Interval Methods for the GPU in Global Optimization

November 17, 2023

Department of Electrical Engineering and Computer Science

Lorenz Gillner, Ekaterina Auer

https://fiw.hs-wismar.de

Motivation

Rank	System	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
1	**Frontier** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE DOE/SC/Oak Ridge National Laboratory United States	8,699,904	1,194.00	1,679.82	22,703
2	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	442.01	537.21	29,899
3	**LUMI** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE EuroHPC/CSC Finland	2,220,288	309.10	428.70	6,016
4	**Leonardo** - BullSequana XH2000, Xeon Platinum 8358 32C 2.6GHz, NVIDIA A100 SXM4 64 GB, Quad-rail NVIDIA HDR100 Infiniband, Atos EuroHPC/CINECA Italy	1,824,768	238.70	304.47	7,404
5	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM DOE/SC/Oak Ridge National Laboratory United States	2,414,592	148.60	200.79	10,096

Fig. 1: Top 5 supercomputers of the top500

Outline

1 Intervals Libraries for the CPU

2 GPU Computing with Intervals

3 Experiments in Parameter Estimation

4 Outlook

Intervals Libraries for the CPU

Interval Libraries for the CPU

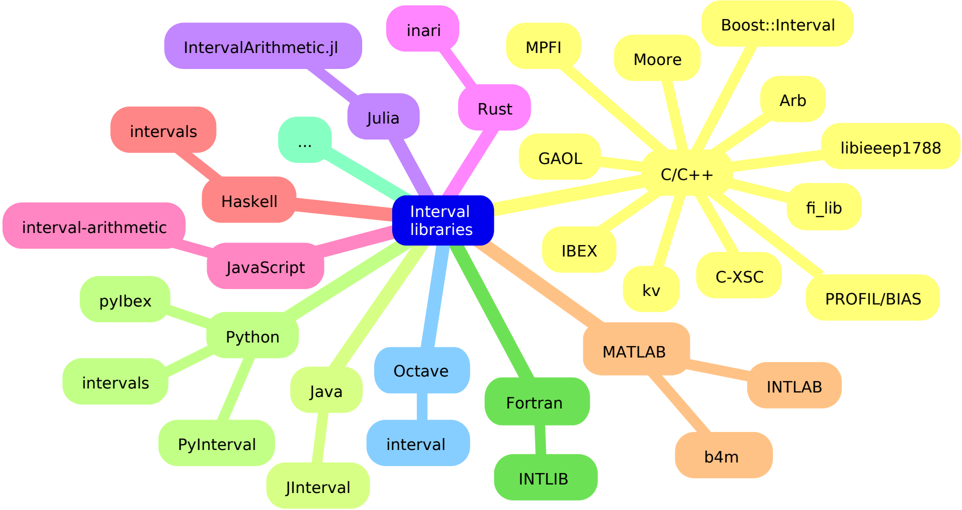

Fig. 2: Overview of interval libraries

State of the Art

Interval Libraries . . .

- support most major programming languages
- offer intuitive programming, thanks to operator overloading and OOP
- are extensible and interoperable (e.g. FADBAD++)
- include specialized toolboxes (verified solvers, optimizers, visualizations)

What about parallelization?

☞ C-XSC is considered thread-safe and has been tested with MPI and OpenMP [1, 2]
☞ many traditional optimization methods are B&B type algorithms

What about portability?

☞ dependence on platform-specific header files, libraries (MPFR, CRlibm) or inline assembly

GPU Computing with Intervals

A short introduction to GPU computing

- co-processors, originally made for computer graphics
- nowadays also used in many other domains,
  due to higher core frequencies and more RAM, while still being relatively affordable
  ☞ GPGPU
- SIMT: many low performance cores solving the same task in parallel
- abstract view of threads, blocks and grids
- focus on NVIDIA-manufactured devices, because of their high market share

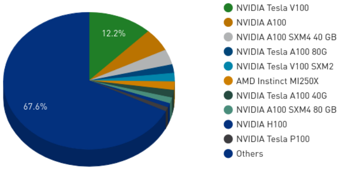

Fig. 3: Accelerator/co-processor system share [3]

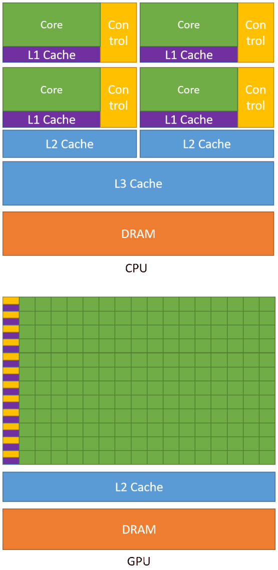

Fig. 4: Architectural differences between CPU and GPU [4]

GPU Programming

Main obstacle: vendor-specific libraries (e.g. CUDA, ROCm, oneAPI, MUSA, Metal)

Common platform: OpenCL (not supported on every device)

Standard paradigm: kernel programming (vectorization)

Rounding control: direct, stateless (arithmetic operations)

Traditionally: C/C++

```
__global__ void kernel(float* X, float* Y) {
    int i = threadIdx.x;
    Y[i] = ...
}

int main() {
    ...
    cudaMalloc(&X, T*sizeof(float));
    ...
    kernel<<<1, T>>>(X, Y);
    ...
}
```

Higher-level approach: Numba

```
@cuda.jit
def kernel(X, Y):
    i = cuda.threadIdx.x
    Y[i] = ...

...
kernel[1, T](X,Y)
...
```

... however, JIT-compiled CUDA code performs worse than native code [5]
alternative: Codon or Mojo; AOT-compiled Python code (under development) [6]

The Issue with CUDA Intervals

What is currently possible in CUDA C++

+ basic interval arithmetic with `cuda_interval_lib.h`
    - incomplete, not included in CUDA anymore
+ automatic differentiation with `Clad`
    - but only for native data types
+ solving IVPs using `Odeint` or `DifferentialEquations.jl`
    - intervals are not supported

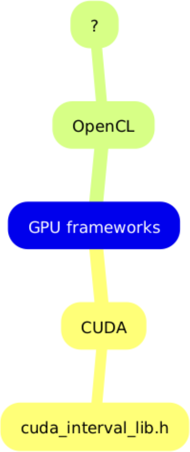

Fig. 5: Interval libraries for the GPU

High-level GPU Programming with Intervals

Standard Julia code

```julia
A = rand(1_000_000)

# define a simple function
f(x) = 0.5 * exp(x)

# apply the function to A
B = f.(A)
```

CUDA Julia code

```julia
using CUDA # or AMDGPU, Metal, ...

# this data lives in the VRAM
A = CuArray(rand(1_000_000))

f(x) = 0.5 * exp(x)

# generate and run GPU kernel
B = Array(f.(A))
```

☞ eliminating hardware barriers with generic code!

This also works with `IntervalArithmetic.jl`, though with limitations:

- no "tight" rounding on the GPU (`nextfloat`/`prevfloat` only)
- use of trigonometric functions can be tricky with intervals

Experiments in Parameter Estimation

Parameter Estimation by Example

Two-compartment model

$$\begin{aligned}\dot{y}_1 &= -y_1 \cdot (p_1 + p_3) + p_1 \cdot y_2 \\ \dot{y}_2 &= -p_2 \cdot y_2 + p_3 \cdot y_1\end{aligned} \tag{1}$$

$$y(0) = (1.0, 0.0)^T, \; \mathbf{p} \in [0.01, 2.0] \times [0.05, 3.0] \times [0.05, 3.0]$$

Analytical solution

$$\begin{aligned}D &= \sqrt{(p_1 - p_2 + p_3)^2 + 4 \cdot p_2 \cdot p_3}, \quad \alpha = \frac{p_3}{D} \\ \lambda_1 &= \frac{1}{2}(p_1 + p_2 + p_3 - D), \quad \lambda_2 = \frac{1}{2}(p_1 + p_2 + p_3 + D) \\ y_1(t) &= \frac{1}{D}((p_1 - \lambda_1) \cdot e^{-\lambda_1 \cdot t} - (p_2 - \lambda_2) \cdot e^{-\lambda_2 \cdot t}) \\ y_2(t) &= \alpha \cdot (e^{-\lambda_1 \cdot t} - e^{-\lambda_2 \cdot t})\end{aligned} \tag{2}$$

Accelerating Parameter Estimation

Objective function

$$\Phi(\mathbf{p}) = \sum_{k=t_b}^{t_e} \sum_{j=1}^{m} (y_j(t_k, \mathbf{p}) - y_{j,m}(t_k))^2 \stackrel{!}{=} min, \text{ wrt. } \mathbf{p} \tag{3}$$

Preconditioning of the search space

1. Initial search space $\mathbf{p}$
2. Bisect $\mathbf{p}$ into sub-boxes of width w:
   $\forall \mathbf{p}_k \in \mathbf{p} : diam(\mathbf{p}_k) \leq w, \mathbf{p}_1 \cup \mathbf{p}_2 \cup \ldots \cup \mathbf{p}_n = \mathbf{p}$
3. Apply the monotonicity test in parallel:
   keep $\mathbf{p}_k$, if $0 \in \nabla\Phi(\mathbf{p}_k)$
4. $\mathbf{p}^*$ is the convex hull of suitable boxes
5. Re-iterate or optimize $\mathbf{p}^*$

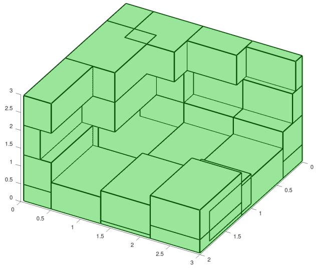

Fig. 6: Reduced search space

Experiments with the Exact Solution

- 16 measurements for y_2 only
- CUDA C++ and Julia implementations
- extended `cuda_interval_lib.h` from [7]
- manual differentiation for comparison
- CUDA port of FADBAD++ for AD

Test Environment

CPU:
2 × Intel Xeon Gold 5215 (20 cores)
GPU:
NVIDIA Quadro RTX 6000 (4608 cores)

t	1	2	3	4	5	6	...	15	16
$y_{2,m}$	0.0532	0.0478	0.0410	0.0328	0.0323	0.0148	...	0.0060	0.0126

$$\Phi(\mathbf{p}) = \sum_{k=t_b}^{t_e} (y_2(t_k, \mathbf{p}) - y_{2,m}(t_k))^2 \tag{4}$$

Experiments without the Exact Solution

Approximation of Φ with Euler's Method

$$\Phi(\mathbf{p}) = \sum_{k=t_b}^{t_e} (y_2(t_k, \mathbf{p}) - y_{2,m}(t_k))^2 \tag{5}$$

$$y^{(k)} = y^{(k-1)} + h \cdot f(y^{(k-1)}, \mathbf{p}) \tag{6}$$

$$\Phi_{approx}(\mathbf{p}) = \sum_{k=t_b}^{t_e} (y_2^{(k-1)} + h \cdot f_2(y^{(k-1)}, \mathbf{p}) - y_{2,m}(t_k))^2 \tag{7}$$

⚠ Not optimal, but there are no verified IVP solvers for the GPU yet!

A Sivia-like approach

Input: $f_{[]}, Y, X_0, \epsilon$
Output: $\mathcal{S}, \mathcal{N}, \mathcal{E}$
$\mathcal{S} \leftarrow \mathcal{N} \leftarrow \mathcal{E} \leftarrow \emptyset$;
$\mathcal{L} \leftarrow \{X_0\}$;
while $\mathcal{L} \neq \emptyset$ **do**
    $X \leftarrow$ pop($\mathcal{L}$);
    if $f_{[]}(X) \subset Y$ **then**
        push($\mathcal{S}$, X);
    else if $f_{[]}(X) \cap Y = \emptyset$ **then**
        push($\mathcal{N}$, X);
    else if $diam(X) < \epsilon$ **then**
        push($\mathcal{E}$, X);
    else
        $X_L, X_R \leftarrow$ bisect(X);
        push($\mathcal{L}$, X_L);
        push($\mathcal{L}$, X_R);
    end
end

Example

Let $X_0 = [-3, 3]$, $Y = [1, 2]$ with $\epsilon = 0.05$.

$$f_{[]}(x, y) = x^2 + y^2 + x \cdot y$$

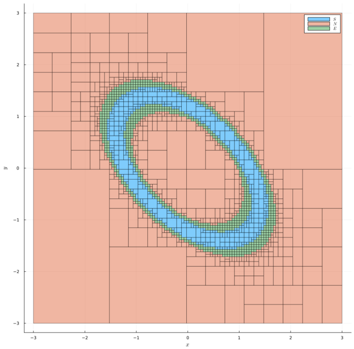

Fig. 7: Paving of $f_{[]}$

A SIVIA-like approach

Input: $f_{[]}, Y, X_0, \epsilon$
Output: $\mathcal{S}, \mathcal{N}, \mathcal{E}$
$\mathcal{S} \leftarrow \mathcal{N} \leftarrow \mathcal{E} \leftarrow \emptyset$;
$\mathcal{L} \leftarrow$ {bisect_until(X_0, w)};
forall $X_i \in \mathcal{L}$ **do**
    if $f_{[]}(X_i) \subset Y$ **then**
        push($\mathcal{S}$, X_i);
    else if $f_{[]}(X_i) \cap Y = \emptyset$ **then**
        push($\mathcal{N}$, X_i);
    else
        push($\mathcal{E}$, X_i);
    end
end

Example

Let $X_0 = [-3, 3]$, $Y = [1, 2]$ with $\epsilon = 0.05$.

$$f_{[]}(x, y) = x^2 + y^2 + x \cdot y$$

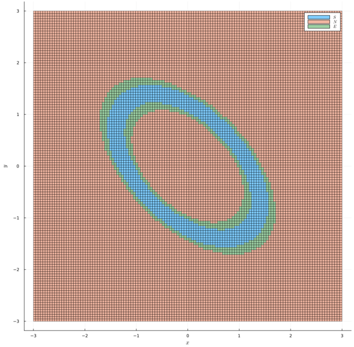

Fig. 8: Parallelized paving of $f_{[]}$

Experimental Results

Initial search space $\mathbf{p} \in [0.01, 2.0] \times [0.05, 3.0] \times [0.05, 3.0]$, $w = 0.05$

Method	Enclosure	$\Phi(\mathbf{p}_{best})$	t_{exec} (s)
I	$[0.01, 2.0] \times [2.262, 3.0] \times [0.142, 3.0]$	$[0.000092, 0.005273]$	0.350
II	$[0.383, 2.0] \times [2.539, 3.0] \times [0.142, 1.709]$	$[0.000092, 0.005273]$	9.741
III	$[1.440, 2.0] \times [1.617, 3.0] \times [0.326, 3.0]$	$[0.0, 0.005111]$	2.212
IV	$[0.383, 2.0] \times [2.539, 3.0] \times [0.142, 3.0]$	$[0.000092, 0.005273]$	0.253
V	$[0.009, 2.0] \times [2.435, 3.0] \times [0.228, 3.0]$	$[0.000135, 0.005245]$	75.558^*
VI	$[0.437, 2.0] \times [2.622, 3.0] \times [0.228, 1.695]$	$[-0.001776, 0.005245]$	504.75^*

- I Monotonicity test with MD of the exact solution (CUDA C++)
- II Monotonicity test with AD of the exact solution (CUDA C++)
- III Monotonicity test with AD of the approximate solution (CUDA C++)
- IV SIVIA-like approach with the exact solution (CUDA C++)
- V Monotonicity test with MD of the exact solution (Julia + CUDA.jl)
- VI Monotonicity test with AD of the exact solution (Julia + CUDA.jl)

* including compile time

Performance Considerations

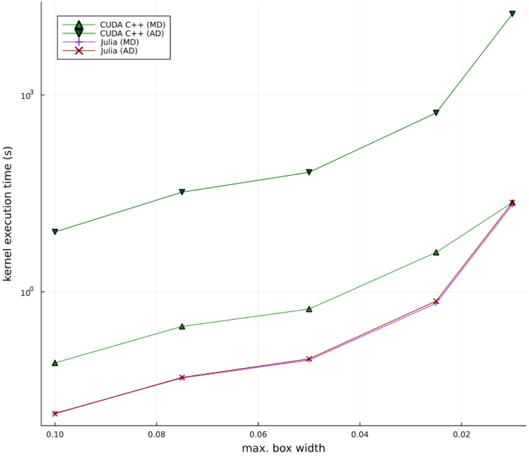

Fig. 9: Kernel execution times for C++ and Julia

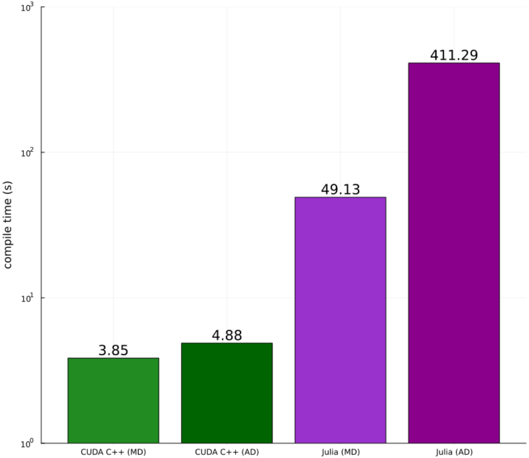

Fig. 10: Comparison of compile times

Outlook

Outlook

Future experiments . . .

- exploration of dynamic parallelism
- performance comparison between GPU and low cost CPU clusters
- investigation of energy use compared to CPU clusters

What's still needed . . .

- complete, platform-independent interval library
- GPU-compatible ODE solvers

Thank you for your attention!

References I

[1] M. Zimmer, G. Rebner, and W. Krämer.
An overview of c-xsc as a tool for interval arithmetic and its application in computing verified uncertain probabilistic models under dempster-shafer theory.
Soft Computing, 17(8):1453–1465, 2013.

[2] Markus Grimmer and W. Krämer.
An mpi and extension for the use and of c-xsc.

[3] List statistics: Top500.

[4] NVIDIA.
Cuda c++ programming guide.

[5] Lena Oden.
Lessons learned from comparing c-cuda and python-numba for gpu-computing.
In *2020 28th Euromicro international conference on parallel, distributed and network-based processing (PDP)*, pages 216–223. IEEE, 2020.

References II

[6] Ariya Shajii, Gabriel Ramirez, Haris Smajlović, Jessica Ray, Bonnie Berger, Saman Amarasinghe, and Ibrahim Numanagić.
Codon: A compiler for high-performance pythonic applications and dsls.
In *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction*, pages 191–202, 2023.

[7] Mads B Eriksen and Søren Rasmussen.
Gpu accelerated parameter estimation by global optimization using interval analysis, 2013.

[8] Grzegorz Kozikowski and Bartłomiej Jacek Kubica.
Interval arithmetic and automatic differentiation on gpu using opencl.
In *Applied Parallel and Scientific Computing: 11th International Conference, PARA 2012, Helsinki, Finland, June 10-13, 2012, Revised Selected Papers 11*, pages 489–503. Springer, 2013.

[9] Ekaterina Auer, Andreas Rauh, and Julia Kersten.
Experiments-based parameter identification on the gpu for cooperative systems.
Journal of Computational and Applied Mathematics, 371:112657, 2020.

References III

[10] Caroline Collange, Jorge Flórez, and David Defour.
A gpu interval library based on boost.interval.
In *8th conference on real numbers and computers*, pages 61–71, 2008.

[11] Caroline Collange, Marc Daumas, and David Defour.
Interval arithmetic in cuda.
In *GPU Computing Gems Jade Edition*, pages 99–107. Elsevier, 2012.

[12] Gabor Rebner and Michael Beer.
Cuda accelerated fault tree analysis with c-xsc.
In *Scalable Uncertainty Management: 6th International Conference, SUM 2012, Marburg, Germany, September 17-19, 2012. Proceedings 6*, pages 539–549. Springer, 2012.

[13] Stefan Kiel, Ekaterina Auer, and Andreas Rauh.
Uses of gpu powered interval optimization for parameter identification in the context of so fuel cells.
IFAC Proceedings Volumes, 46(23):558–563, 2013.

[14] David P. Sanders and Valentin Churavy.
Branch-and-bound interval methods and constraint propagation on the gpu using julia.
SCAN-2020, page 64, 2021.

References IV

[15] Ioana Ifrim, Vassil Vassilev, and David J Lange.
Gpu accelerated automatic differentiation with clad.
In *Journal of Physics: Conference Series*, volume 2438, page 012043. IOP Publishing, 2023.

[16] Parallel ensemble simulations.

[17] Bartłomiej Jacek Kubica.
Interval Methods for Solving Nonlinear Constraint Satisfaction, Optimization and Similar Problems.
Springer, 2019.

[18] Claus Bendtsen and Ole Stauning.
Fadbad, a flexible c++ package for automatic differentiation.
Technical report, Technical University of Denmark, 1996.