Software and Tools for Computational Engineering
RWTH AACHEN UNIVERSITY

Relaxed Differentiation & Differentiation of Relaxations

Implementation considerations for the GPU

Neil Kichler DiffProg '25 Sunday 2nd March, 2025

Relaxations

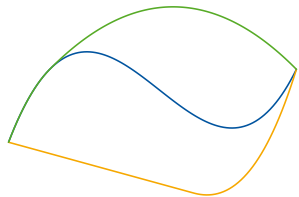
A deterministic way to bound outputs of a function.

Interval Arithmetic:

- Box with lower/upperbound in range

McCormick Relaxations:

- Interval +
- Convex/Concave relaxation



Relaxation of nonconvex, nonconcave function.

Why relax?

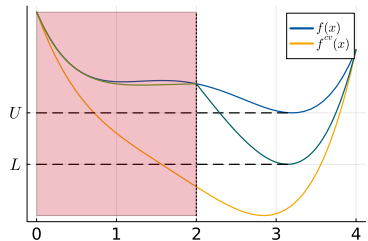
- uncertainty quantification
- verified computing
- constraint propagation/satisfaction
- nonconvex optimization:
  - subdomain separability [1]
  - lower bounding in (deterministic) global optimization [2]

Example: Deterministic Global Optimization

Deterministic Global Optimization

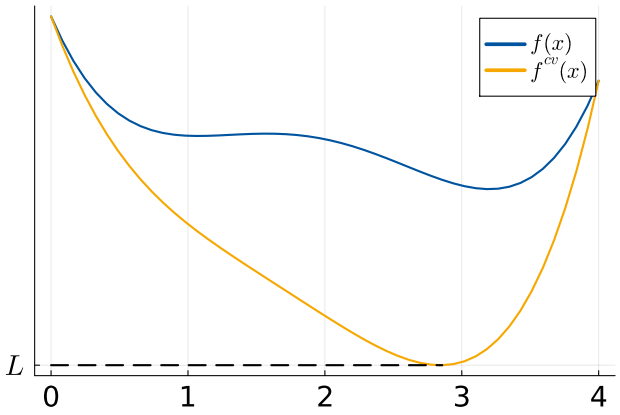
Overview:

- Usually done via Branch & Bound
- Performs:
  - Lower bounding: relaxations.
  - Upper bounding: local solver.
- Relies heavily on (convex) relaxations
  $\hookrightarrow$ McCormick relaxations.
- subgradients of relaxation computed for LP solver

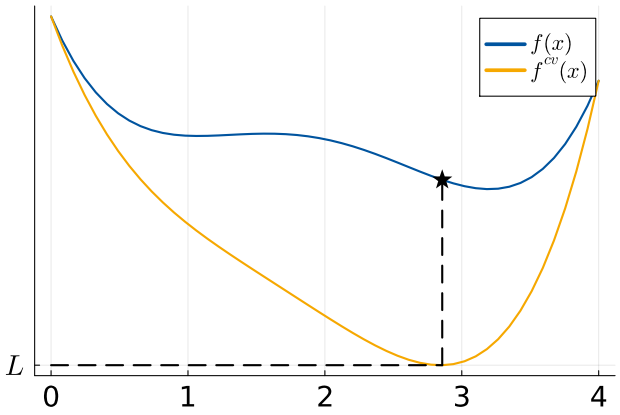


High-level illustration

Deterministic Global Optimization

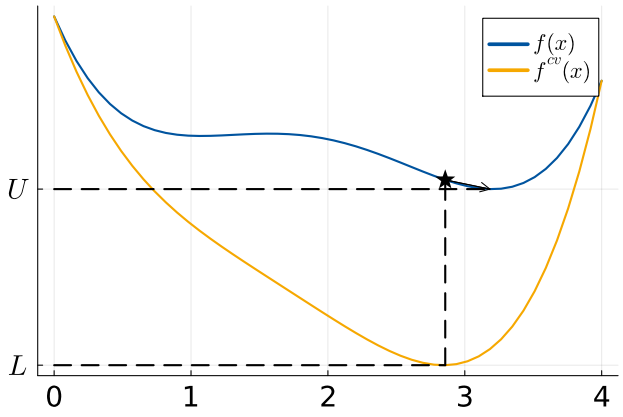


Deterministic Global Optimization

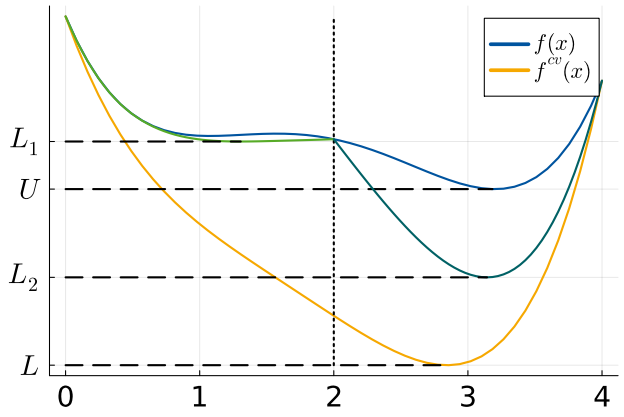


Local solver start at ★

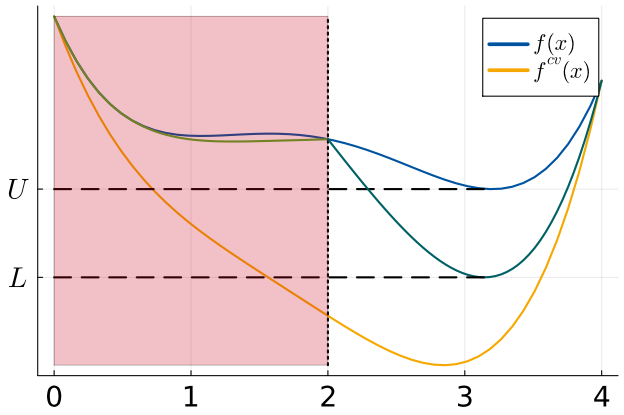
Deterministic Global Optimization



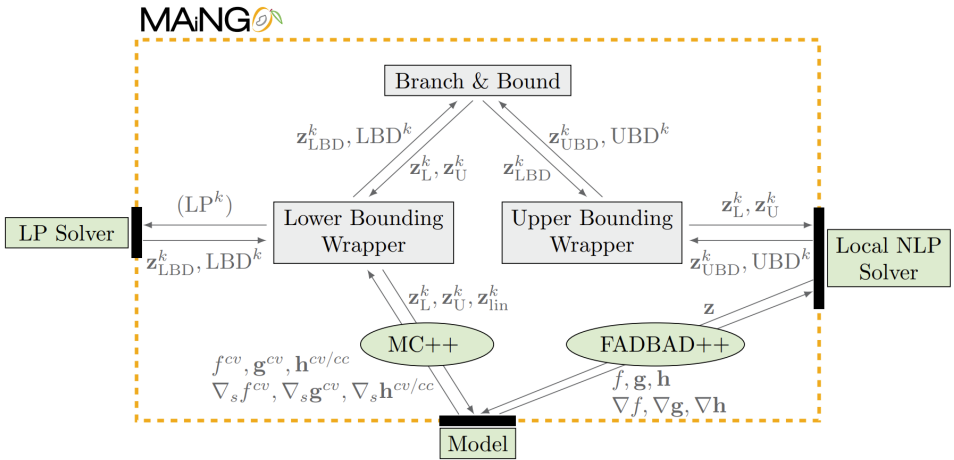
Deterministic Global Optimization



Deterministic Global Optimization



Deterministic Global Optimization



Why here at DiffProg?

Want to explore the applicability of GPUs for Relaxation $\leftrightarrow$ AD, besides

- shares a lot of similarities with AD
- one application of diff. programming
- yet another perspective on "reinterpretation" of computational programs

Outline

1. Relaxations + AD
2. Example
3. Applicability to modern GPUs
4. Scaling behavior
5. Conclusion

Relaxations + AD

Interval Arithmetic

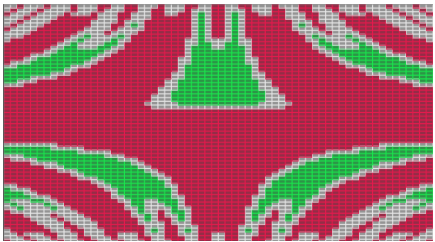
Computes function bounds:

Replace $x \in \mathbb{R}$ with $X \in \mathbb{IR}$:

$$\mathbb{IR} := \{[a, b] \mid a \leq b \wedge a, b \in \overline{\mathbb{R}}\},$$
$$X = [a, b] := \{x \in \mathbb{R} \mid a \leq x \leq b\}.$$

$\hookrightarrow$ interval extension:

$$F(X) \supseteq \{f(x) \mid x \in X\}.$$



Refinement of: $\sin(\cos(xy)y) > 0.5$

Fundamental Theorem of Interval Arithmetic

The interval extension $F : \mathbb{IR}^n \to \mathbb{IR}$ of $f : \mathbb{R}^n \to \mathbb{R}$ is guaranteed to enclose the range of f over the inputs in $X = (X_0, \cdots, X_n)$, i.e., $\mathrm{range}(f) \subseteq F(X)$ [3].

Interval Arithmetic in CUDA

`CuInterval`: Based on IEEE Std 1788-2015 (Interval Arithmetic Standard [4])

- Implements CUDA kernels for all basic operations:

  pos, neg, add, sub, mul, div, recip, sqr, sqrt, fma, pown, pow, rootn, cbrt, exp, exp2, exp10, expm1, log, log2, log10, log1p, sin, cos, tan , asin, acos, atan, atan2, sinpi, cospi, sinh, cosh, tanh, asinh, acosh, atanh, sign, ceil, floor, trunc, roundTiesToEven, roundTiesToAway, abs, min, max

- and set-based operations:

  inf, sup, mid, wid, rad, mag, mi, equa, subset, interior, disjoint, isEmpty, isEntire, less, strictLess, precedes, strictPrecedes, isMember, isSingleton, isCommonInterval, intersection, convexHul, cancelMinus, cancelPlus.

- with outward rounding accounting for intrinsic errors
  (e.g., exp(x) w/ 1 ulp error).

We only support bare and set-based (extended) interval operations.
No other flavors (e.g., Kaucher), no decorators, or other aspects of the standard.

CuInterval: Example

```
#include <cuinterval/cuinterval.h>

constexpr auto f(auto x, auto y) { return pow(x - 1, 3) - sqr(x) + 4; }

__global__ void kernel(auto *xs, auto *ys, auto *res, int n) {
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    if (i < n) { res[i] = f(xs[i], ys[i]); }
}

int main() {
    constexpr int n = 256;
    using T = cu::interval<double>;
    T xs[n], ys[n], res[n], *d_xs, *d_ys, *d_res;
    for (int i = 0; i < n; i++) { // generate dummy data
        double v = i;
        xs[i] = {{ .lb = -v, .ub = v }};
        ys[i] = {-v, v};
    }

    // ... alloc, memcpy host -> device
    kernel<<<n, 1>>>(d_xs, d_ys, d_res, n);
    // ... memcpy device -> host
}
```

Interval Arithmetic

IA has fundamental shortcomings:

- Dependency problem: partially addressable through symbolic rewriting. E.g., $x^2 - 4x$ vs. $(x-2)^2 - 4$ vs. $x(x-4)$.
- Wrapping Effect: fix would require different arithmetic.
- Limited by hardware intrinsics accuracy and rounding support.

McCormick Relaxations

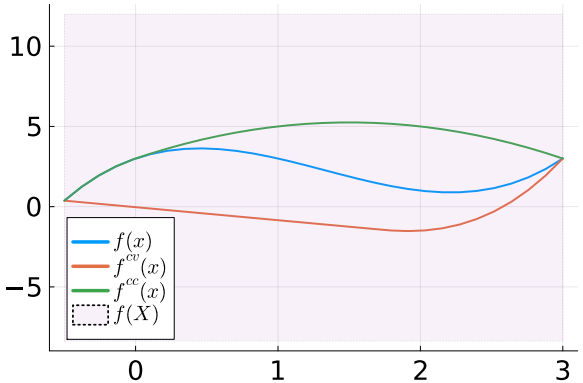
Used in lower bounding of Branch & Bound methods.

Provides:

$f^{cv}(x)$: Convex relaxation at x

$f^{cc}(x)$: Concave relaxation at x

$f(X)$: Interval over X.



McCormick relaxation of $f(x) = (x-1)^3 - x^2 + 4$

McCormick Relaxations from an AD perspective

Both conceptually start with a computational graph. Then:

McCormick relaxation:

- relaxations of elementary ops
- composition rule

AD:

- diff. rules of elementary ops
- chain rule

McCormick Composition Rule

Let $X \subseteq \mathbb{R}^n, Z \subseteq \mathbb{R}$ be nonempty convex sets. For a composite function $g = F \circ f$, where $f : X \to Z$ and $F : Z \to \mathbb{R}$, with known convex relaxations $f^{cv} : X \to \mathbb{R}$, $F^{cv} : Z \to \mathbb{R}$, and concave relaxations $f^{cc} : X \to \mathbb{R}$, $F^{cc} : Z \to \mathbb{R}$, the convex and concave relaxations of g can be computed by

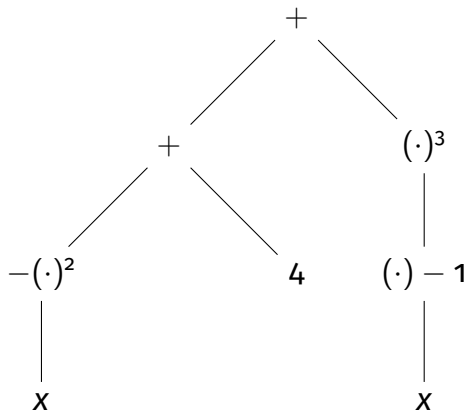
$$g^{cv}(\boldsymbol{x}) = F^{cv}(\mathtt{mid}(f^{cv}(\boldsymbol{x}), f^{cc}(\boldsymbol{x}), z_{\min})), \tag{1}$$

and

$$g^{cc}(\boldsymbol{x}) = F^{cc}(\mathtt{mid}(f^{cv}(\boldsymbol{x}), f^{cc}(\boldsymbol{x}), z_{\max})), \tag{2}$$

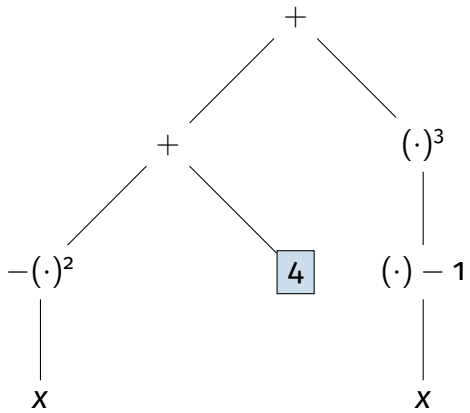
respectively.

McCormick Relaxations: By Example



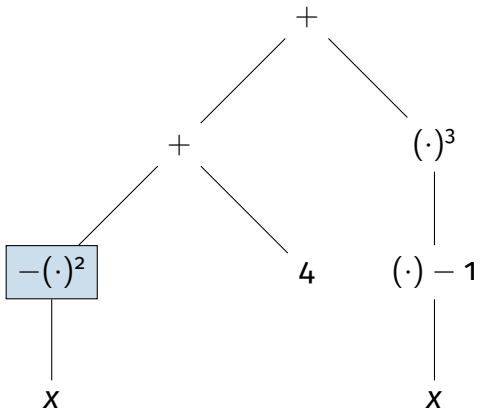
Example: $f(x) = (x-1)^3 - x^2 + 4$ with $x \in [0, 3] \subset \mathbb{R}$.

McCormick Relaxations: Constant



$$f_1(x) = 4 \Rightarrow f_1^{cv}(x) = f_1^{cc}(x) = 4$$

McCormick Relaxations: Concave



$f_2(x) = -x^2 \Rightarrow f_2^{cv}(x) = -3x, \quad f_2^{cc}(x) = -x^2$ (for $x \in [0, 3]$)

McCormick Relaxations: Concave

$f_2(x) = -x^2$ over $x \in [0, 3]$

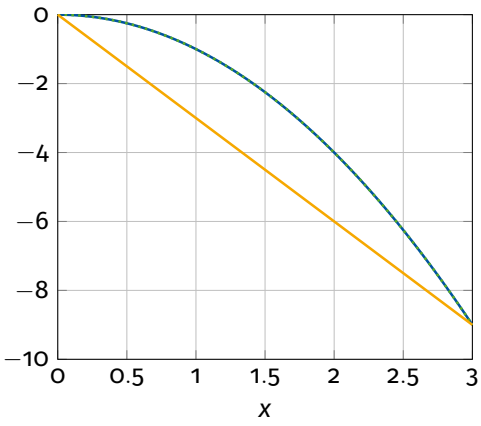
As f_2 is concave $\rightarrow$ compute chord:

$$x^L = 0, \quad x^U = 3$$

$$f_2^{cv}(x)$$

$$= -(x^U)^2 + \frac{-(x^U)^2 - (-(x^L)^2)}{x^U - x^L}(x - x^U)$$
$$= -3x$$

$$f_2^{cc}(x) = -x^2$$



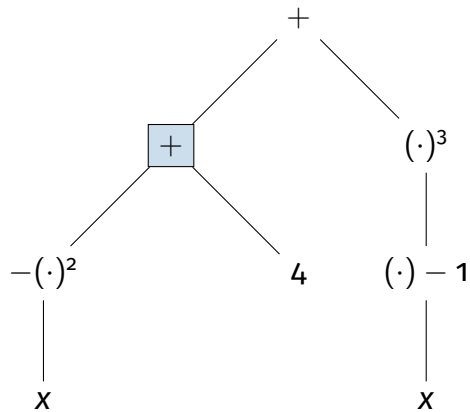
McCormick Relaxations: Add

In general:

$$f_3(x) = f_1(x) + f_2(x)$$
$$f_3^{cv}(x) = f_1^{cv}(x) + f_2^{cv}(x)$$
$$f_3^{cc}(x) = f_1^{cc}(x) + f_2^{cc}(x)$$

So:

$$f_3^{cv}(x) = -3x + 4$$
$$f_3^{cc}(x) = -x^2 + 4$$



McCormick Relaxations: Subtract

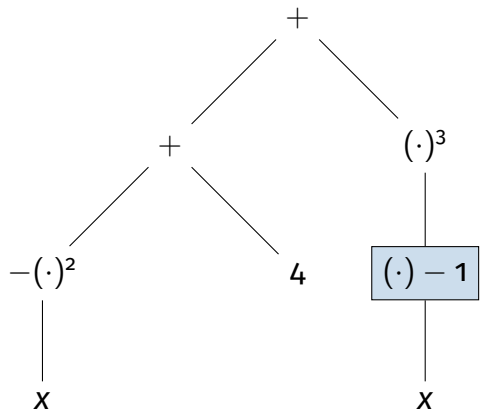
As

$$f_4(x) = x - 1$$

is affine:

$$f_4^{cv}(x) = x - 1$$

$$f_4^{cc}(x) = x - 1$$



McCormick Relaxations: Cubed

Let

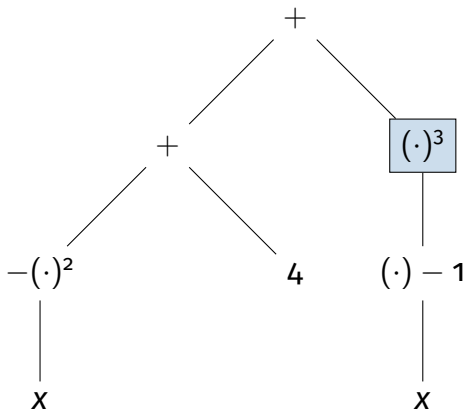
$$f_5(x) = x^3,$$

for $x \in [-1, 2]$.

McCormick Composition

$$g^{cv}(x) = F^{cv}(\text{mid}\{f^{cv}(x), f^{cc}(x), z^{min}\})$$
$$g^{cc}(x) = F^{cc}(\text{mid}\{f^{cv}(x), f^{cc}(x), z^{max}\})$$

See Desmos.



McCormick Relaxations: Result

Back to

$$f(x) = (x-1)^3 + (-x^2+4)$$

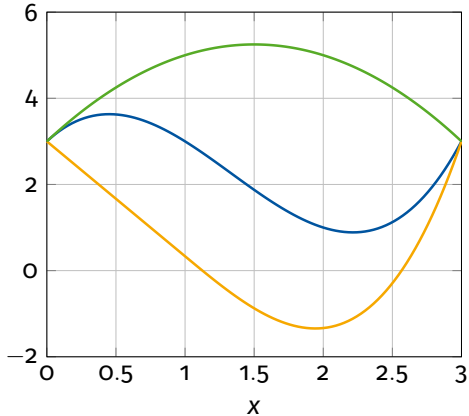
for $x \in [0, 3]$.

Relaxations:

$$f^{cv}(x) = f_5^{cv}(x) + f_3^{cv}(x)$$

$$f^{cc}(x) = f_5^{cc}(x) + f_3^{cc}(x)$$

See Desmos.



CuMcCormick: Example

```
#include <cumccormick/cumccormick.cuh>

constexpr auto f(auto x, auto y) { return pow(x - 1, 3) - sqr(x) + 4; }

__global__ void kernel(auto *xs, auto *ys, auto *res, int n) {
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    if (i < n) { res[i] = f(xs[i], ys[i]); }
}

int main() {
    constexpr int n = 256;
    using T = cu::mccormick<double>;
    T xs[n], ys[n], res[n], *d_xs, *d_ys, *d_res;
    for (int i = 0; i < n; i++) { // generate dummy data
        double v = i;
        xs[i] = {{ .lb = -v, .cv = -v, .cc = v, .ub = v }};
        ys[i] = {-v, v};
    }

    // ... alloc, memcpy host -> device
    kernel<<<n, 1>>>(d_xs, d_ys, d_res, n);
    // ... memcpy device -> host
}
```

(Vector) Tangent AD

`CuTangent`:

- forward-mode operator-overloading based AD
- supports all the C++11 `std::math` functions
- supports all functions used in `CuInterval/CuMcCormick`

  ↪ `mid`, `min`, `max` have subgradient extensions

Tangent AD $\leftrightarrow$ McCormick relaxation

`mccormick<tangent<T>>`	`tangent<mccormick<T>>`
Tangent of McCormick relaxation	McCormick relaxation of Tangent
↓	↓
Linearized relaxations	Nonlocal derivative information

Tangent of McCormick: Example

```
#include <cumccormick/cumccormick.cuh>
#include <cutangent/cutangent.cuh>

constexpr auto f(auto x, auto y) { return pow(x - 1, 3) - sqr(x) + 4; }

__global__ void kernel(auto *xs, auto *ys, auto *res, int n) {
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    if (i < n) { res[i] = f(xs[i], ys[i]); }
}

int main() {
    constexpr int n = 256;
    using T = cu::mccormick<cu::tangent<double>>;
    T xs[n], ys[n], res[n], *d_xs, *d_ys, *d_res;
    for (int i = 0; i < n; i++) { // generate dummy data
        double v = i;
        xs[i] = {{ .lb = {-v, 0.0}, .cv = {-v, double(i == 0)}, .cc = {v, double(i == 0)}, .ub = {v, 0.0}}};
        ys[i] = {-v, v};            // w.r.t variable 0
    }
    // ... alloc, memcpy host -> device
    kernel<<<n, 1>>>(d_xs, d_ys, d_res, n);
    // ... memcpy device -> host
}
```

Cuda Graph: Example

Using stream capture:

cumccormick/examples/graph/capture.cu

Using manual construction:

cumccormick/examples/graph/manual.cu

Applicability to modern GPUs

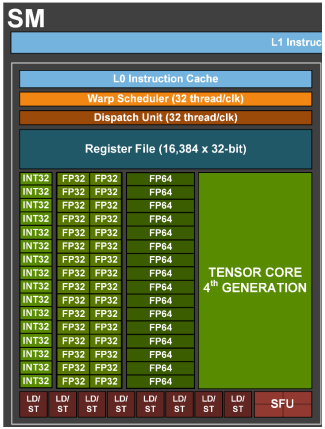
Applicability to modern GPUs: Compute

Consider a H100 SM:

- fp32 vs. fp64: 2:1 ratio (✓)
- INT32: address calculations (✓)
- SFU: special functions can occur frequently (✓)
- use of Tensor core (✗)
- use of TMA (?)

In general:

- quite unlike matmul
- cannot "tile" computation



General GPU considerations

- memory coalescing is vital
- use of (CUDA) shared memory $\rightarrow$ be aware of bank conflicts
- prefer registers > l1/shared > l2 > dram
- not too many registers to have decent potentially occupancy (although not as important on H100) -> depends on considered function
- address calculation for tangent seeding not free
- WIP: latency hiding with double buffering of tangent seeding and computation

GPU Tangent layout

Many possible shared memory layouts imaginable:

AOS:

```
template<typename T>
struct tangent { T v; T d; };

mccormick<tangent<T>> *xs;
```

- unnecessary duplication of value

GPU Tangents layout

Vector-Tangent:

```
template<typename T, int N>
struct tangents { T v; T d[N]; };

mccormick<tangents<T>> *xs;
```

lb.v	lb.d[0]	lb.d[1]	lb.d[2]	...	lb.d[30]	lb.d[31]
cv.v	cv.d[0]	cv.d[1]	cv.d[2]	...	cv.d[30]	cv.d[31]
cc.v	cv.d[0]	cc.d[1]	cc.d[2]	...	cc.d[30]	cc.d[31]
ub.v	ub.d[0]	ub.d[1]	ub.d[2]	...	ub.d[30]	ub.d[31]

+ natural way to implement it
- bank conflicts

GPU Tangents layout

Tangent layout in shared memory could be:

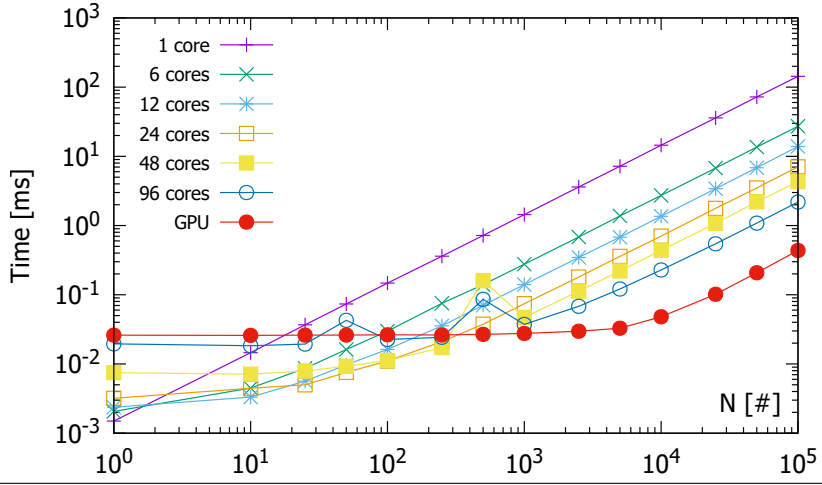
		lb.v	cv.v	cc.v	ub.v	
lb.d[0]	lb.d[1]	lb.d[2]	...	lb.d[30]	lb.d[31]
cv.d[0]	cv.d[1]	cv.d[2]	...	cv.d[30]	cv.d[31]
cv.d[0]	cc.d[1]	cc.d[2]	...	cc.d[30]	cc.d[31]
ub.d[0]	ub.d[1]	ub.d[2]	...	ub.d[30]	ub.d[31]

+ reduced bank conflicts
- inter-warp bank conflicts still present
- addressing/data movement more nuanced

What scaling behavior can we expect?

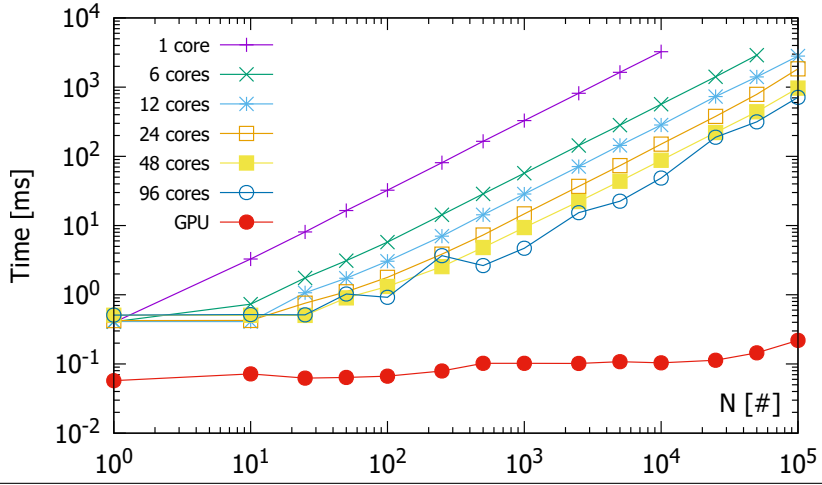
CuMcCormick vs MC++

1 variable, on 1 node @CLAIX-2023-ML [5]



CuMcCormick CUDA Graph vs MC++ Graph

1 variable, on 1 node @CLAIX-2023-ML [5]



Next steps

0. Further optimizations with async load and TMA swizzle operations
1. Further vector tangent $\leftrightarrow$ mccormick exploration
2. Explore JIT landscape
3. Higher-order combinations of AD and Relaxations?
   (will benefit from differentiable McCormick relaxations [6])
4. Compressed seeding?
5. Adjoint AD $\leftrightarrow$ Relaxation possible [7]
6. New applications? (e.g., Multistart non-convex optimization)

Conclusion

Ultimately:

- next-generation scientific software frameworks should be open to extension and allow reinterpretation of existing code
- challenges for GPUs persist
- CUDA graphs + operator overloading is a flexible solution, but with pain points.
- opportunities for JIT-compilation?


RELAX
AND
APPLY
AUTODIFF

References

[1] Jens Deussen and Uwe Naumann. “Subdomain separability in global optimization”. In: *Journal of Global Optimization* 86.3 (2023), pp. 573–588.

[2] Dominik Bongartz et al. “Deterministic global optimization of steam cycles using the IAPWS-IF97 model”. In: *Optimization & Engineering* 21 (2020), pp. 1095–1131.

[3] Ramon E. Moore et al. *Introduction to Interval Analysis*. Society for Industrial and Applied Mathematics, 2009.

[4] IEEE. “IEEE Standard for Interval Arithmetic”. In: *IEEE Std 1788-2015* (2015), pp. 1–97.

[5] *CLAIX 2023 hardware overview*. 2023. URL: https://help.itc.rwth-aachen.de/service/rhr4fjjutttf/article/fbd107191cf14c4b8307f44f545cf68a/.

[6] Kamil A Khan et al. “Differentiable mccormick relaxations”. In: *Journal of Global Optimization* 67 (2017), pp. 687–729.

[7] Markus Beckers et al. “Adjoint Mode Computation of Subgradients for McCormick Relaxations”. In: *Recent Advances in Algorithmic Differentiation*. 2012, pp. 103–113. ISBN: 978-3-642-30023-3.

Backup Slides

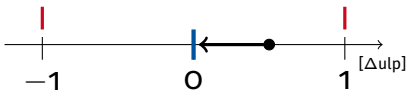
More on Interval Arithmetic Rounding

We use the CUDA intrinsic functions which use round-to-nearest-even.

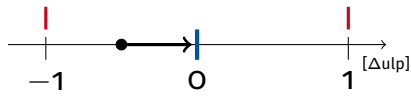
- if halway between two floating point numbers $\rightarrow$ pick even.
- else $\rightarrow$ pick nearest.
- Error of 1 ulp in intrinsic results in 2.5 ulp max error for interval lower & upper bound, i.e. for lower bound: $|\inf(F_{\text{analytic}}(X)) - \inf(F(X))| \leq 2.5$.
- 8 Scenarios:
  - halfway (odd above/below),
  - closer to even/odd (left/right),
  - exact (even/odd).

Interval Arithmetic Rounding

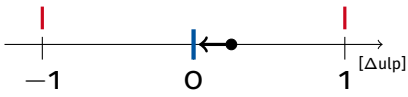
Scenarios given 1 ulp error (round-to-nearest-even):



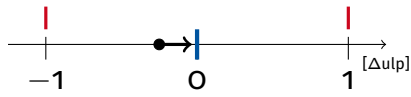
halfway (odd above)



halfway (odd below)



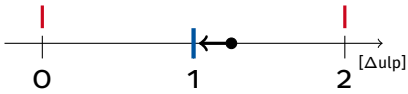
closer to even (left)



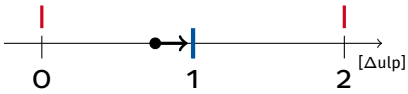
closer to even (right)

Interval Arithmetic Rounding

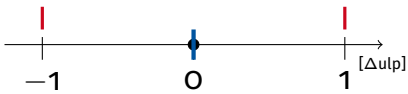
Scenarios given 1 ulp error (round-to-nearest-even):



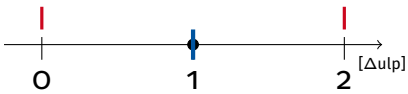
closer to odd (left)



closer to odd (right)



exact (even)



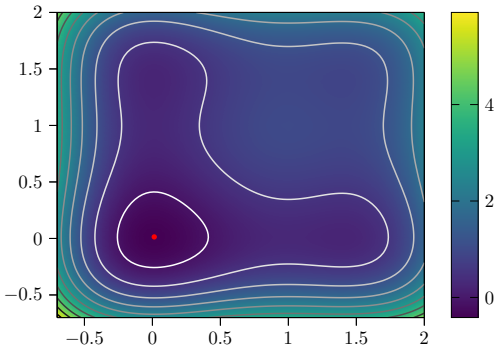
exact (odd)

Custom problem for scaling behavior

Consider

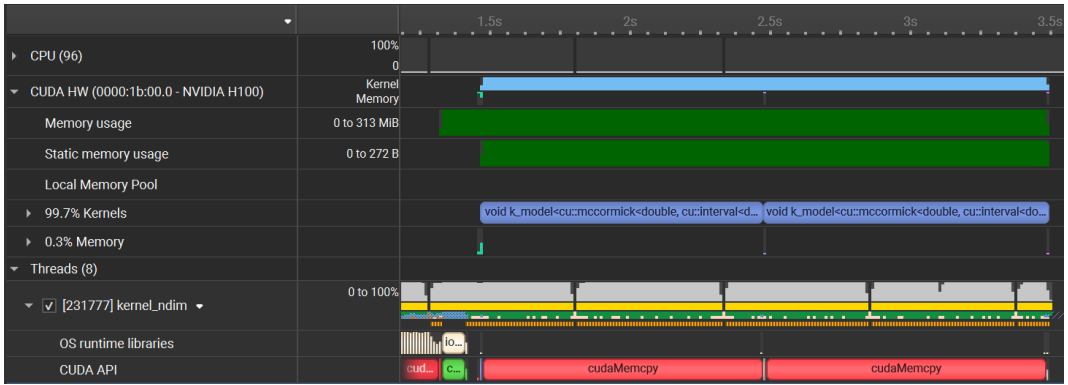
$$f(x) = \sum_{i=1}^{n} \cos(x_i - 1)^2 + (x_i - 1)^3 + (x_i - 1)^4$$

Let n = 10000, and compute 1024 McCormick relaxations at the same time.



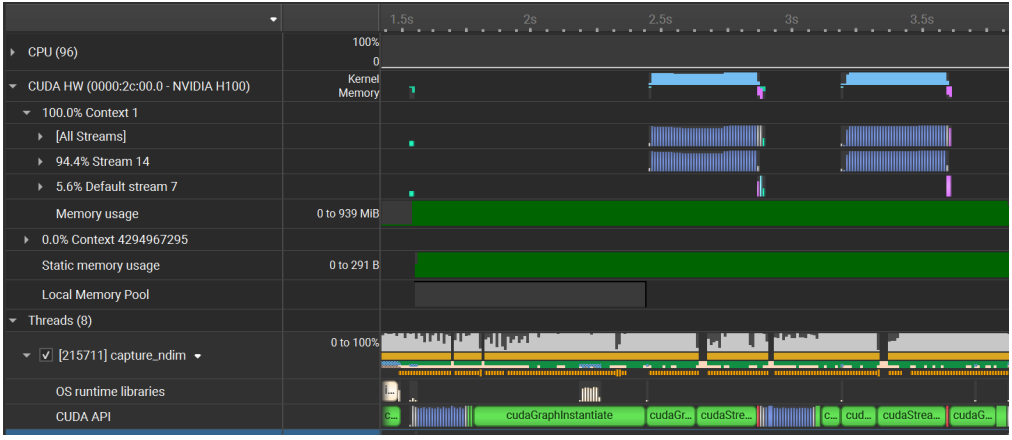
Contour plot for n=2

Direct kernel



CUDA single kernel launch

Cuda graphs (stream captured)



CUDA graph capture launch

Direct kernel vs Cuda graphs

Summary:

- For small test functions (Ackley, Beale, Rosenbrock) Cuda graphs are 6-8 times as slow as a single kernel
- Larger test function on par
- Memcpy dominates for larger n
- Most CPU time wasted waiting for synchronization
- Actual kernel launch overhead tiny with CUDA graphs.

Integration in MAiNGO

Approach:

- Create CUDA graph from existing MC++ graph

\+ Rest of the solver unchanged

\+ Same code for interval arithmetic

\- Efficiency problems of DAG remain present

\- MAiNGO currently not built for executing many McCormick relaxations at the same time.