Fast global optimization on the GPU

David P. Sanders

Faculty of Sciences, National University of Mexico (UNAM)
& Department of Mathematics, MIT

Joint work with Alan Edelman & Valentin Churavy, MIT
Luis Benet, UNAM

We show how to write generic code for global optimization of non-convex functions using interval arithmetic, which runs on both the CPU and the GPU.

Interval arithmetic

- Idea: Calculate the range of a function f over a set X:
  Define $f(X) := \{f(x) : x \in X\}$

- Standard numerical methods cannot do this

```
struct Interval
    lo::Float64
    hi::Float64
end

Base.exp(x::Interval) =
Interval(exp(x.lo), exp(x.hi))
```

Motivation: Structure of atomic clusters and proteins

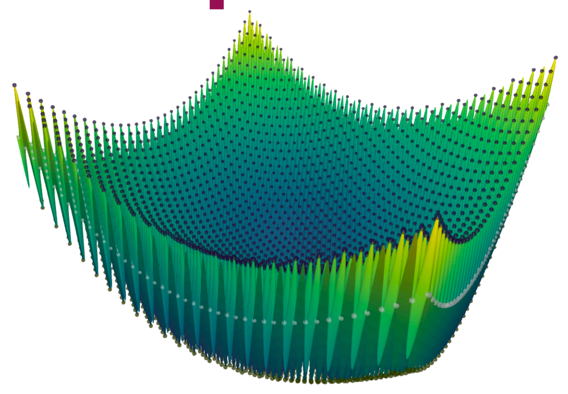
- N atoms at positions $\mathbf{r}_i$

- Potential energy landscape in $3N$ dimensions:

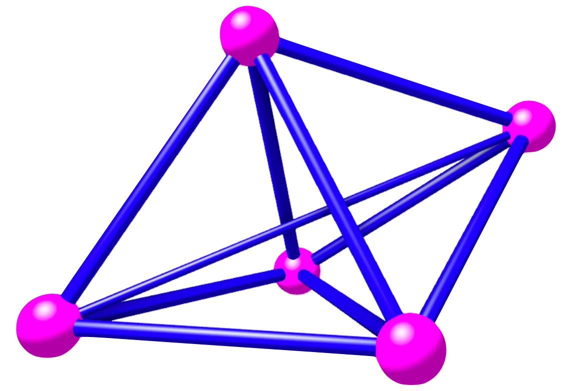
$$V_N(\mathbf{r}) := \sum_{i<j} u\left(\|\mathbf{r}_i - \mathbf{r}_j\|\right)$$

- Want to find *all* local minima and saddle points
  i.e. roots (zeros) of ∇V

- And global minimum (ground state)



model energy landscape



Lennard-Jones cluster ground state

$u(r) = 4(r^{-12} - r^{-6})$

Interval arithmetic can be quite fast

- Compare evaluation speed of Lennard-Jones potential V_5:

  - $V_5(X)$ vector of 9 intervals
    vs $V_5(m(X))$ vector of 9 `Float64s`

  - Only ~5x penalty (with fast directed rounding)

  - We gain much more information: rigorous bound on range

NVIDIA V100 GPU (~5000 cores): 450x speed-up

To vectorise, or not to vectorise…?

- Languages like Python and MATLAB demand that you **vectorise** your code
- To get around limitations of the tool due to **interpreting** code
- GPUs also require that you **vectorise** your code
- Due to the internal structure of the processor
- Single instruction, many data in parallel (**SIMD**)

Parallel branch and bound on GPU

- Need to discard some boxes
- E.g. `v[inf.(v) .<= m]`
- Not obvious how to do in parallel on GPU
- Solution: `cumsum` on boolean array

Parallelisation on the GPU

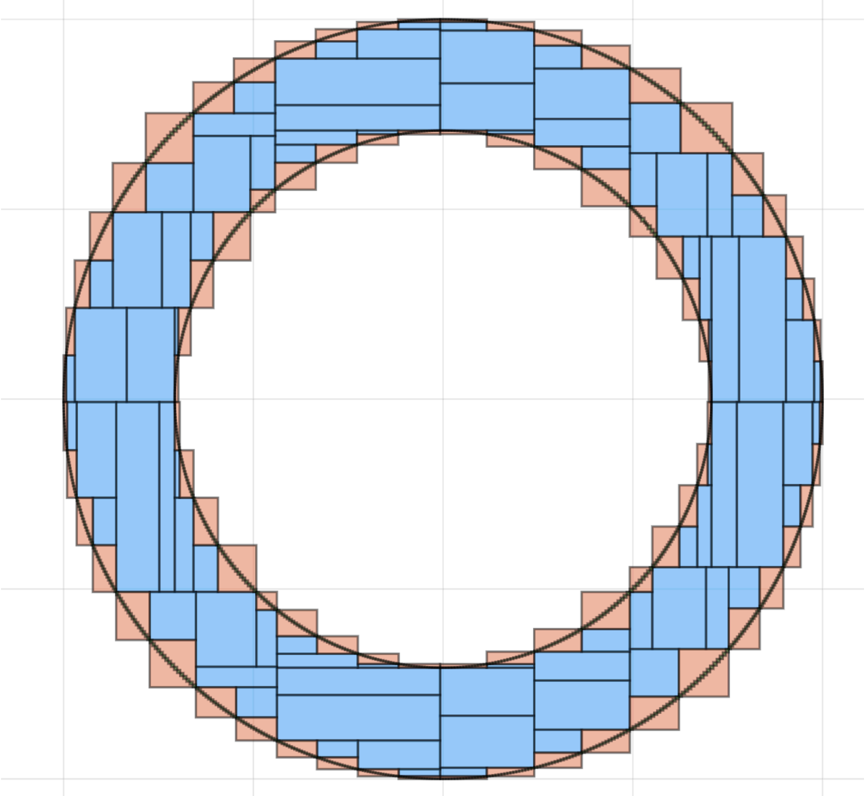
- GPU (Graphics Processing Unit):
  - Run same code in parallel on many cores, with different data (SIMD)
  - 100,000s of lightweight threads
- Compile specific parts to the GPU
- Vectorized element-wise broadcast works on GPU: `f.(v)`

Performance

- Current implementation: ~80x speed-up compared to single CPU core
- Factor of 5 slower than just interval arithmetic
- Probably due to excessive array allocation

(Spatial) branch and bound

- Global exhaustive search using bisection to find all global minimisers
- Interval arithmetic guarantees that results are correct
- Track upper bound m for global minimum
- For each box X:
  - Eliminate X if $f(X)$ lies above m — cannot contain global minimum
  - Update m if $f(\text{mid}(X)) < m$
  - bisect X ("branch")



bounding a set using bisection

Generic implementation

```julia
using CUDA, IntervalArithmetic   # dps/configure  branch of IntervalArithmetic

make_box(x) = IntervalBox(Interval.(x))   # wrap a vector in a box

function minimize(f, v, numsteps = 10)

    m = +∞        # upper bound for global minimum

    for i in 1:numsteps
        fs = f.(make_box.(mid.(v)))    # interval evaluation at midpoints
        m = min(minimum(sup.(fs)), m)  # update upper bound

        v = v[inf.(f.(v)) .≤ m]        # eliminate if cannot contain global minimum

        bisected = bisect.(v)
        v = vcat(first.(bisected), last.(bisected))
    end

    return m, v
end

f( (x, y) ) = (x^2 - 2)^2 + (y^2 - 3)^2

v = [IntervalBox(-5..5, 2)]
minval, minimizers = minimize(f, v, 10);

vv = CuArray(v)
IntervalArithmetic.configure!(directed_rounding=:fast, powers=:fast)

minval, minimizers = minimize(f, vv, 50)


julia> minval, minimizers = minimize(f, vv, 50)
(2.5227748965770077e-14,
IntervalBox{2,Float64}[[-1.41422, -1.41421] × [1.73205, 1.73206],
[-1.41422, -1.41421] × [-1.73206, -1.73205],  ...)
```