Event Generator & Code Acceleration Workshop , CERN 14 November 2023

# pySecDec:
# Experiences Evaluating Multi-loop Amplitudes on GPUs

Stephen Jones
IPPP Durham / Royal Society URF

In collaboration with:
Heinrich, Jahn, Kerner, Langer, Magerya, Olsson, Põldaru, Schlenk, Villa
[2108.10807, 2305.19768 ]

# Workflow of a Calculation

1. Generate Feynman diagrams/amplitude (**seconds**)

2. Process amplitude (**hours**)

3. Solve system of equations relating Feynman Integrals
   ``Integral Reduction'' (**weeks/months**) Chetyrkin, Tkachov 81; Laporta 01;

4. Compute the remaining Feynman Integrals ``Master Integrals''
   (**analytic: <seconds/pspoint, numeric: ~minutes/pspoint**)

5. Generate events & compute (differential) cross-section
   (**~days/weeks**)

**I will mostly focus on this step**

# Workflow of a Calculation

1. Generate Feynman diagrams/amplitude (**seconds**)

2. Process amplitude (**hours**)

3. Solve system of equations relating Feynman Integrals
   ``Integral Reduction’’ (**weeks/months**) Chetyrkin, Tkachov 81; Laporta 01;

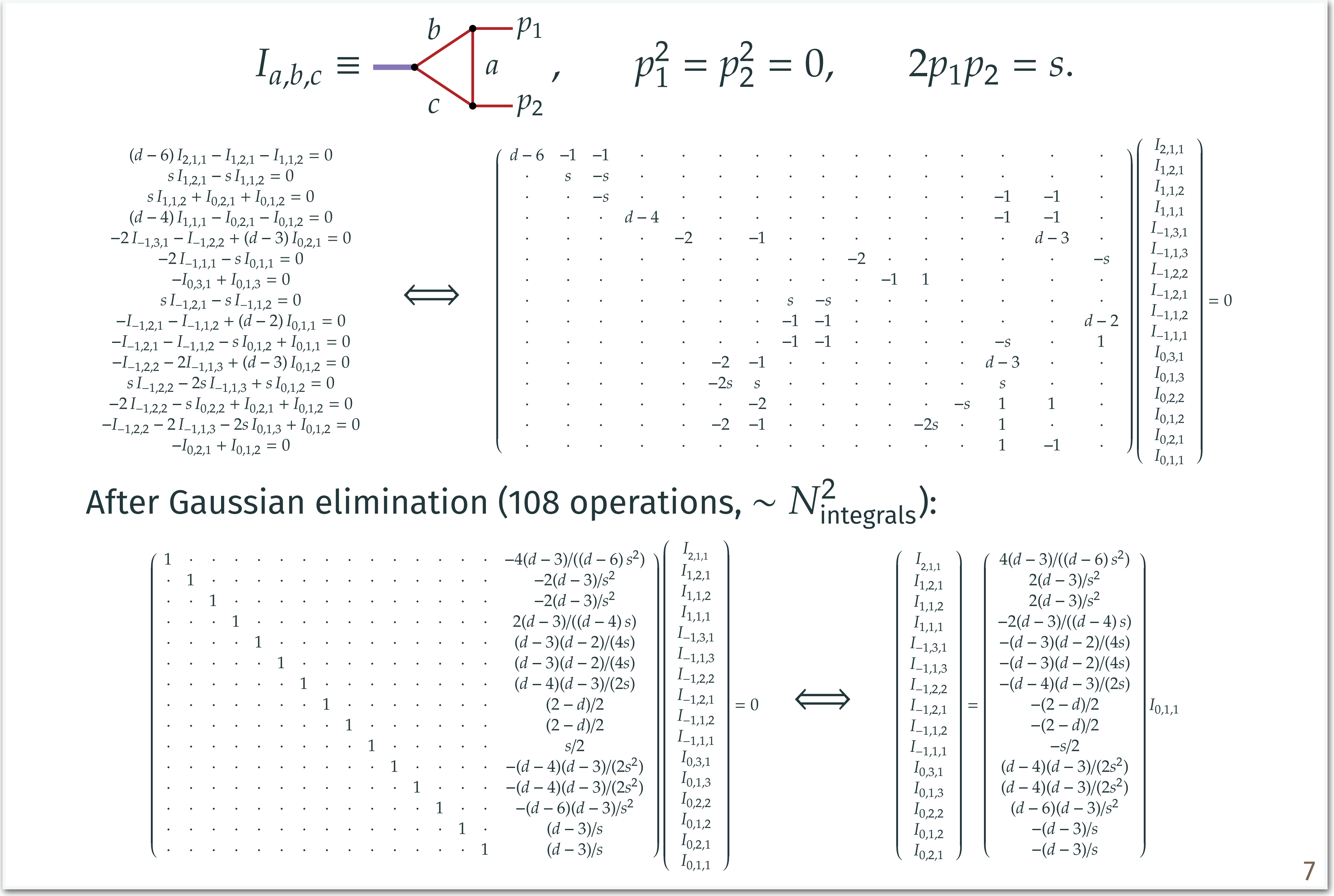
4. Compute the remaining Feynman Integrals ``Master Integrals’’
   (**analytic: <seconds/pspoint, numeric: ~minutes/pspoint**)

5. Generate events & compute (differential) cross-section
   (**~days/weeks**)

**First, a comment on this step**

# Integral Reduction

Difficulty comes from immense size of linear system (#equations & size of equations)



$$I_{a,b,c} \equiv \quad , \qquad p_1^2 = p_2^2 = 0, \qquad 2p_1p_2 = s.$$

$$\begin{array}{c}
(d-6)\,I_{2,1,1} - I_{1,2,1} - I_{1,1,2} = 0\\
s\,I_{1,2,1} - s\,I_{1,1,2} = 0\\
s\,I_{1,1,2} + I_{0,2,1} + I_{0,1,2} = 0\\
(d-4)\,I_{1,1,1} - I_{0,2,1} - I_{0,1,2} = 0\\
-2\,I_{-1,3,1} - I_{-1,2,2} + (d-3)\,I_{0,2,1} = 0\\
-2\,I_{-1,1,1} - s\,I_{0,1,1} = 0\\
-I_{0,3,1} + I_{0,1,3} = 0\\
s\,I_{-1,2,1} - s\,I_{-1,1,2} = 0\\
-I_{-1,2,1} - I_{-1,1,2} + (d-2)\,I_{0,1,1} = 0\\
-I_{-1,2,1} - I_{-1,1,2} - s\,I_{0,1,2} + I_{0,1,1} = 0\\
-I_{-1,2,2} - 2I_{-1,1,3} + (d-3)\,I_{0,1,2} = 0\\
s\,I_{-1,2,2} - 2s\,I_{-1,1,3} + s\,I_{0,1,2} = 0\\
-2\,I_{-1,2,2} - s\,I_{0,2,2} + I_{0,2,1} + I_{0,1,2} = 0\\
-I_{-1,2,2} - 2\,I_{-1,1,3} - 2s\,I_{0,1,3} + I_{0,1,2} = 0\\
-I_{0,2,1} + I_{0,1,2} = 0
\end{array}
\quad \Longleftrightarrow \quad
\begin{pmatrix}
d-6 & -1 & -1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot\\
\cdot & s & -s & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot\\
\cdot & \cdot & -s & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & -1 & -1 & \cdot\\
\cdot & \cdot & \cdot & d-4 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & -1 & -1 & \cdot\\
\cdot & \cdot & \cdot & \cdot & -2 & \cdot & -1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & d-3 & \cdot\\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & -2 & \cdot & \cdot & \cdot & \cdot & \cdot & -s\\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & -1 & 1 & \cdot & \cdot & \cdot & \cdot\\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & s & -s & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot\\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & -1 & -1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & d-2\\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & -1 & -1 & \cdot & \cdot & \cdot & \cdot & -s & \cdot & 1\\
\cdot & \cdot & \cdot & \cdot & \cdot & -2 & -1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & d-3 & \cdot & \cdot\\
\cdot & \cdot & \cdot & \cdot & \cdot & -2s & s & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & s & \cdot & \cdot\\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & -2 & \cdot & \cdot & \cdot & \cdot & \cdot & -s & 1 & 1 & \cdot\\
\cdot & \cdot & \cdot & \cdot & \cdot & -2 & -1 & \cdot & \cdot & \cdot & \cdot & -2s & \cdot & 1 & \cdot & \cdot\\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & -1 & \cdot
\end{pmatrix}
\begin{pmatrix}
I_{2,1,1}\\ I_{1,2,1}\\ I_{1,1,2}\\ I_{1,1,1}\\ I_{-1,3,1}\\ I_{-1,1,3}\\ I_{-1,2,2}\\ I_{-1,2,1}\\ I_{-1,1,2}\\ I_{-1,1,1}\\ I_{0,3,1}\\ I_{0,1,3}\\ I_{0,2,2}\\ I_{0,1,2}\\ I_{0,2,1}\\ I_{0,1,1}
\end{pmatrix} = 0$$

After Gaussian elimination (108 operations, $\sim N^2_{\text{integrals}}$):

$$\begin{pmatrix}
1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & -4(d-3)/((d-6)\,s^2)\\
\cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & -2(d-3)/s^2\\
\cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & -2(d-3)/s^2\\
\cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 2(d-3)/((d-4)\,s)\\
\cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & (d-3)(d-2)/(4s)\\
\cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & (d-3)(d-2)/(4s)\\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & (d-4)(d-3)/(2s)\\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & (2-d)/2\\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & (2-d)/2\\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & s/2\\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & -(d-4)(d-3)/(2s^2)\\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & -(d-4)(d-3)/(2s^2)\\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & -(d-6)(d-3)/s^2\\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & (d-3)/s\\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & (d-3)/s
\end{pmatrix}
\begin{pmatrix}
I_{2,1,1}\\ I_{1,2,1}\\ I_{1,1,2}\\ I_{1,1,1}\\ I_{-1,3,1}\\ I_{-1,1,3}\\ I_{-1,2,2}\\ I_{-1,2,1}\\ I_{-1,1,2}\\ I_{-1,1,1}\\ I_{0,3,1}\\ I_{0,1,3}\\ I_{0,2,2}\\ I_{0,1,2}\\ I_{0,2,1}\\ I_{0,1,1}
\end{pmatrix} = 0
\quad \Longleftrightarrow \quad
\begin{pmatrix}
I_{2,1,1}\\ I_{1,2,1}\\ I_{1,1,2}\\ I_{1,1,1}\\ I_{-1,3,1}\\ I_{-1,1,3}\\ I_{-1,2,2}\\ I_{-1,2,1}\\ I_{-1,1,2}\\ I_{-1,1,1}\\ I_{0,3,1}\\ I_{0,1,3}\\ I_{0,2,2}\\ I_{0,1,2}\\ I_{0,2,1}
\end{pmatrix} =
\begin{pmatrix}
4(d-3)/((d-6)\,s^2)\\
2(d-3)/s^2\\
2(d-3)/s^2\\
-2(d-3)/((d-4)\,s)\\
-(d-3)(d-2)/(4s)\\
-(d-3)(d-2)/(4s)\\
-(d-4)(d-3)/(2s)\\
-(2-d)/2\\
-(2-d)/2\\
-s/2\\
(d-4)(d-3)/(2s^2)\\
(d-4)(d-3)/(2s^2)\\
(d-6)(d-3)/s^2\\
-(d-3)/s\\
-(d-3)/s
\end{pmatrix} I_{0,1,1}$$

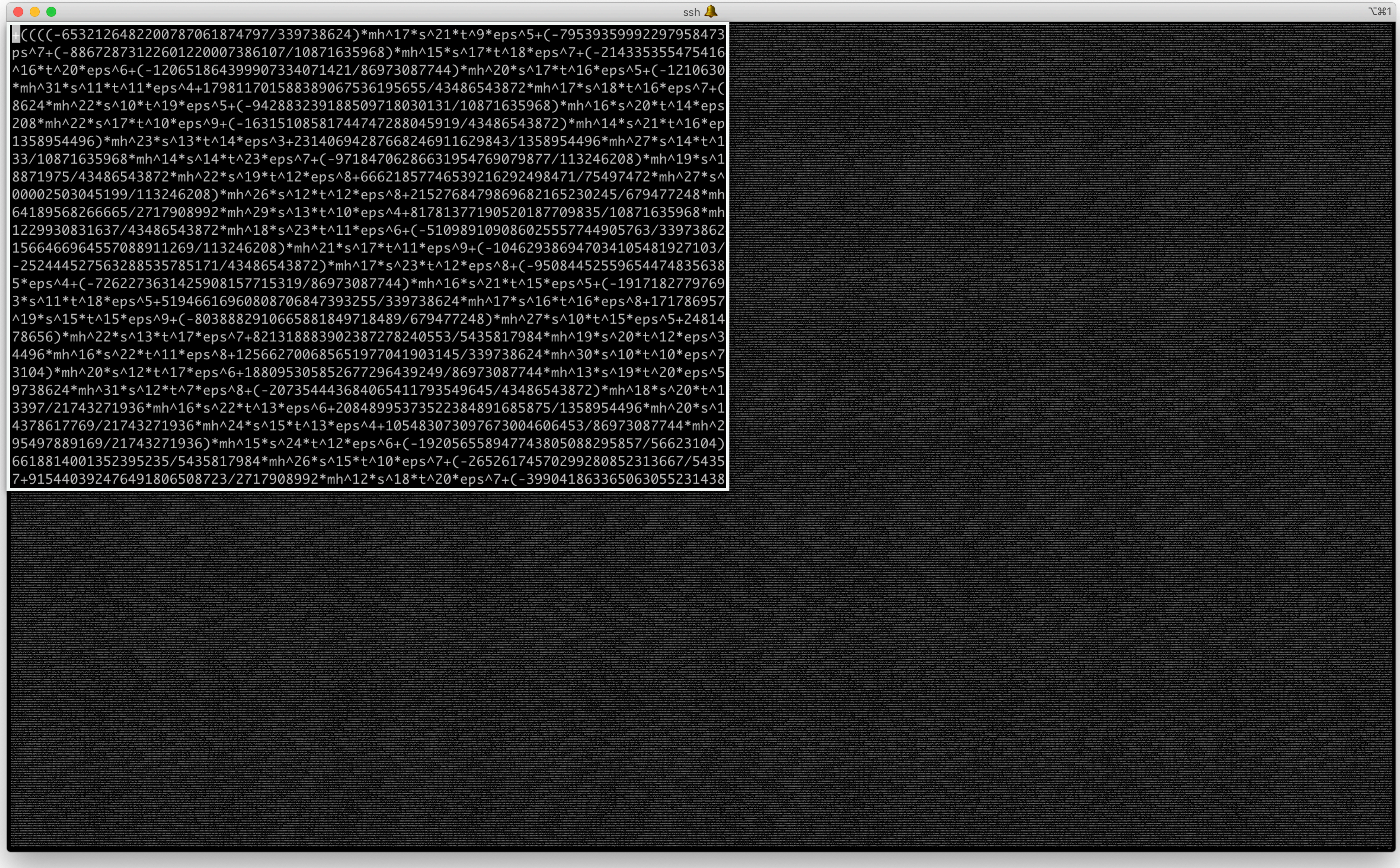
7

Slide: Vitaly Magerya (Siegen 2023)

# Handling Rational Functions

**Typical system sizes:** $\mathcal{O}(10k)$ integrals → $\mathcal{O}(500)$

**Typical coefficient:** $\mathcal{O}(1) - \mathcal{O}(100)$ mb



```
+((((-6532126482200787061874797/339738624)*mh^17*s^21*t^9*eps^5+(-79539359992297958473
ps^7+(-8867287312260122000738610 7/10871635968)*mh^15*s^17*t^18*eps^7+(-214335355475416
^16*t^20*eps^6+(-12065186439990733407 1421/86973087744)*mh^20*s^17*t^16*eps^5+(-1210630
*mh^31*s^11*t^11*eps^4+17981170158838906753 6195655/43486543872*mh^17*s^18*t^16*eps^7+(
8624*mh^22*s^10*t^19*eps^5+(-942883239188509718030131/10871635968)*mh^16*s^20*t^14*eps
208*mh^22*s^17*t^10*eps^9+(-163151085817447472880 45919/43486543872)*mh^14*s^21*t^16*ep
1358954496)*mh^23*s^13*t^14*eps^3+231406942876682469116 29843/1358954496*mh^27*s^14*t^1
33/10871635968*mh^14*s^14*t^23*eps^7+(-971847062866319547690 79877/113246208)*mh^19*s^1
8871975/43486543872*mh^22*s^19*t^12*eps^8+666218577465392162924 98471/75497472*mh^27*s^
00002503045199/113246208)*mh^26*s^12*t^12*eps^8+215276847986968216523 0245/679477248*mh
64189568266665/2717908992*mh^29*s^13*t^10*eps^4+81781377190520187709835/10871635968*mh
1229930831637/43486543872*mh^18*s^23*t^11*eps^6+(-5109891090860255577449 05763/33973862
156646696455708891126 9/113246208)*mh^21*s^17*t^11*eps^9+(-10462938694703410548192 7103/
-25244452756328853578517 1/43486543872)*mh^17*s^23*t^12*eps^8+(-9508445255965447483563 8
5*eps^4+(-72622736314259081577153 19/86973087744)*mh^16*s^21*t^15*eps^5+(-1917182779769
3*s^11*t^18*eps^5+519466169608087068473932 55/339738624*mh^17*s^16*t^16*eps^8+171786957
^19*s^15*t^15*eps^9+(-803888291066588184971848 9/679477248)*mh^27*s^10*t^15*eps^5+24814
78656)*mh^22*s^13*t^17*eps^7+821318883902387278240553/5435817984*mh^19*s^20*t^12*eps^3
4496*mh^16*s^22*t^11*eps^8+125662700685651977041903145/339738624*mh^30*s^10*t^10*eps^7
3104)*mh^20*s^12*t^17*eps^6+188095305852677296439249/86973087744*mh^13*s^19*t^20*eps^5
9738624*mh^31*s^12*t^7*eps^8+(-2073544436840654117935496 45/43486543872)*mh^18*s^20*t^1
3397/21743271936*mh^16*s^22*t^13*eps^6+208489953735223848916858 75/1358954496*mh^20*s^1
4378617769/21743271936*mh^24*s^15*t^13*eps^4+105483073097673004606453/86973087744*mh^2
95497889169/21743271936)*mh^15*s^24*t^12*eps^6+(-192056558947743805088295857/56623104)
661881400135239523 5/5435817984*mh^26*s^15*t^10*eps^7+(-265261745702992808523136 67/5435
7+915440392476491806508723/2717908992*mh^12*s^18*t^20*eps^7+(-399041863365063055231438
```

# Handling Rational Functions

**Rational Reconstruction:** recover analytic results from numerical samples

1) Evaluate rational function $f$ over $\mathbb{Z}_p$ (integers modulo prime) several times

$$(\mathbf{z}, p) \longrightarrow \boxed{\quad f \quad} \longrightarrow f(\mathbf{z}) \bmod p.$$

2) Use multivariate rational reconstruction, Chinese remainder theorem to infer analytic form of $f$ von Manteuffel, Schabinger 14, Peraro 16, 19; Klappert, Lange 19, Wang 81;

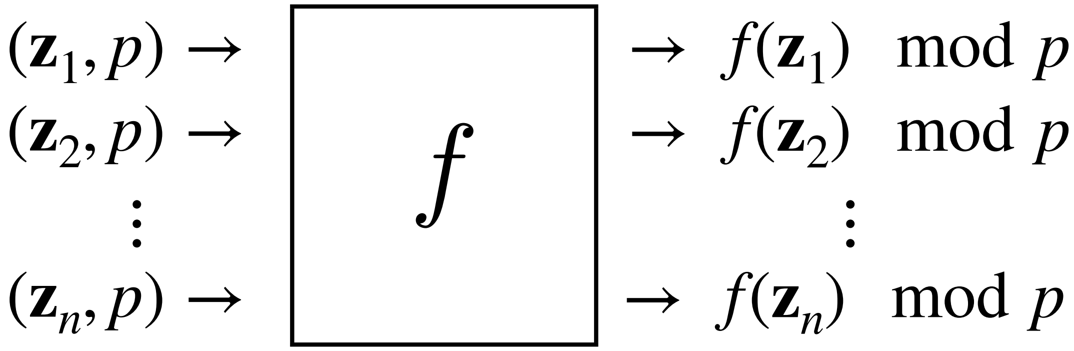
**Avoids:** intermediate expression swell, intermediate arbitrary precision

Implented in several public computer packages:
Fermat, FinRed, FiniteFlow, Kira+FireFly, Caravel, Ratracer, …

Lewis 94; von Mantueffel (Private); Peraro 16; Maierhöfer, Usovitsch, Uwer 18; Klappert, Lange, Maierhöfer, Usovitsch 20; Klappert, Klein, Lange 20; Abreu, Dormans, Febres Cordero, Ita, Kraus, Page, Pascual, Ruf, Sotnikov; Magerya 22; …

# Handling Rational Functions

**To my naive mind:** This problem seems like it might work well on a GPU…



$$\begin{array}{rcl} (\mathbf{z}_1, p) \rightarrow & & \rightarrow f(\mathbf{z}_1) \mod p \\ (\mathbf{z}_2, p) \rightarrow & \boxed{f} & \rightarrow f(\mathbf{z}_2) \mod p \\ \vdots & & \vdots \\ (\mathbf{z}_n, p) \rightarrow & & \rightarrow f(\mathbf{z}_n) \mod p \end{array}$$

**Profit?**

**Obvious issues:**

1. Are GPUs really much faster with modular arithmetic?
2. Sampling vs reconstructing time?
3. Enough memory?

**Reconstruction ~ Solving Linear Systems (also on GPU?)**

**Common trick: ``mask" parts of system, reconstruct in batches**

**→Talk of Alessandro**

# Computing Feynman Integrals

**Feynman integrals can be difficult to compute analytically**

**Various methods to approximate/evaluate them numerically**

Numerical differential equations

**ODE/PDE**

Series solutions of differential equations (AMFlow, DiffExp, Seasyde)

Taylor expansion in Feynman parameters (TayInt)

**Series Solutions**

Numerical Mellin-Barnes (MB, Ambre)

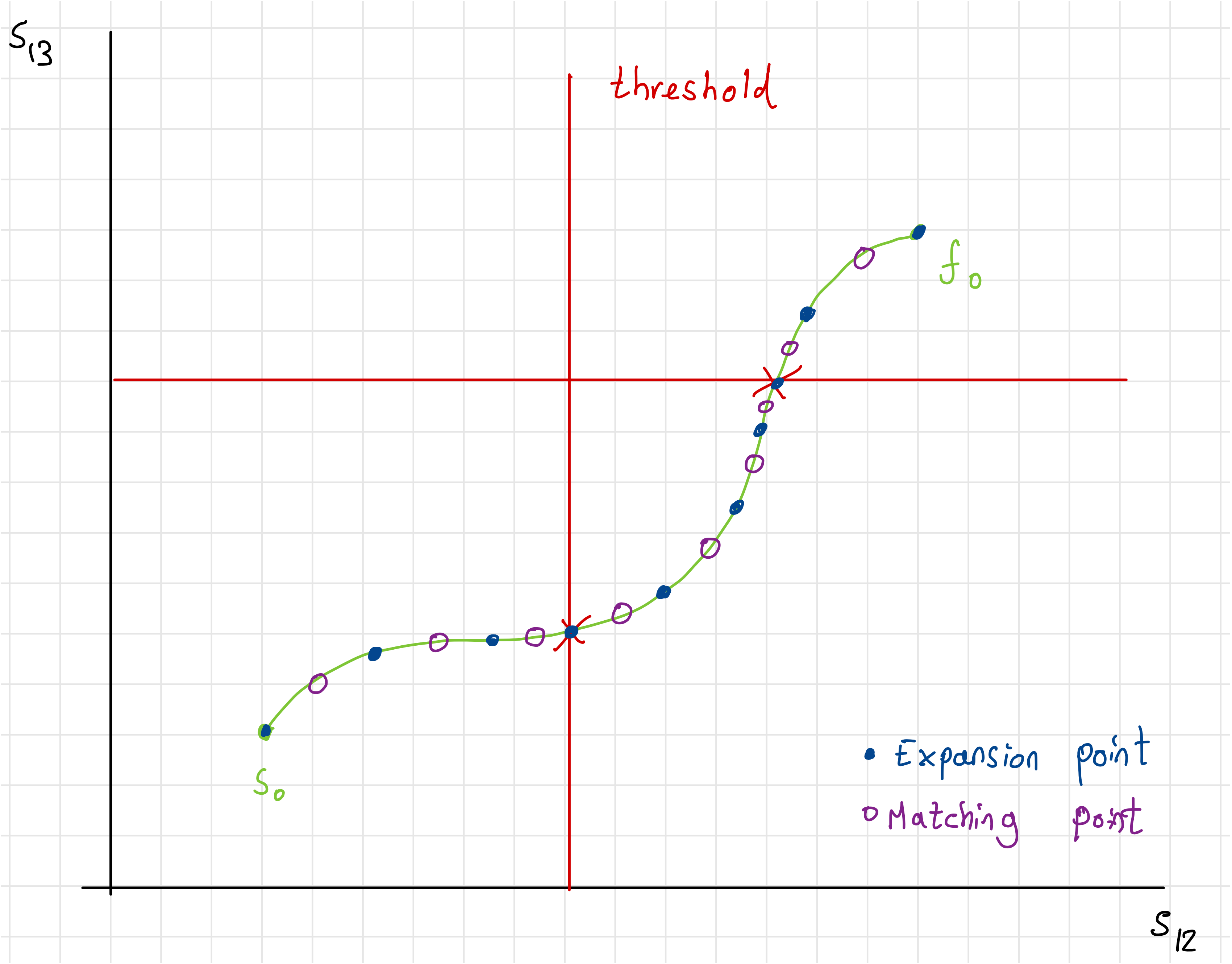
Tropical sampling (Feyntrop)

Numerical Loop-Tree Duality (cLTD, Lotty)

Sector decomposition (Sector_decomposition, FIESTA, pySecDec)

**~Monte Carlo Integration**

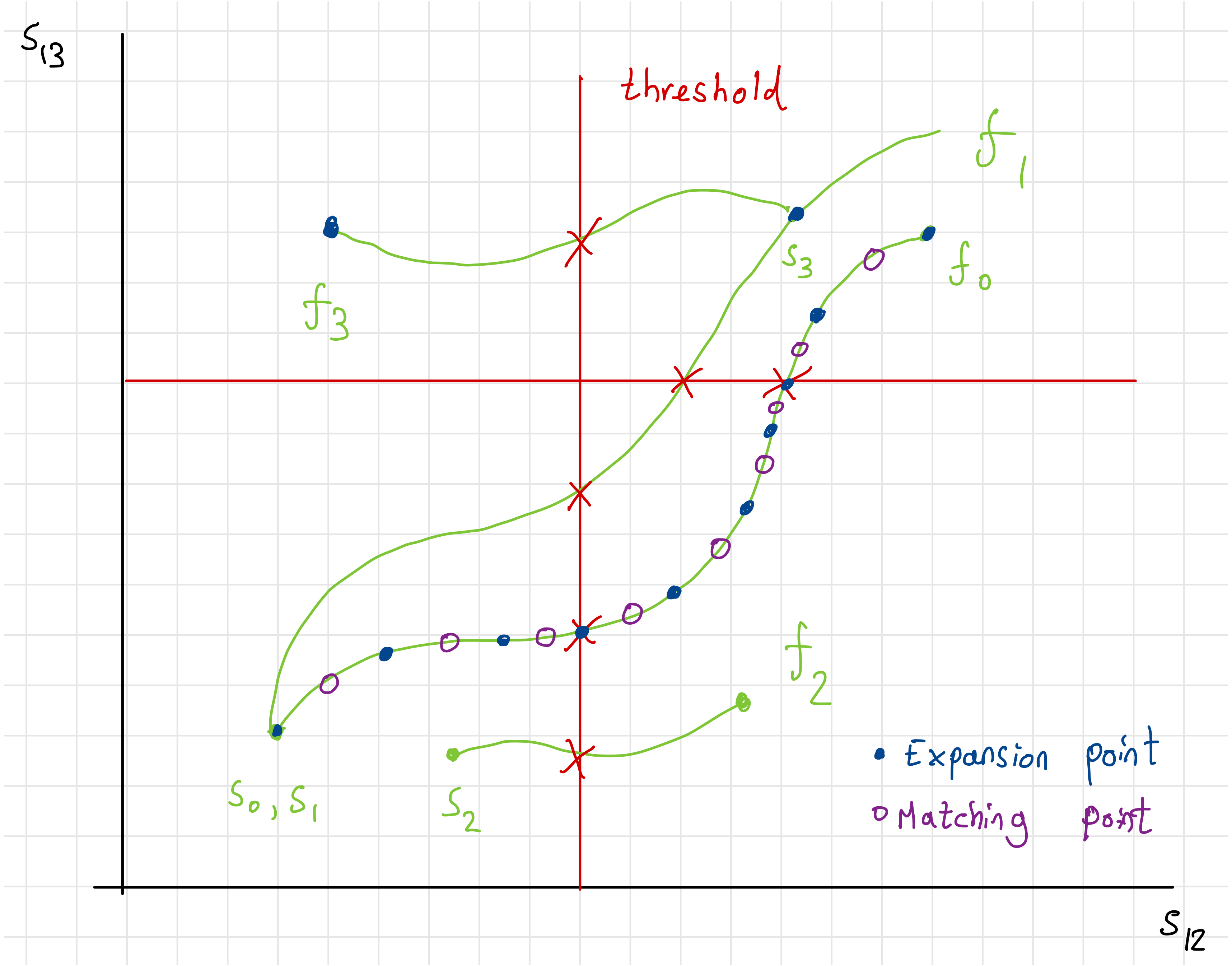
# Series Solutions



**My naive guess at one challenge:**

Often want/require very high precision intermediate results → high/arb. precision

# Series Solutions



**My naive guess at one challenge:**

Often want/require very high precision intermediate results → high/arb. precision

# Sector Decomposition
# & Quasi-Monte Carlo Integration

# Sector Decomposition in a Nutshell

$$I \sim \int_{\mathbb{R}^{N+1}_{>0}} [\mathrm{d}\boldsymbol{x}]\, \boldsymbol{x}^{\boldsymbol{\nu}} \frac{[\mathcal{U}(\boldsymbol{x})]^{N-(L+1)D/2}}{[\mathcal{F}(\boldsymbol{x},\mathbf{s}) - i\delta]^{N-LD/2}} \delta(1 - H(\boldsymbol{x}))$$

**Singularities**

1. UV/IR singularities when some $x \to 0$ simultaneously $\Longrightarrow$ Sector Decomposition

2. Thresholds when $\mathcal{F}$ vanishes inside integration region $\Longrightarrow$ $i\delta$

**Sector decomposition**

Find a local change of coordinates for each singularity that factorises it (blow-up)

# Sector Decomposition in a Nutshell (II)

$$I \sim \int_{\mathbb{R}^N_{>0}} \left[\mathrm{d}\boldsymbol{x}\right] \boldsymbol{x}^{\boldsymbol{\nu}} \left(c_i\, \boldsymbol{x}^{\mathbf{r}_i}\right)^t$$

$$\mathcal{N}(I) = \mathrm{convHull}(\mathbf{r}_1, \mathbf{r}_2, \ldots) = \bigcap_{f\in F} \left\{ \mathbf{m} \in \mathbb{R}^N \mid \langle \mathbf{m}, \mathbf{n}_f\rangle + a_f \geq 0 \right\}$$

Normal vectors incident to each extremal vertex define a local change of variables*

Kaneko, Ueda 10

$$x_i = \prod_{f\in S_j} y_f^{\langle \mathbf{n}_f, \mathbf{e}_i\rangle}$$

$$I \sim \sum_{\sigma\in\Delta^T_{\mathcal{N}}} |\sigma| \int_0^1 \left[\mathrm{d}\mathbf{y}_f\right] \underbrace{\prod_{f\in\sigma} y_f^{\langle \mathbf{n}_f,\boldsymbol{\nu}\rangle - t a_f}}_{\textbf{Singularities}} \left( \underbrace{c_i \prod_{f\in\sigma} y_f^{\langle \mathbf{n}_f,\mathbf{r}_i\rangle + a_f}}_{\textbf{Finite}} \right)^t$$

*If $|S_j| > N$, need triangulation to define variables (simplicial normal cones $\sigma \in \Delta^T_{\mathcal{N}}$ )

# Quasi-Monte Carlo

Li, Wang, Yan, Zhao 15; de Doncker, Almulihi, Yuasa 17, 18; de Doncker, Almulihi 17;
Kato, de Doncker, Ishikawa, Yuasa 18

$$Q_n^{(k)}[f] \equiv \frac{1}{n} \sum_{i=0}^{n-1} f\left(\left\{\frac{i\mathbf{z}}{n} + \mathbf{\Delta}_k\right\}\right) \qquad I[f] \approx \bar{Q}_{n,m}[f] \equiv \frac{1}{m} \sum_{k=0}^{m-1} Q_n^{(k)}[f],$$

$\{\}$ - Fractional part

$\Delta_k$ - Random shift vector

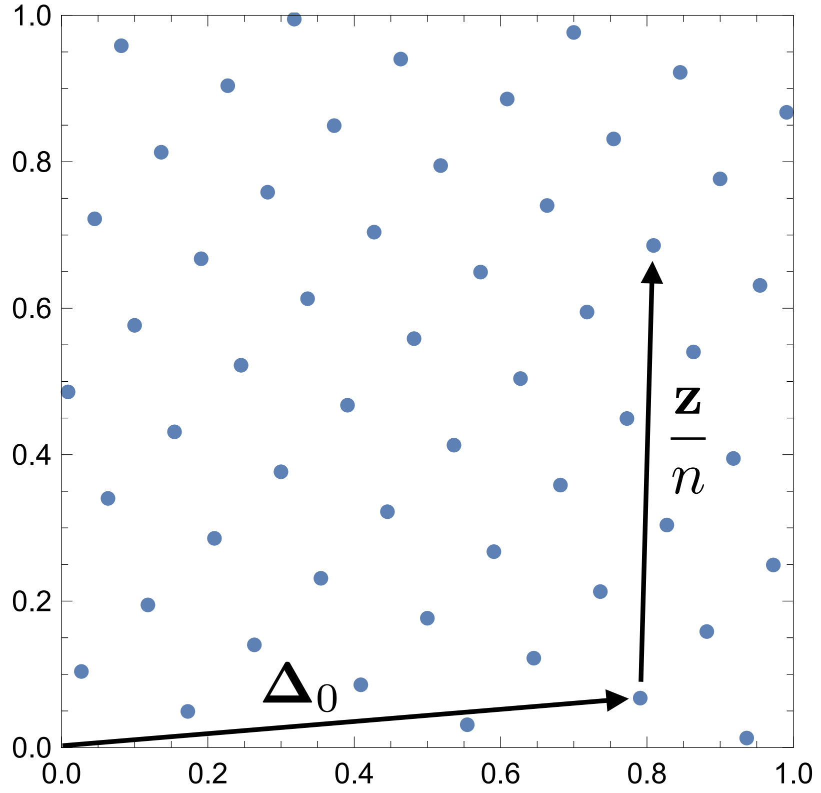
$\mathbf{z}$ - Generating vector

**Previously:**

Precompute $\mathbf{z}$ with (CBC) construction
Nuyens, Cools 06

Guarantee error $\sim 1/n^\alpha$ if $\delta_x^{(\alpha)} I(\mathbf{x})$ is square-integrable and periodic Dick, Kuo, Sloan 13

CBC needs $\mathcal{O}(n)$ bytes memory $n \lesssim 4.10^{10}$ @ 2TB
Can encounter ``unlucky'' lattices



# Periodising Transforms

Lattice rules work especially well for continuous, smooth and periodic functions
Functions can be periodized by a suitable change of variables: $\mathbf{x} = \phi(\mathbf{u})$

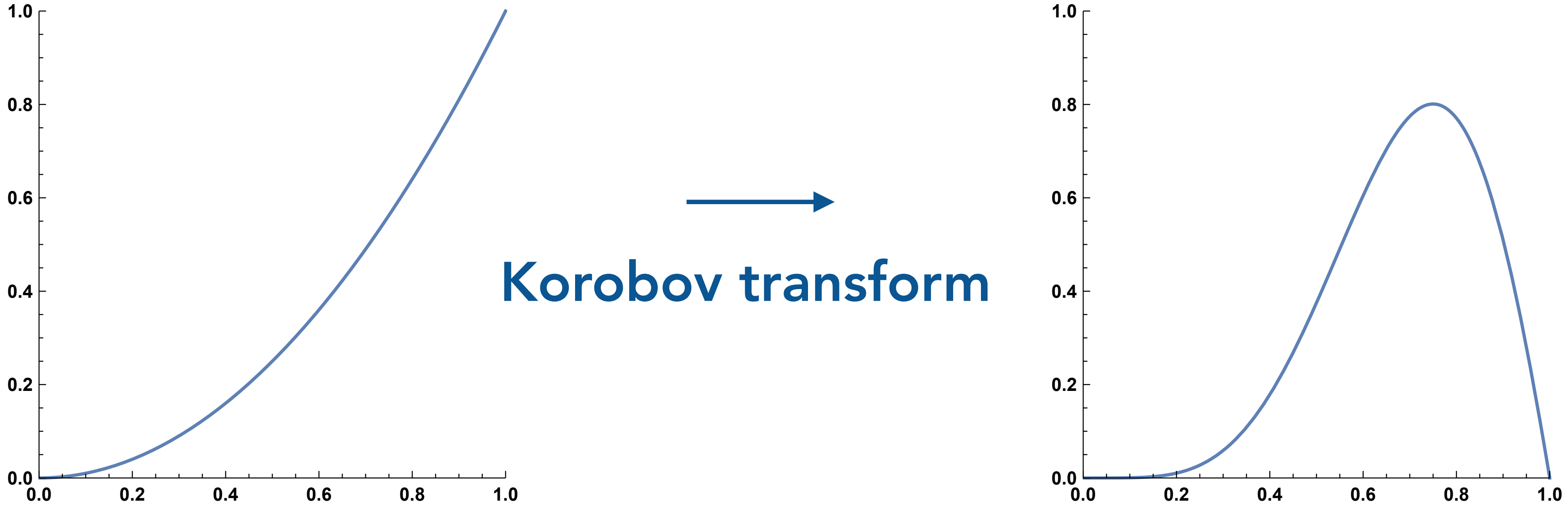
$$I[f] \equiv \int_{[0,1]^d} \mathrm{d}\mathbf{x}\ f(\mathbf{x}) = \int_{[0,1]^d} \mathrm{d}\mathbf{u}\ \omega_d(\mathbf{u}) f(\phi(\mathbf{u}))$$

$$\phi(\mathbf{u}) = (\phi(u_1), \ldots, \phi(u_d)), \qquad \omega_d(\mathbf{u}) = \prod_{j=1}^{d} \omega(u_j) \qquad \text{and} \qquad \omega(u) = \phi'(u)$$

**Korobov transform:** $\omega(u) = 6u(1-u), \quad \phi(u) = 3u^2 - 2u^3$

**Sidi transform:** $\omega(u) = \pi/2 \sin(\pi u), \quad \phi(u) = 1/2(1 - \cos \pi t)$

**Baker transform:** $\phi(u) = 1 - |2u - 1|$



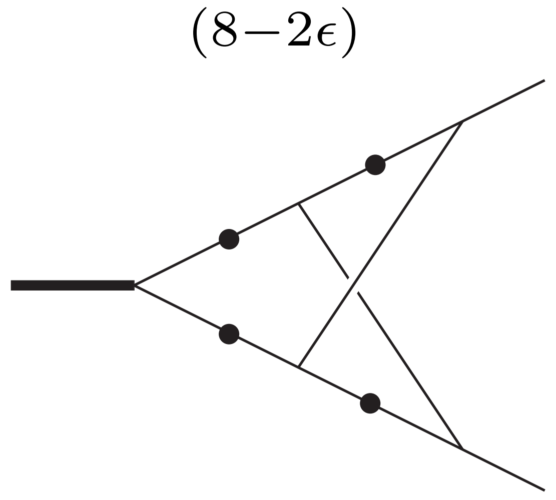
**Korobov transform**

# 1. Performance Improvements

# IntLib Style Usage

Let’s compute a finite Feynman integral (à la de Doncker, Almulihi, Yuasa 17 )

```
#include <iostream>
#include "qmc.hpp"
struct formfactor2L_t {
    const unsigned long long int number_of_integration_variables = 5;
#ifdef __CUDACC__
    __host__ __device__
#endif
    double operator()(const double arg[]) const
    {
        // Simplex to cube transformation
        double x0  = arg[0];
        double x1  = (1.-x0)*arg[1];
        double x2  = (1.-x0-x1)*arg[2];
        double x3  = (1.-x0-x1-x2)*arg[3];
        double x4  = (1.-x0-x1-x2-x3)*arg[4];
        double x5  = (1.-x0-x1-x2-x3-x4);
        double wgt =
        (1.-x0)*
        (1.-x0-x1)*
        (1.-x0-x1-x2)*
        (1.-x0-x1-x2-x3);
        if(wgt <= 0) return 0;
        // Integrand
        double u=x2*(x3+x4)+x1*(x2+x3+x4)+(x2+x3+x4)*x5+x0*(x1+x3+x4+x5);
        double f=x1*x2*x4+x0*x2*(x1+x3+x4)+x0*(x2+x3)*x5;
        double n=x0*x1*x2*x3;
        double d = f*f*u*u;
        return wgt*n/d;
    }
} formfactor2L;
```



Note: can compile this code with or without CUDA

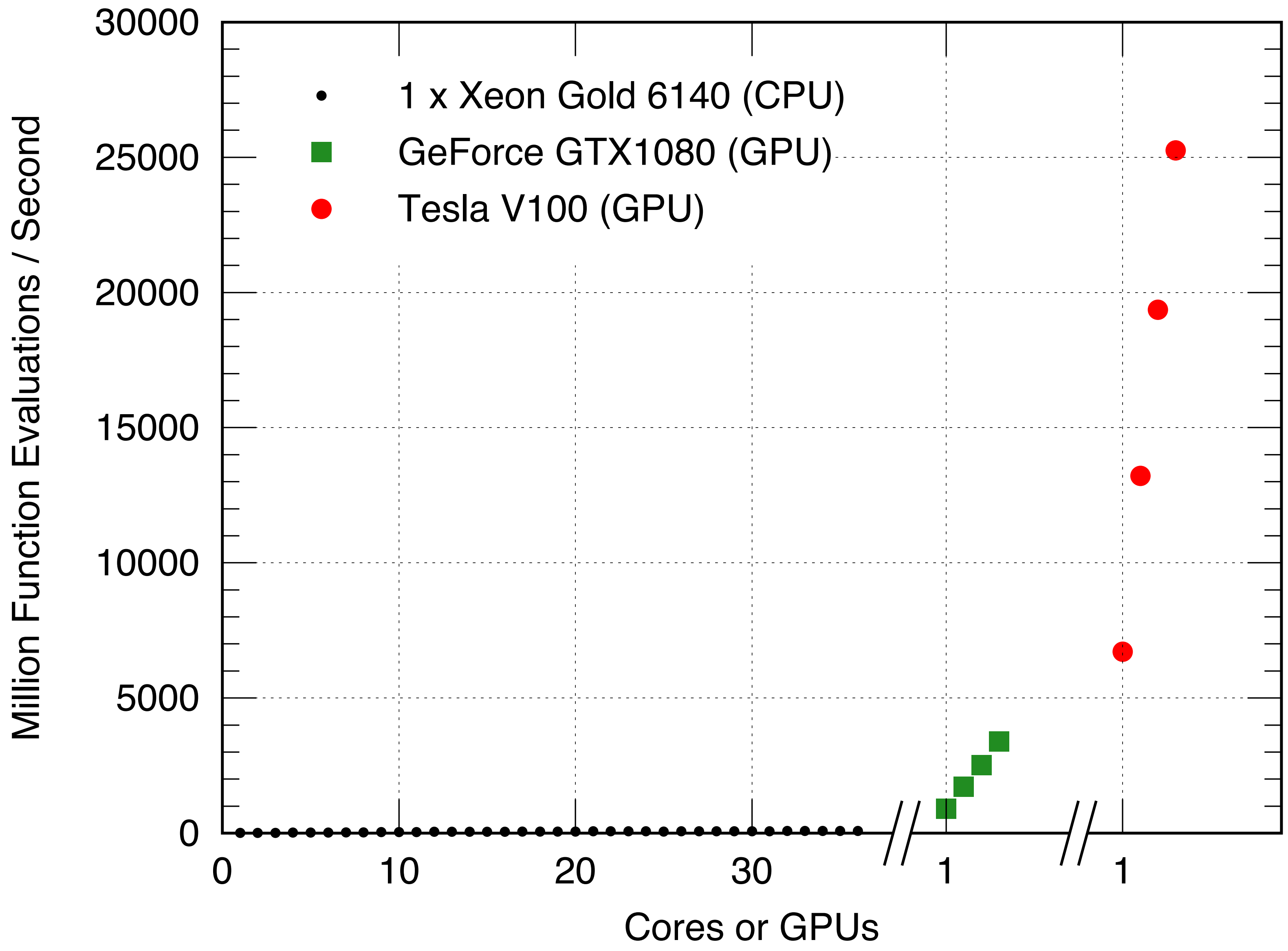
```
int main() {
    integrators::Qmc<double,double,5,integrators::transforms::Korobov<3>::type> integrator;
    integrator.minn = 100000000; // (optional) lattice size
    integrators::result<double> result = integrator.integrate(formfactor2L);
    std::cout << "integral = " << result.integral << std::endl;
    std::cout << "error    = " << result.error    << std::endl;
    return 0;
}
```

```
$ nvcc -O3 -arch=sm_70 -std=c++11 -x cu -I../src 102_ff2_demo.cpp -o 102_ff2_demo.out -lgsl -lgslcblas && ./102_ff2_demo.out
integral = 0.27621
error    = 4.49751e-07
```

(Agrees with analytic result)

# Performance (v1.4)

Accuracy limited by number of function evaluations



| Device | M Func. Evals/s | Speedup |
|---|---|---|
| Xeon 6140 | 80.6 | - |
| GTX1080 | 897 | 11x |
| Tesla V100 | 6710 | 83x |

**Note:** Performance gain highly dependent on integrand & hardware

# Performance Improvements (since v1.4)

**v1.5:** Adaptive sampling of sectors, automatic contour def. adjustment

**v1.5.6:** Optimisations in integrand code

**v1.6: New Quasi-Monte Carlo integrator ``Disteval"**

Faster implementation of old integrator ``IntLib"

CPU & GPU: fusion of integration/integrand code

GPU: sum result on GPU, less synchronisation

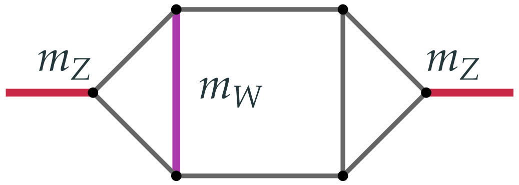
CPU: better utilisation via SIMD instructions (AVX2, FMA)

Parse amplitude coefficients w/GiNaC (supports e.g. partial fractioned input)

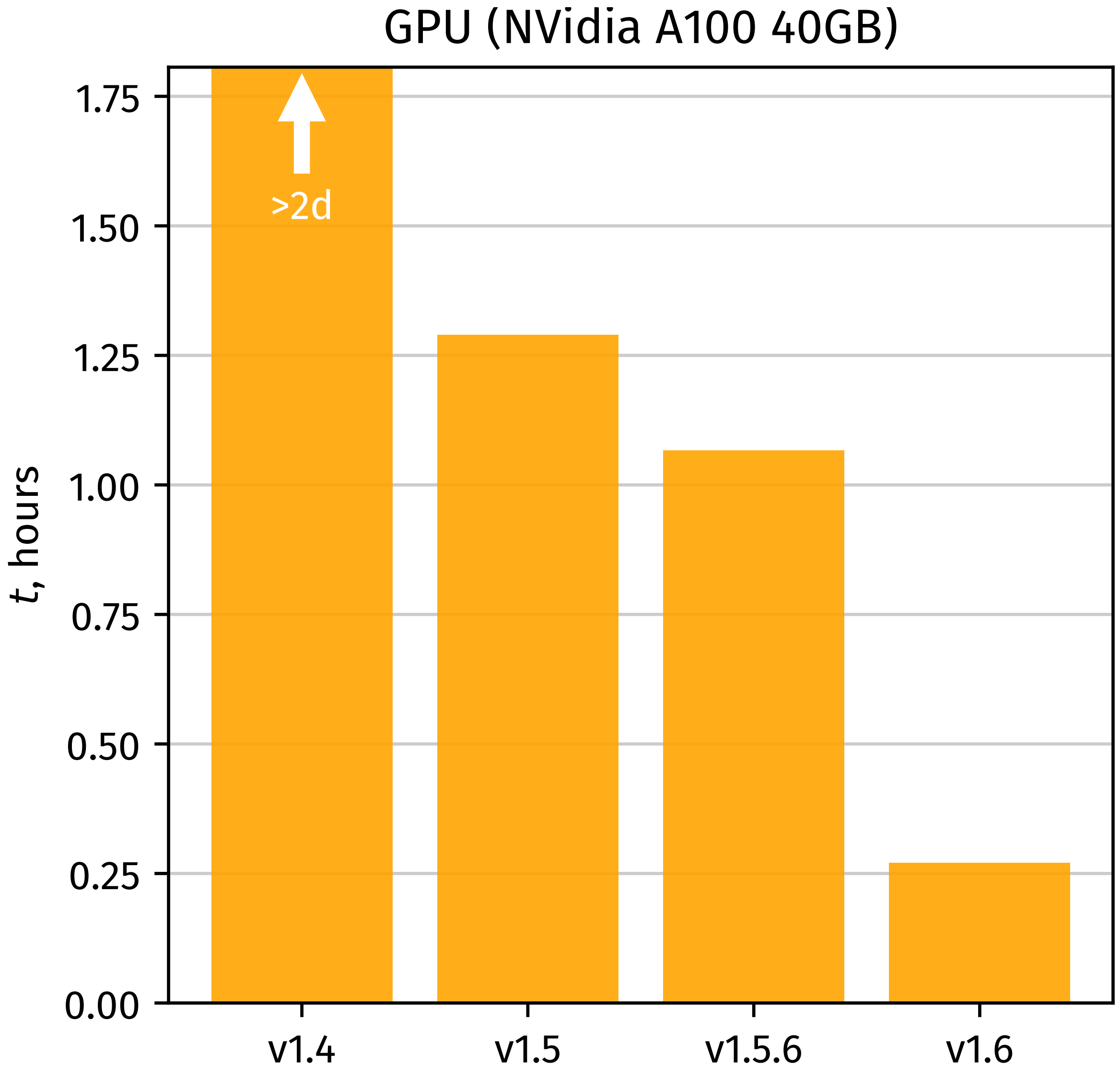
Workers can run on remote machines (via ssh)

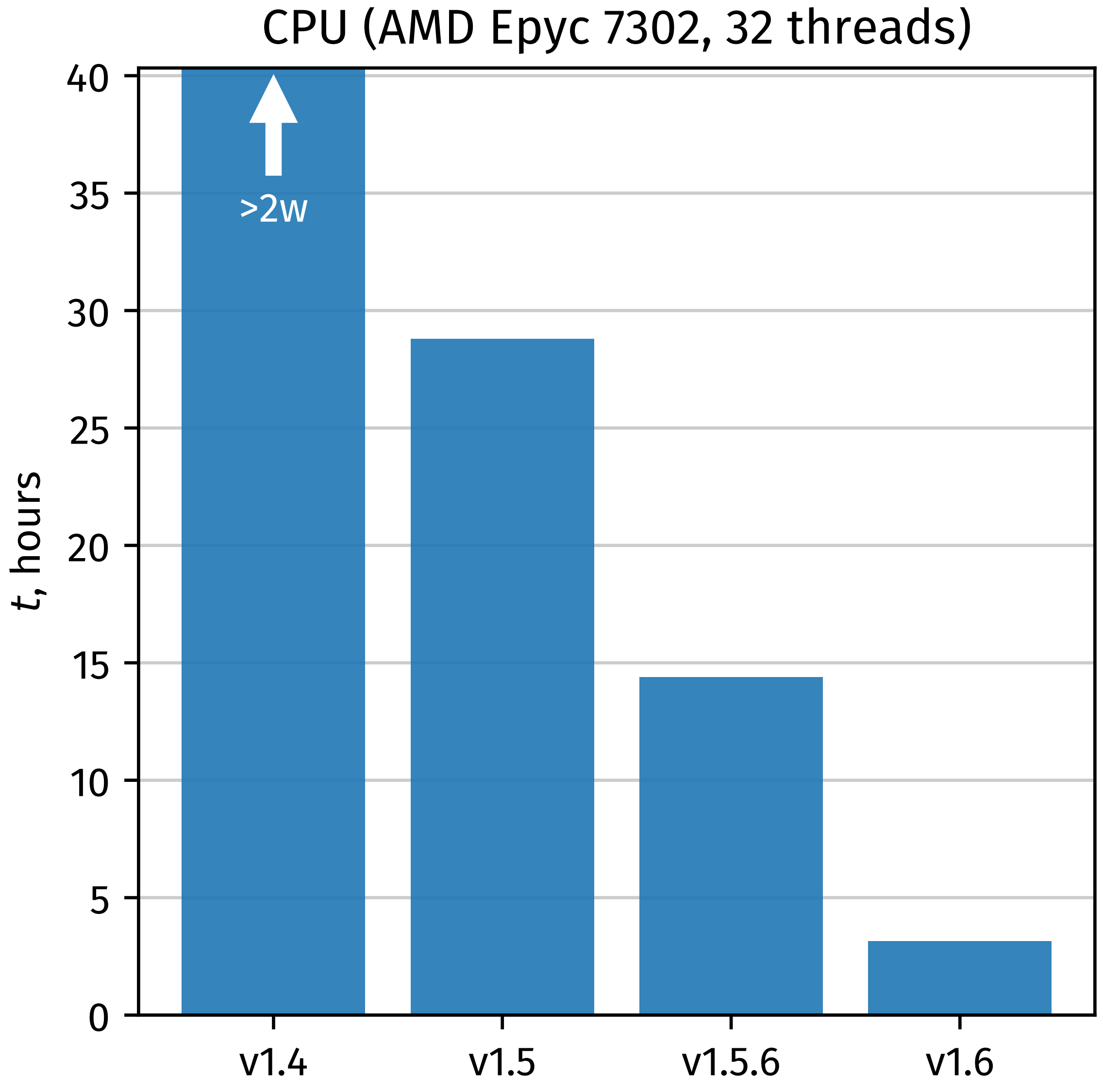
**Does it help?**

# Performance Improvements Impact



7 digits

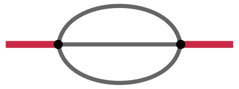
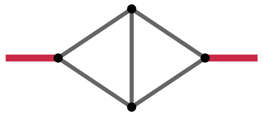
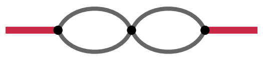

GPU (NVidia A100 40GB)


CPU (AMD Epyc 7302, 32 threads)

Vitaly Magerya (Radcor 2023)

**Yes**

# Adaptive Sampling

| Amplitude term | Naive sampling | Naive error | Better sampling | Better error |
|---|---|---|---|---|
| 1  | $10^6$ samples | $1 \cdot 10^{-6}$ | $\frac{1}{2} \cdot 10^6$ samples | $2 \cdot 10^{-6}$ |
| 10  | $10^6$ samples | $10 \cdot 10^{-6}$ | $\frac{1}{2} \cdot 10^6$ samples | $20 \cdot 10^{-6}$ |
| 50  | $10^6$ samples | $50 \cdot 10^{-6}$ | $2 \cdot 10^6$ samples | $25 \cdot 10^{-6}$ |
| Total: | $3 \cdot 10^6$ | $51 \cdot 10^{-6}$ | $3 \cdot 10^6$ | $32 \cdot 10^{-6}$ |

[Example assumes integration error = $1/n$]

pySecDec now automatically optimizes the total integration time based on

- how fast each integral can be sampled,
- how well it converges,
- how large its coefficient is.

⟹ Automatic *speedup for amplitudes* (weighted sums of integrals).

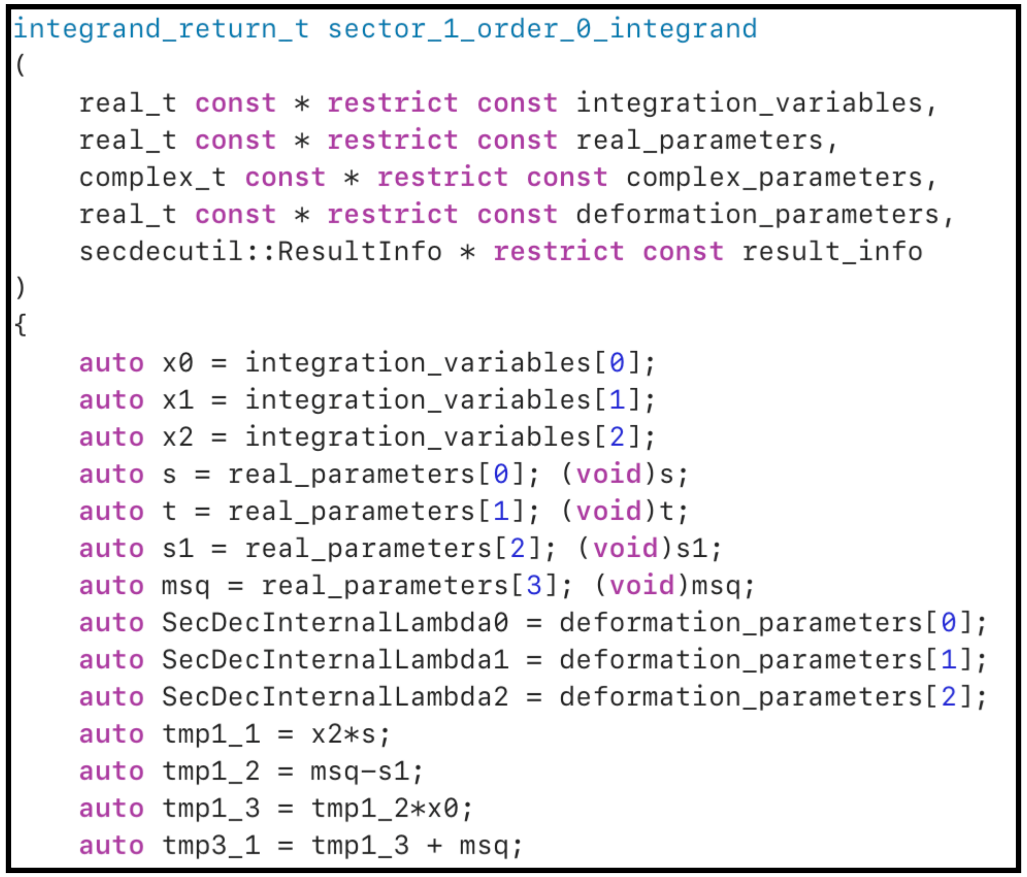
⟹ Automatic speedup for single integrals too (sums of sectors).

⟹ Already used in 2-loop $gg \to ZZ$ (talk by Bakul Agarwal), $gg \to ZH$ (2011.12325), $gg \to \gamma\gamma$ (1911.09314), and $H$ + jet (1802.00349).

Slide:Vitaly Magerya (Radcor 2023)

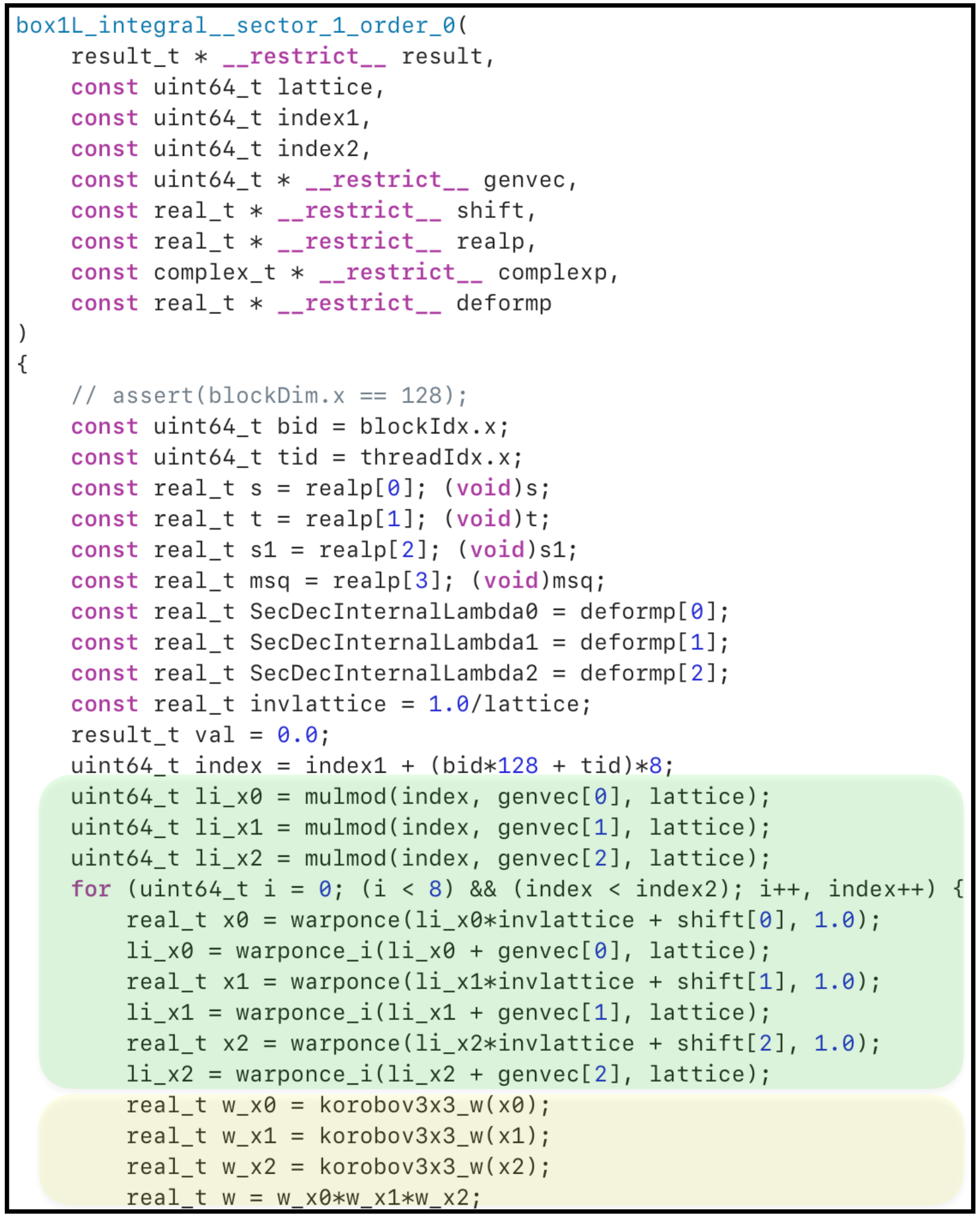
# Fusion of Integration/Integrand Code

## IntLib (old style)



```
integrand_return_t sector_1_order_0_integrand
(
    real_t const * restrict const integration_variables,
    real_t const * restrict const real_parameters,
    complex_t const * restrict const complex_parameters,
    real_t const * restrict const deformation_parameters,
    secdecutil::ResultInfo * restrict const result_info
)
{
    auto x0 = integration_variables[0];
    auto x1 = integration_variables[1];
    auto x2 = integration_variables[2];
    auto s = real_parameters[0]; (void)s;
    auto t = real_parameters[1]; (void)t;
    auto s1 = real_parameters[2]; (void)s1;
    auto msq = real_parameters[3]; (void)msq;
    auto SecDecInternalLambda0 = deformation_parameters[0];
    auto SecDecInternalLambda1 = deformation_parameters[1];
    auto SecDecInternalLambda2 = deformation_parameters[2];
    auto tmp1_1 = x2*s;
    auto tmp1_2 = msq-s1;
    auto tmp1_3 = tmp1_2*x0;
    auto tmp3_1 = tmp1_3 + msq;
```

## Disteval (new style)



```
box1L_integral__sector_1_order_0(
    result_t * __restrict__ result,
    const uint64_t lattice,
    const uint64_t index1,
    const uint64_t index2,
    const uint64_t * __restrict__ genvec,
    const real_t * __restrict__ shift,
    const real_t * __restrict__ realp,
    const complex_t * __restrict__ complexp,
    const real_t * __restrict__ deformp
)
{
    // assert(blockDim.x == 128);
    const uint64_t bid = blockIdx.x;
    const uint64_t tid = threadIdx.x;
    const real_t s = realp[0]; (void)s;
    const real_t t = realp[1]; (void)t;
    const real_t s1 = realp[2]; (void)s1;
    const real_t msq = realp[3]; (void)msq;
    const real_t SecDecInternalLambda0 = deformp[0];
    const real_t SecDecInternalLambda1 = deformp[1];
    const real_t SecDecInternalLambda2 = deformp[2];
    const real_t invlattice = 1.0/lattice;
    result_t val = 0.0;
    uint64_t index = index1 + (bid*128 + tid)*8;
    uint64_t li_x0 = mulmod(index, genvec[0], lattice);
    uint64_t li_x1 = mulmod(index, genvec[1], lattice);
    uint64_t li_x2 = mulmod(index, genvec[2], lattice);
    for (uint64_t i = 0; (i < 8) && (index < index2); i++, index++) {
        real_t x0 = warponce(li_x0*invlattice + shift[0], 1.0);
        li_x0 = warponce_i(li_x0 + genvec[0], lattice);
        real_t x1 = warponce(li_x1*invlattice + shift[1], 1.0);
        li_x1 = warponce_i(li_x1 + genvec[1], lattice);
        real_t x2 = warponce(li_x2*invlattice + shift[2], 1.0);
        li_x2 = warponce_i(li_x2 + genvec[2], lattice);
        real_t w_x0 = korobov3x3_w(x0);
        real_t w_x1 = korobov3x3_w(x1);
        real_t w_x2 = korobov3x3_w(x2);
        real_t w = w_x0*w_x1*w_x2;
```

### Main advantages:

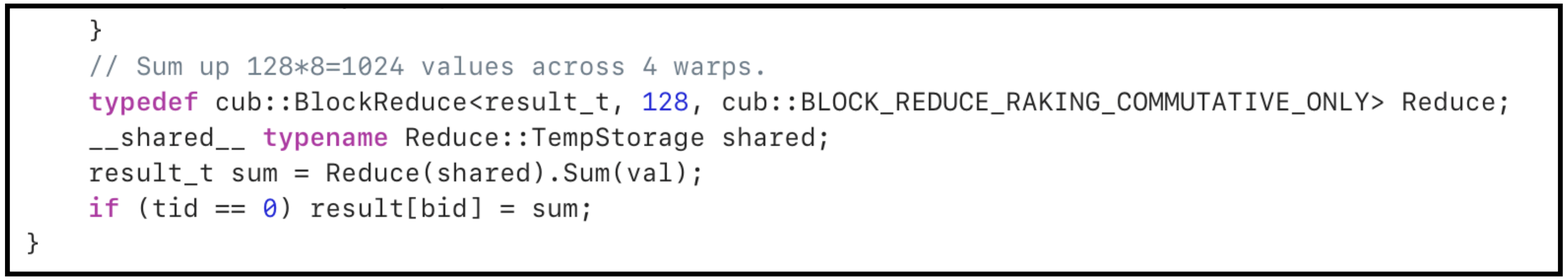
Reduce calls to mulmod

Fuse integral transforms into integrand (less overhead)

# Sum Results on Device

**Inside integrand**

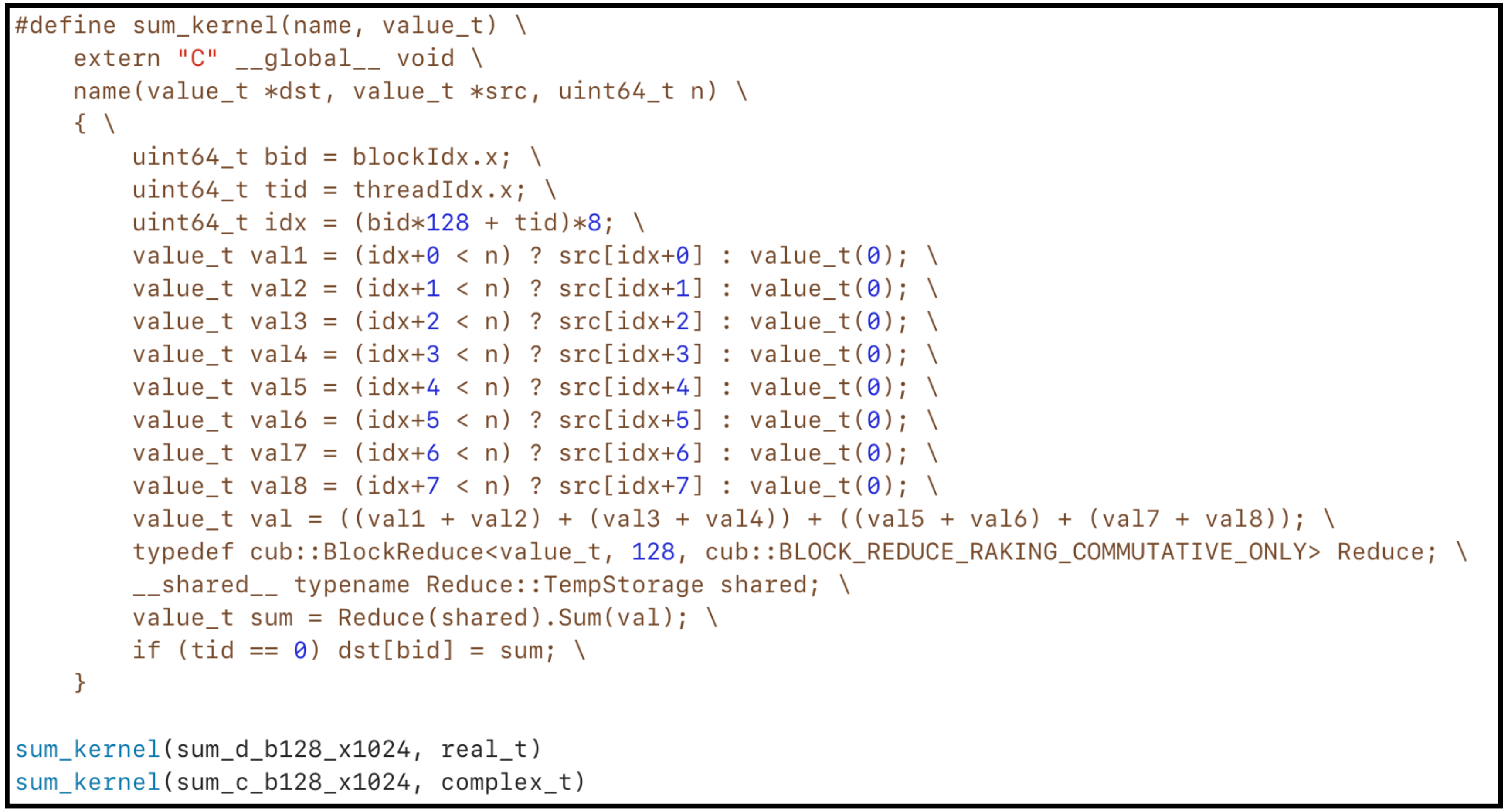
Partial reduction of result



```
    }
    // Sum up 128*8=1024 values across 4 warps.
    typedef cub::BlockReduce<result_t, 128, cub::BLOCK_REDUCE_RAKING_COMMUTATIVE_ONLY> Reduce;
    __shared__ typename Reduce::TempStorage shared;
    result_t sum = Reduce(shared).Sum(val);
    if (tid == 0) result[bid] = sum;
}
```

**Outside integrand**

Use additional sum_kernel to complete the reduction on device

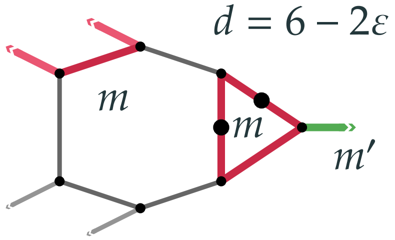


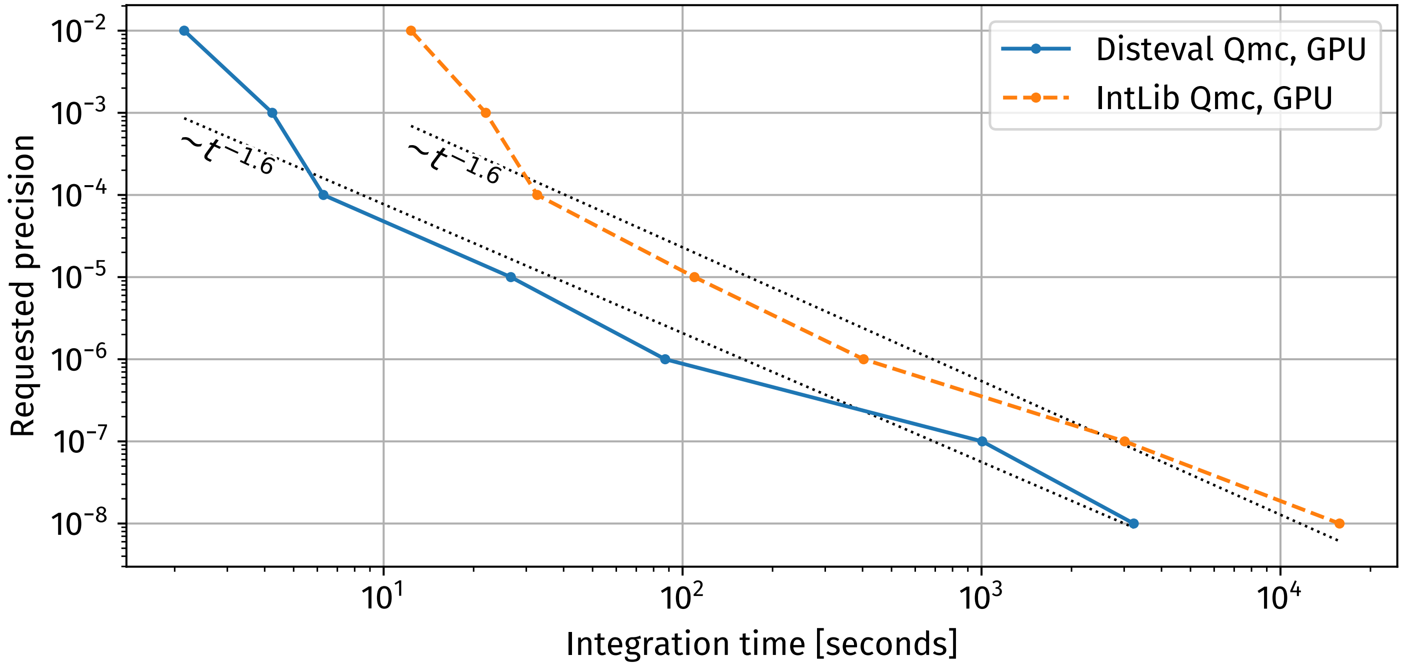
```
#define sum_kernel(name, value_t) \
    extern "C" __global__ void \
    name(value_t *dst, value_t *src, uint64_t n) \
    { \
        uint64_t bid = blockIdx.x; \
        uint64_t tid = threadIdx.x; \
        uint64_t idx = (bid*128 + tid)*8; \
        value_t val1 = (idx+0 < n) ? src[idx+0] : value_t(0); \
        value_t val2 = (idx+1 < n) ? src[idx+1] : value_t(0); \
        value_t val3 = (idx+2 < n) ? src[idx+2] : value_t(0); \
        value_t val4 = (idx+3 < n) ? src[idx+3] : value_t(0); \
        value_t val5 = (idx+4 < n) ? src[idx+4] : value_t(0); \
        value_t val6 = (idx+5 < n) ? src[idx+5] : value_t(0); \
        value_t val7 = (idx+6 < n) ? src[idx+6] : value_t(0); \
        value_t val8 = (idx+7 < n) ? src[idx+7] : value_t(0); \
        value_t val = ((val1 + val2) + (val3 + val4)) + ((val5 + val6) + (val7 + val8)); \
        typedef cub::BlockReduce<value_t, 128, cub::BLOCK_REDUCE_RAKING_COMMUTATIVE_ONLY> Reduce; \
        __shared__ typename Reduce::TempStorage shared; \
        value_t sum = Reduce(shared).Sum(val); \
        if (tid == 0) dst[bid] = sum; \
    }

sum_kernel(sum_d_b128_x1024, real_t)
sum_kernel(sum_c_b128_x1024, complex_t)
```

**Main advantages:** Reduce dev ↔ host memcopy

# Profiling (I)





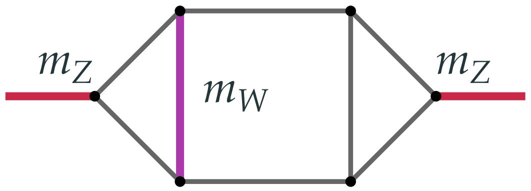
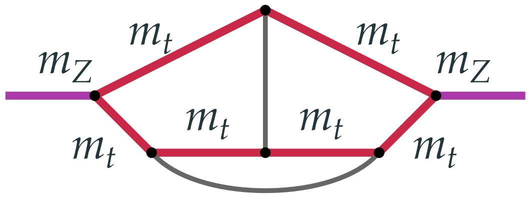
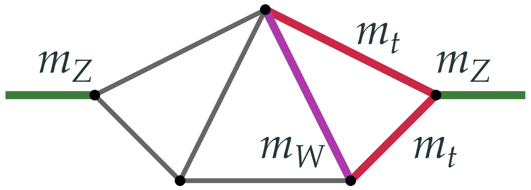
| Integrator \ Accuracy | | $10^{-3}$ | $10^{-4}$ | $10^{-5}$ | $10^{-6}$ | $10^{-7}$ | $10^{-8}$ |
|---|---|---|---|---|---|---|---|
| GPU | DISTEVAL | 4.2 s | 6.3 s | 27 s | 1.5 m | 17 m | 54 m |
| | INTLIB | 22.0 s | 22.0 s | 110 s | 6.7 m | 50 m | 263 m |
| | Speedup | 5.2 | 5.2 | 4.1 | 5.6 | 3.0 | 4.9 |
| CPU | DISTEVAL | 5.1 s | 14 s | 1.6 m | 8.3 m | 57 m | 4.7 h |
| | INTLIB | 20.8 s | 86 s | 14.2 m | 62.2 m | 480 m | 43.1 h |
| | Speedup | 4.1 | 6.1 | 8.7 | 7.5 | 8.4 | 9.2 |

[GPU: NVidia A100 40GB; CPU: AMD EPYC 7F32 with 32 threads]

Vitaly Magerya (Radcor 2023)

# Profiling (II)

pySecDec Disteval *integration times* for 3-loop self-energy integrals:[3]

| Diagram \ Relative precision | | $10^{-3}$ | $10^{-4}$ | $10^{-5}$ | $10^{-6}$ | $10^{-7}$ | $10^{-8}$ |
|---|---|---|---|---|---|---|---|
|  | GPU | 15s | 20s | 40s | 200s | 13m | 50m |
| | CPU | 10s | 50s | 400s | 4000s | 180m | 1200m |
|  $m_t$<br>$m_t$<br>$m_Z$<br>$m_Z$<br>$m_t$<br>$m_t$<br>$m_t$<br>$m_t$ | GPU | 18s | 19s | 30s | 20s | 1.2m | 2m |
| | CPU | 5s | 14s | 60s | 50s | 12m | 16m |
|  | GPU | 6s | 11s | 12s | 30s | 3m | 24m |
| | CPU | 5s | 10s | 50s | 800s | 60m | 800m |

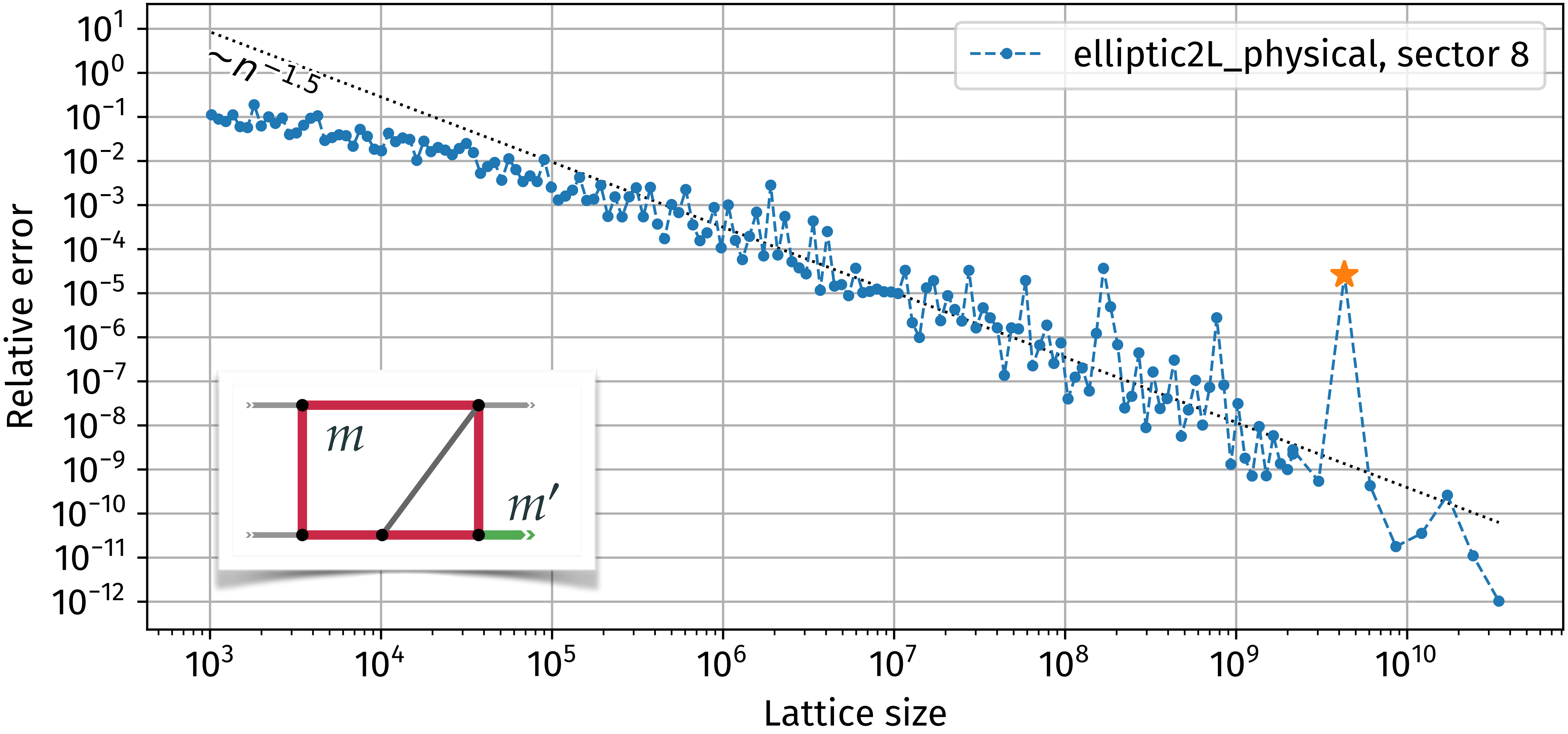
[Same diagrams as in Dubovyk, Usovitsch, Grzanka ’21]

In short: *seconds to minutes per integral* to achieve practical precision.

[GPU: NVidia A100 40GB; CPU: AMD Epyc 7F32 with 32 threads]

# 2. Integration: Algorithmic Improvements

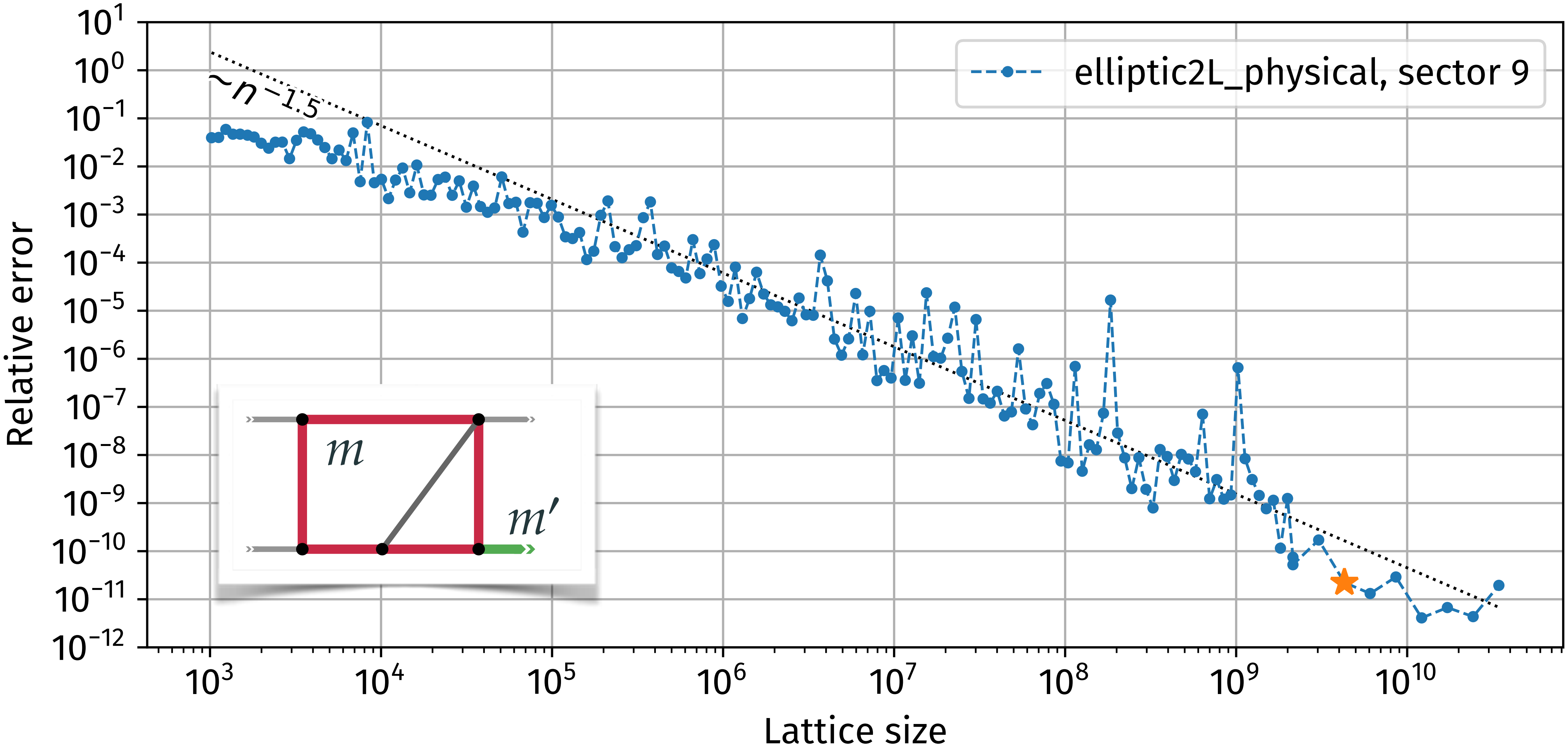
# Quasi-Monte Carlo: Unlucky Lattices



**Good:** Asymptotic error scaling $\sim 1/n^{1.5}$

**Bad:** Huge drop in precision for some "unlucky" lattices
Not consistent across integrands

# Quasi-Monte Carlo: Unlucky Lattices (II)



**Good:** Asymptotic error scaling $\sim 1/n^{1.5}$

**Bad:** Huge drop in precision for some “unlucky” lattices
Not consistent across integrands

# Median Lattice Rules

**Instead:**

Compute $\mathbf{z}$ on-the-fly

1. Choose $R$ random $\mathbf{z} \in \mathrm{Uniform}(0; N-1)$
2. Estimate integral on each lattice
3. Choose lattice with median integral value

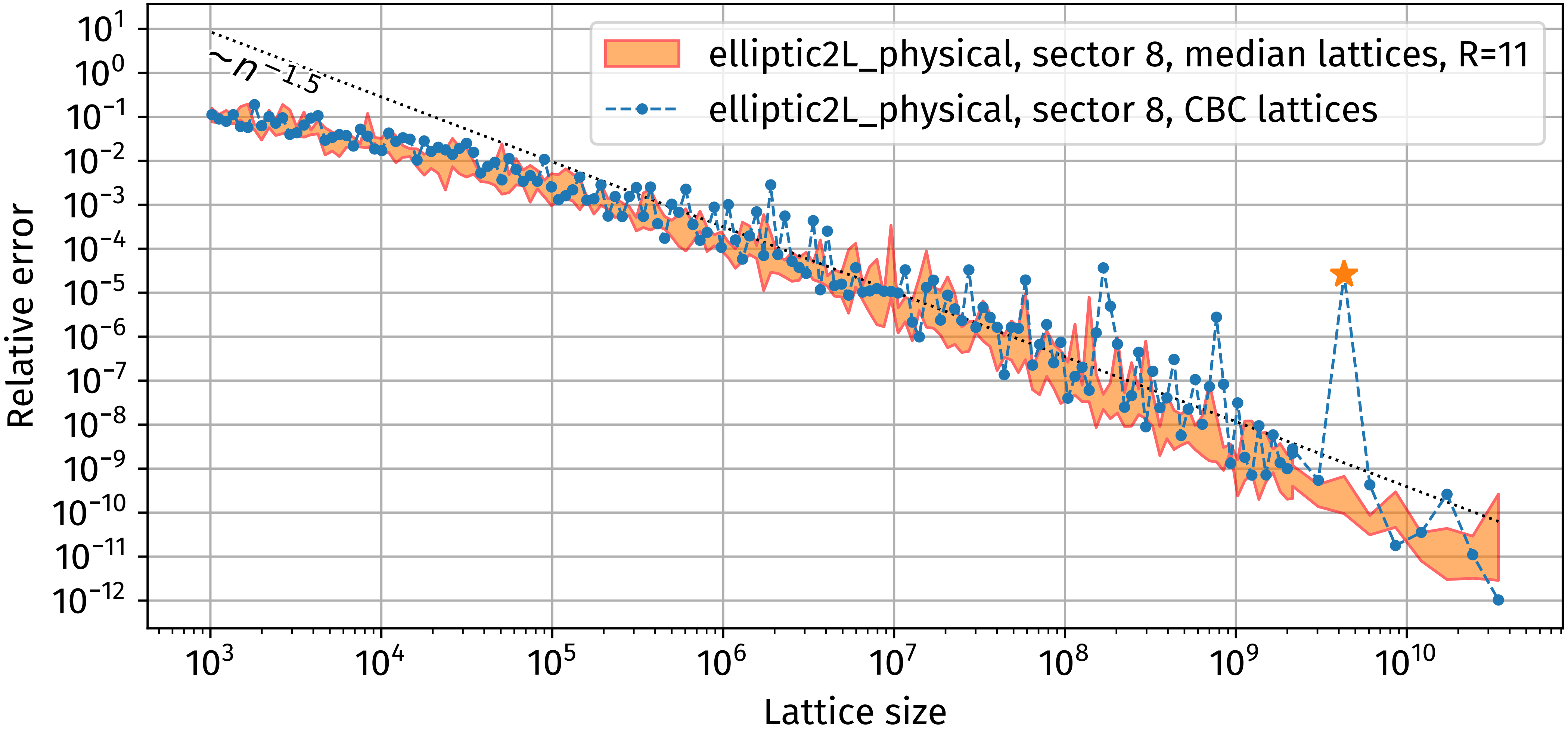
If $\delta_x^{(\alpha)} I(\mathbf{x})$ is square-integrable and periodic

Integration error: $C(\alpha, \varepsilon)/(\rho n)^{\alpha-\epsilon}$

With probability: $1 - \rho^{R+1/2}/4$

$\forall \quad 0 < \varepsilon \;\&\; 0 < \rho < 1$

Goda, L'Ecuyer 22



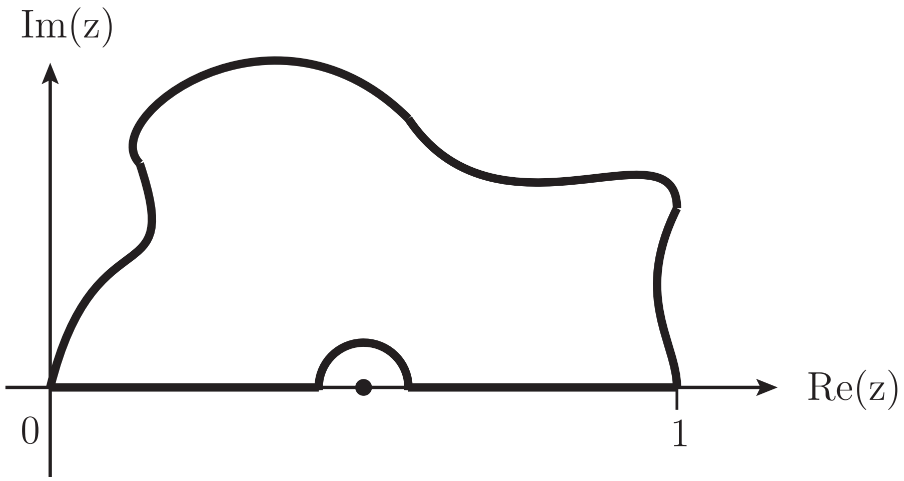
# 3. Contour Deformation

# Neural Networks for Contour Deformation

## Feynman integral (multi-loop/leg):

$$I \sim \int_0^1 [\mathrm{d}\boldsymbol{x}]\, \boldsymbol{x}^{\nu} \frac{[\mathcal{U}(\boldsymbol{x})]^{N-(L+1)D/2}}{[\mathcal{F}(\boldsymbol{x}, \mathbf{s})]^{N-LD/2}}$$

Must deform contour to avoid poles on real axis



Feynman prescription $\mathcal{F} \to \mathcal{F} - i\delta$ tells us how to do this

Expand $\mathcal{F}(z = \boldsymbol{x} - i\boldsymbol{\tau})$ around $\boldsymbol{x}$: $\mathcal{F}(z) = \mathcal{F}(\boldsymbol{x}) - i \sum_j \tau_j \frac{\partial \mathcal{F}(\boldsymbol{x})}{\partial x_j} + \mathcal{O}(\tau^2)$

## Old Method

$\tau_j = \lambda_j\, x_j(1 - x_j)\frac{\partial \mathcal{F}(\boldsymbol{x})}{\partial x_j}$ with small constants $\lambda_j > 0$

Soper 99; Binoth, Guillet, Heinrich, Pilon, Schubert 05; Nagy, Soper 06; Anastasiou, Beerli, Daleo 07; Beerli 08; Borowka, Carter, Heinrich 12; Borowka 14;…

## New Method

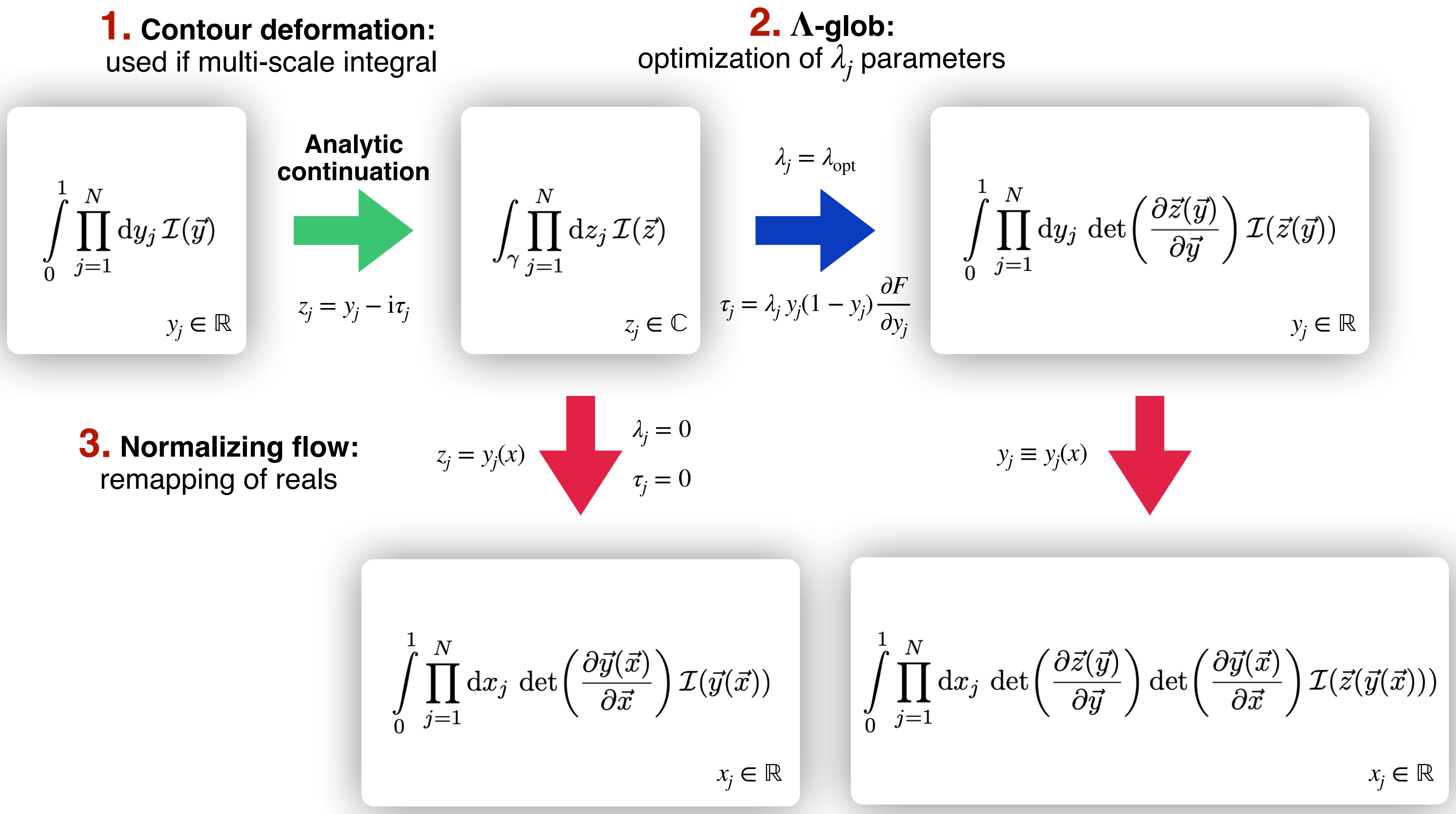
Generalise $\lambda_j \to \lambda_j(\boldsymbol{x})$ and use Neural Network (Normalizing Flows) to pick contour

Winterhalder, Magerya, Villa, SJ, Kerner, Butter, Heinrich, Plehn 22

# Neural Networks for Contour Deformation (II)

Normalizing Flows consist of a series of (trainable) bijective mappings for which we can efficiently compute the Jacobian

## Procedure



**1. Contour deformation:**
used if multi-scale integral

**2. $\Lambda$-glob:**
optimization of $\lambda_j$ parameters

**3. Normalizing flow:**
remapping of reals

$$\int_0^1 \prod_{j=1}^N \mathrm{d}y_j\, \mathcal{I}(\vec{y})$$

$y_j \in \mathbb{R}$

**Analytic continuation**

$z_j = y_j - \mathrm{i}\tau_j$

$$\int_\gamma \prod_{j=1}^N \mathrm{d}z_j\, \mathcal{I}(\vec{z})$$

$z_j \in \mathbb{C}$

$\lambda_j = \lambda_{\mathrm{opt}}$

$\tau_j = \lambda_j\, y_j(1-y_j)\dfrac{\partial F}{\partial y_j}$

$$\int_0^1 \prod_{j=1}^N \mathrm{d}y_j\, \det\left(\frac{\partial \vec{z}(\vec{y})}{\partial \vec{y}}\right) \mathcal{I}(\vec{z}(\vec{y}))$$

$y_j \in \mathbb{R}$

$z_j = y_j(x)$

$\lambda_j = 0$

$\tau_j = 0$

$y_j \equiv y_j(x)$

$$\int_0^1 \prod_{j=1}^N \mathrm{d}x_j\, \det\left(\frac{\partial \vec{y}(\vec{x})}{\partial \vec{x}}\right) \mathcal{I}(\vec{y}(\vec{x}))$$

$x_j \in \mathbb{R}$

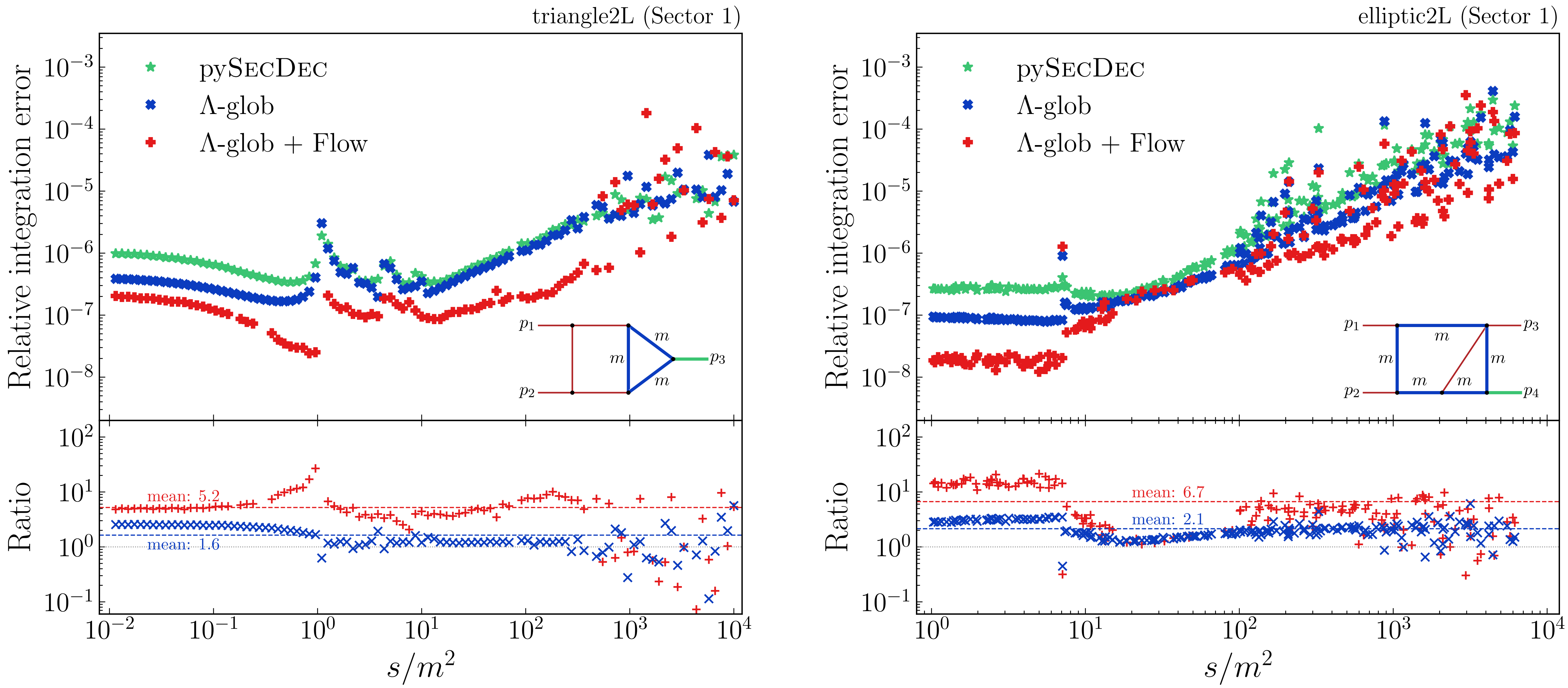
$$\int_0^1 \prod_{j=1}^N \mathrm{d}x_j\, \det\left(\frac{\partial \vec{z}(\vec{y})}{\partial \vec{y}}\right) \det\left(\frac{\partial \vec{y}(\vec{x})}{\partial \vec{x}}\right) \mathcal{I}(\vec{z}(\vec{y}(\vec{x})))$$

$x_j \in \mathbb{R}$

Loss: $L = L_{\mathrm{MC}} + L_{\mathrm{sign}}$ constructed to minimise variance without crossing poles

# Neural Networks for Contour Deformation (III)

Applied to several 1 & 2-loop Feynman Integrals with multiple masses/thresholds using tensorflow



Proof of principle that Machine Learning can help to find improved contours and reduce variance, still a tradeoff between training time/ integrating time

# 4. Expansions: Method of Regions

# Method of Regions

Consider expanding an integral about some limit:

$p_i^2 \sim \lambda Q^2$ , $p_i \cdot p_j \to \lambda Q^2$ or $m^2 \sim \lambda Q^2$ for $\lambda \to 0$

**Issue:** integration and series expansion do not necessarily commute

## Method of Regions

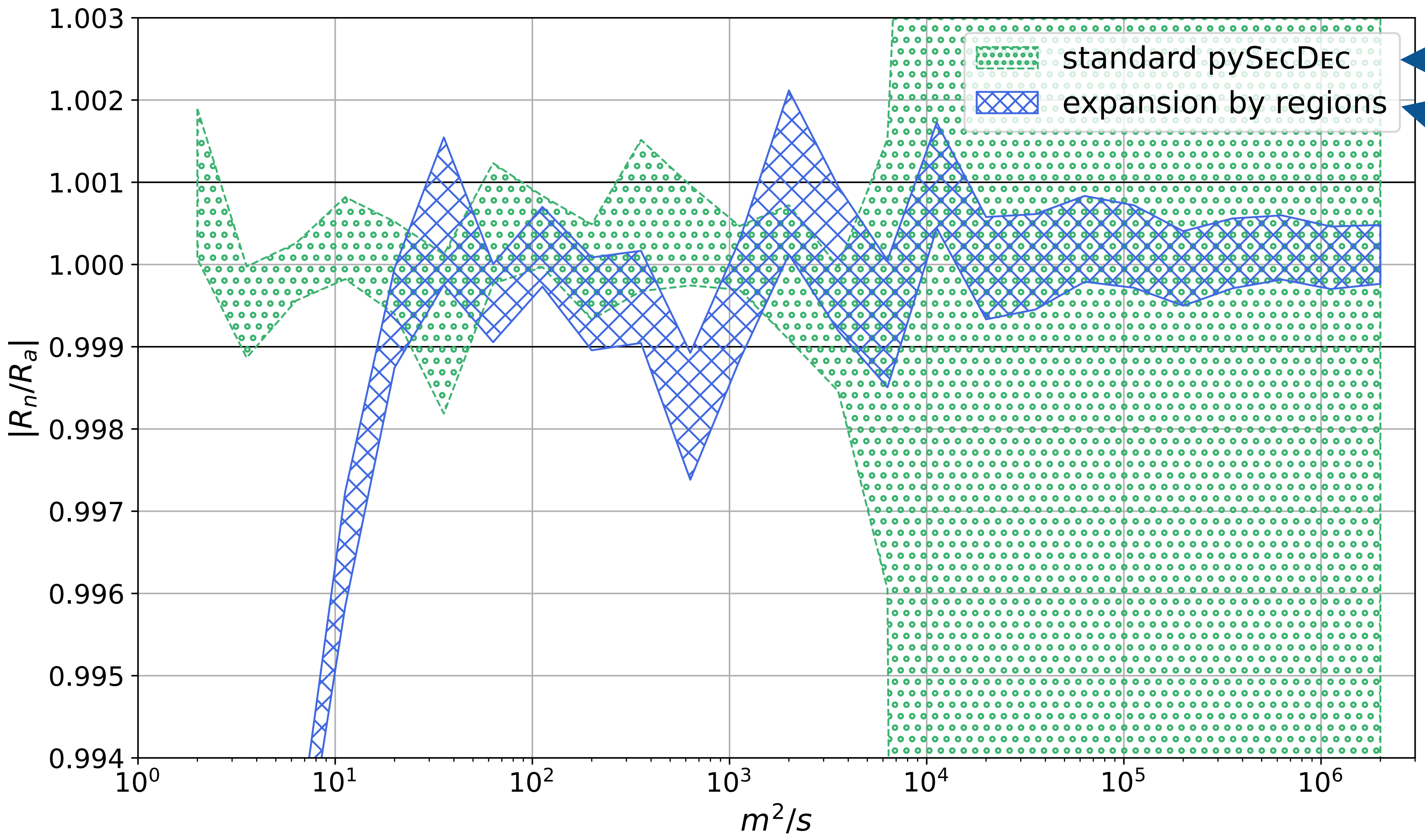
$$I(\mathbf{s}) = \sum_R I^{(R)}(\mathbf{s}) = \sum_R T_{\mathbf{t}}^{(R)} I(\mathbf{s})$$

1. Split integrand up into regions ($R$)
2. Series expand each region in $\lambda$
3. Integrate each expansion over the whole integration domain
4. Discard scaleless integrals (= 0 in dimensional regularisation)
5. Sum over all regions

Smirnov 91; Beneke, Smirnov 97; Smirnov, Rakhmetov 99; Pak, Smirnov 11; Jantzen 2011; …

# Applying Expansion by Regions

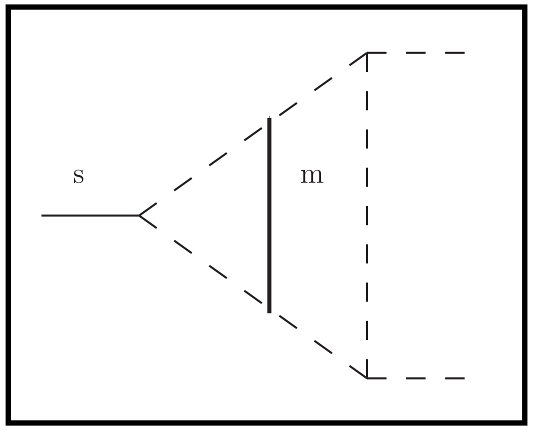
Ratio of the finite $\mathcal{O}(\epsilon^0)$ piece of numerical result $R_n$ to the analytic result $R_a$



**(at large $m^2/s$)**

**~1 day**

**~2 min**



For large ratio of scales ($m^2/s$) the EBR result is **faster** & **easier** to integrate

# Challenges and Opportunities

### Frontiers

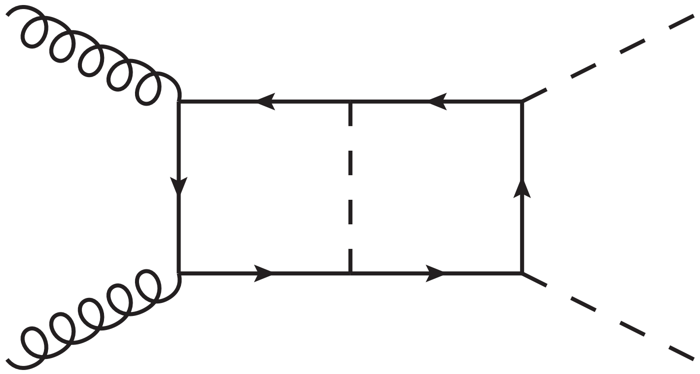
* $2 \to 2$ **@ 2-loop** : fine (e.g. HH, HJ, ZZ, ZH)

  + masses (e.g. EW corrections) - suitable

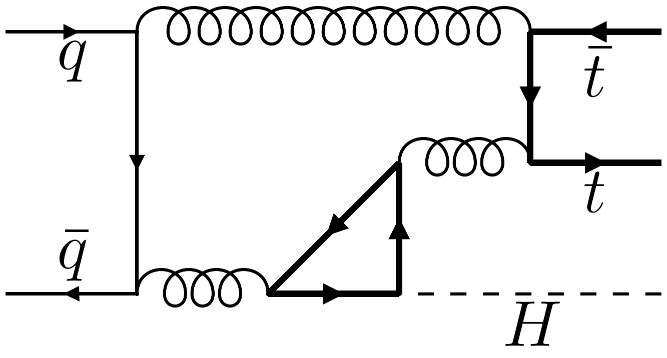
  + large hierarchies (e.g. small $m_b$ , large $s$, thresholds)

* $2 \to 3$ **@ 2-loop** : challenging (high dim phase-space)

* **3-loop+** : suitable, less explored



WIP: Gudrun Heinrich, SJ, Matthias Kerner, Tom Stone, Augustin Vestner



WIP: V. Magerya, G. Heinrich, SJ, M. Kerner, S. Klein, J. Lang, A. Olsson

### Opportunities

1. Improvements in algorithm & implementation
2. Smarter numerical integration routines
3. Improved contour deformation
4. Expansions

# Conclusion

## Updates

- Recent code improvements give ~3-5x speed up
- Performance gains from both **algorithmic improvements** and **optimisation**
- Median lattice rules: lattices of unlimited size, smaller fluctuations in error
- Quite general input can be evaluated, using mixture of CPU/GPU codes

## Applications

- Various processes at $2 \to 2$ with many masses @ 2-loops
- Various applications to 3-loop and 4-loop problems with limited number of scales
- First applications to $2 \to 3$ amplitudes @ 2-loops

## Future Resources

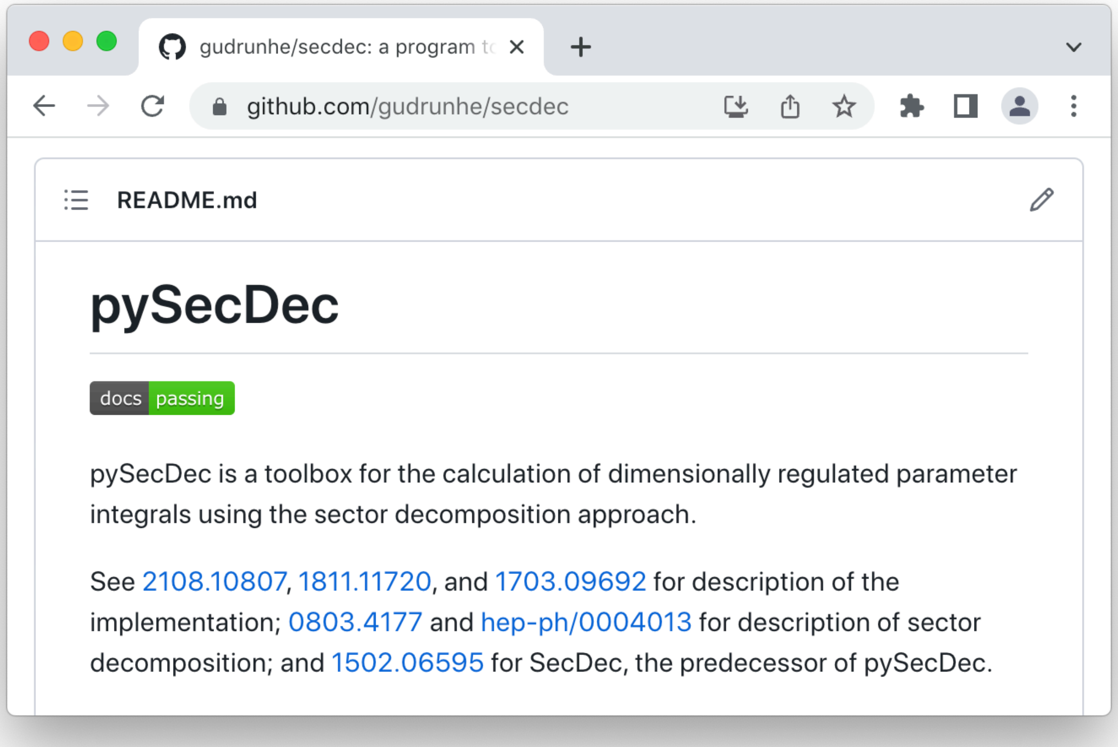
- For our type of numerical integration it is clear that **GPUs are hugely beneficial**
- If this is generally "the way to go" depends a lot on whether other algorithms can be efficiently parallelised (e.g. rational reconstruction, series solutions)

**Thank you for listening!**

# Backup

# pySecDec

**pySecDec:** a program for numerically evaluating dimensionally regulated parameter integrals on CPU or GPU



## pySecDec

pySecDec is a toolbox for the calculation of dimensionally regulated parameter integrals using the sector decomposition approach.

See 2108.10807, 1811.11720, and 1703.09692 for description of the implementation; 0803.4177 and hep-ph/0004013 for description of sector decomposition; and 1502.06595 for SecDec, the predecessor of pySecDec.

Publicly available (Github)

Install with: `python3 -m pip install --user --upgrade pySecDec`

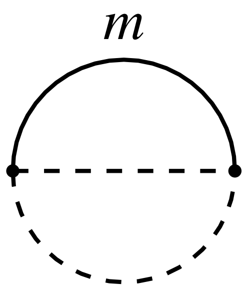
**Other public sector decomposition tools:**

sector_decomposition + CSectors   Bogner, Weinzierl 07; Gluza, Kajda, Riemann, Yundin 10

FIESTA

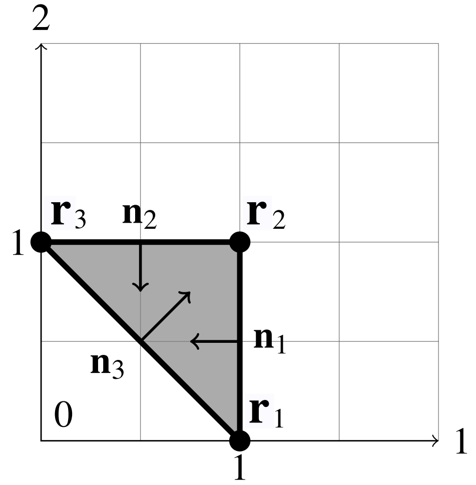
A. Smirnov, V. Smirnov, Tentyukov 08, 09, 13, 15; Smirnov 16; Smirnov, Shapurov, Vysotsky 21

# Sector Decomposition in a Nutshell (III)

$I =$ 

$$= -\Gamma(-1+2\varepsilon)\left(m^2\right)^{1-2\varepsilon}\int_0^\infty \frac{\mathrm{d}x_1\mathrm{d}x_2}{\left(x_1^1x_2^0+x_1^1x_2^1+x_1^0x_2^1\right)^{2-\varepsilon}}.$$

$$\mathbf{r}_1 = \begin{pmatrix}1\\0\end{pmatrix}, \mathbf{r}_2 = \begin{pmatrix}1\\1\end{pmatrix}, \mathbf{r}_3 = \begin{pmatrix}0\\1\end{pmatrix}$$

$\mathcal{N}(I) =$ 

$$= \quad \begin{array}{cccccc} \mathbf{n}_1 = & \begin{pmatrix}-1\\0\end{pmatrix} & \mathbf{n}_2 = & \begin{pmatrix}0\\-1\end{pmatrix} & \mathbf{n}_3 = & \begin{pmatrix}1\\1\end{pmatrix} \\ a_1 = & 1 & a_2 = & 1 & a_3 = & -1 \end{array}$$

For each vertex make the local change of variables

e.g. $\mathbf{r}_1$: $x_1 = y_1^{-1}y_3^1,\ x_2 = y_1^0y_3^1$ , $\mathbf{r}_2 : x_1 = y_1^{-1}y_2^0,\ x_2 = y_1^0y_2^{-1}$, $\mathbf{r}_3 : x_1 = y_2^0y_3^1, x_2 = y_2^{-1}y_3^1$

$$I = -\Gamma(-1+2\varepsilon)\left(m^2\right)^{1-2\varepsilon}\int_0^1 \mathrm{d}y_1\mathrm{d}y_2\mathrm{d}y_3\, \frac{y_1^{-\varepsilon}\,y_2^{-\varepsilon}\,y_3^{-1+\varepsilon}}{(y_1+y_2+y_3)^{2-\varepsilon}}\left[\delta(1-y_2)+\delta(1-y_3)+\delta(1-y_1)\right]$$

Schlenk 2016

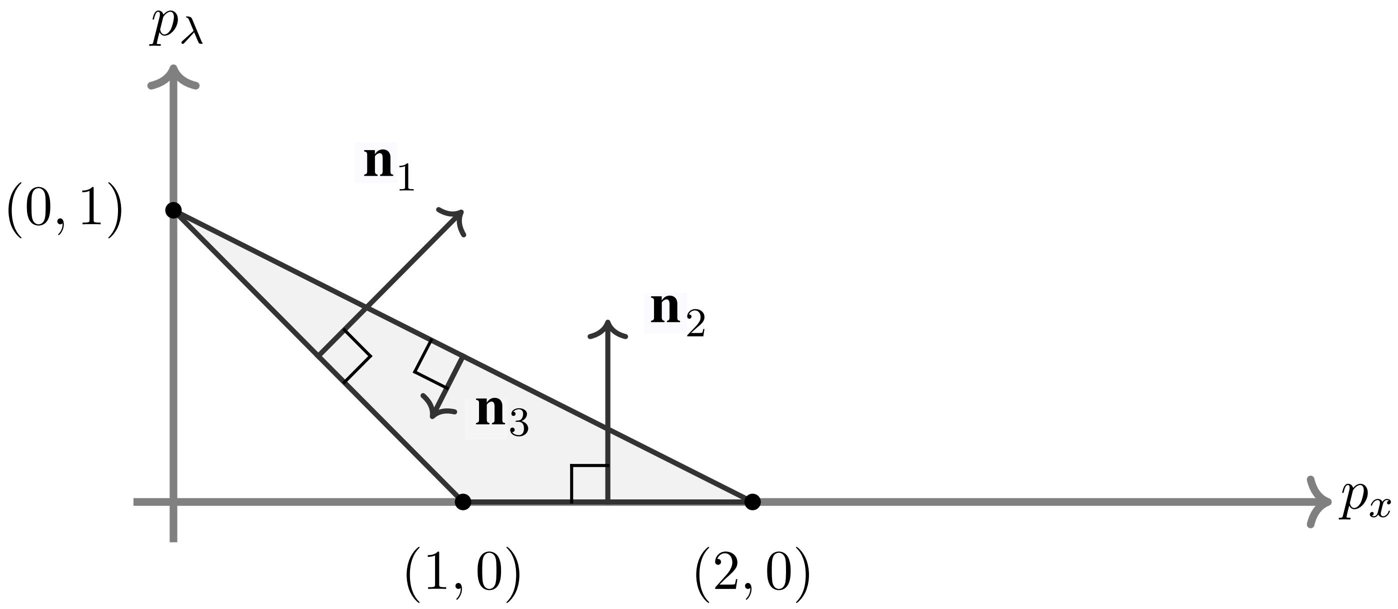
# Finding Regions

$$I \sim \int_{\mathbb{R}^N_{>0}} [\mathrm{d}\boldsymbol{x}]\, \boldsymbol{x}^{\boldsymbol{\nu}} \left(c_i\, \boldsymbol{x}^{\mathbf{r}_i}\right)^t \to \int_{\mathbb{R}^N_{>0}} [\mathrm{d}\boldsymbol{x}]\, \boldsymbol{x}^{\boldsymbol{\nu}} \left(c_i\, \boldsymbol{x}^{\mathbf{r}_i} \lambda^{r_{i,N+1}}\right)^t \to \mathcal{N}^{N+1}$$

Normal vectors w/ positive $\lambda$ component define change of variables $\mathbf{n}_f = (v_1, \ldots, v_N, 1)$

$$\boldsymbol{x} = \boldsymbol{\lambda}^{\mathbf{n}_f}\, \mathbf{y}\,, \qquad \lambda \to \lambda$$

Pak, Smirnov 10; Semenova, A. Smirnov, V. Smirnov 18



$1{,}2 \in F^+$

$3 \notin F^+$

Original integral $I$ may then be approximated as $I = \sum_{f \in F^+} I^{(f)} + \ldots$