phase transition.