New ideas (methods) for UHECR propagation

… and the role of efficient computing techniques

Anatoli Fedynitch
DESY Zeuthen

Efficient computational codes

- Are you sure that your (computational) research won’t change, if your code would run instead of 2h/2 min/40 seconds just **2 seconds** or **tens of milli-seconds**?

- Shan Gao’s case: highly optimized code (semi-analytical approximations where needed, etc.):
  - 5 * (few sec) + 2 (few minutes) parameters
  - Many “local minima” (evaluation time probably a bit too long for MCMC)
  - Need to (pre-)understand physics to set-up proper ranges for grid-scans
  - Can not scan all parameters on fine grids, this would require MCPUh/source

- One of the problems: most radiation calculations are single-core or trivially parallel programs (cluster jobs)

Moores' law or what?

- Some manufacturers present outrageous numbers of floating point performance for their hardware products

PERFORMANCE SPECIFICATION FOR NVIDIA TESLA P100 ACCELERATORS

	P100 for PCIe-Based Servers
Double-Precision Performance	4.7 TeraFLOPS
Single-Precision Performance	9.3 TeraFLOPS
Half-Precision Performance	18.7 TeraFLOPS

- Can I use this somehow in my calculations?

- **You can not,** if you write something like:

Compiler doesn't know N-iterations during compile-time

```
for (int i=0; i < get_upper_idx(); ++i){
    ...
    x[i] = x[i]*x[i] + y[i,i];
    ...
}
```

```
int IMAX = 100000;

for (int i=0; i < IMAX; ++i){
    ...
    x[i] = calculate_something();
    if (x[i] < 5)
        break;
    else ...
}
```

Termination condition depends on intermediate result

Usually, a simple branch in the loop is enough to not optimize

Why do we need another propagation code?

Propagation Codes

- Multi particle approach → Fokker Planck equations

galactic

GALPROP DRAGON

PICARD

extragalactic

TransportCR **DINT**

+ many private codes

- Single particle approach → Particle tracking

galactic

extragalactic

CRPropa **SimProp**

CRT **Hermes / EleCa**

Slide by David Walz (CRPropa 3)

- We (NEUCOS) want to use a self-consistent source-propagation model
  - Nuclear/interaction models
- Flexible and easy to use (by Master/PhD students)
- It has to be super-fast (parameter scans)
- Our code is called PriNCe. We develop it together with **J. Heinze**.
- Precursor for development of high-precision/high-speed non-linear transport equation solvers

DESY

Propagation of nuclei

Solve in comoving number density

$$Y^{A_i}(E_N, z) = \frac{n^{A_i}}{(1+z)^3}$$

Adiabatic + pair production losses

Absorption + Re-injection

Injection from sources

$$-(1+z)H(z)\,\partial_z Y^{A_i}(E_N,z) = A_i^2 \partial_{E_N}(H(z) E_N Y^{A_i}(E_N,z)) + A_i \partial_{E_N}(b_{e^+e^-}(E_N,z,A_i) Y(E_N,z)) -\Gamma_{A_i\gamma}(z) Y^{A_i}(E_N,z) + \sum_{A_j} \int_{E_N}^{\infty} \mathrm{d}E'_N \Gamma^{A_j \to A_i}_{A\gamma}(z) Y^{A_j}(E_N,z) + \mathcal{L}_{\rm CR}(E_N,z)$$

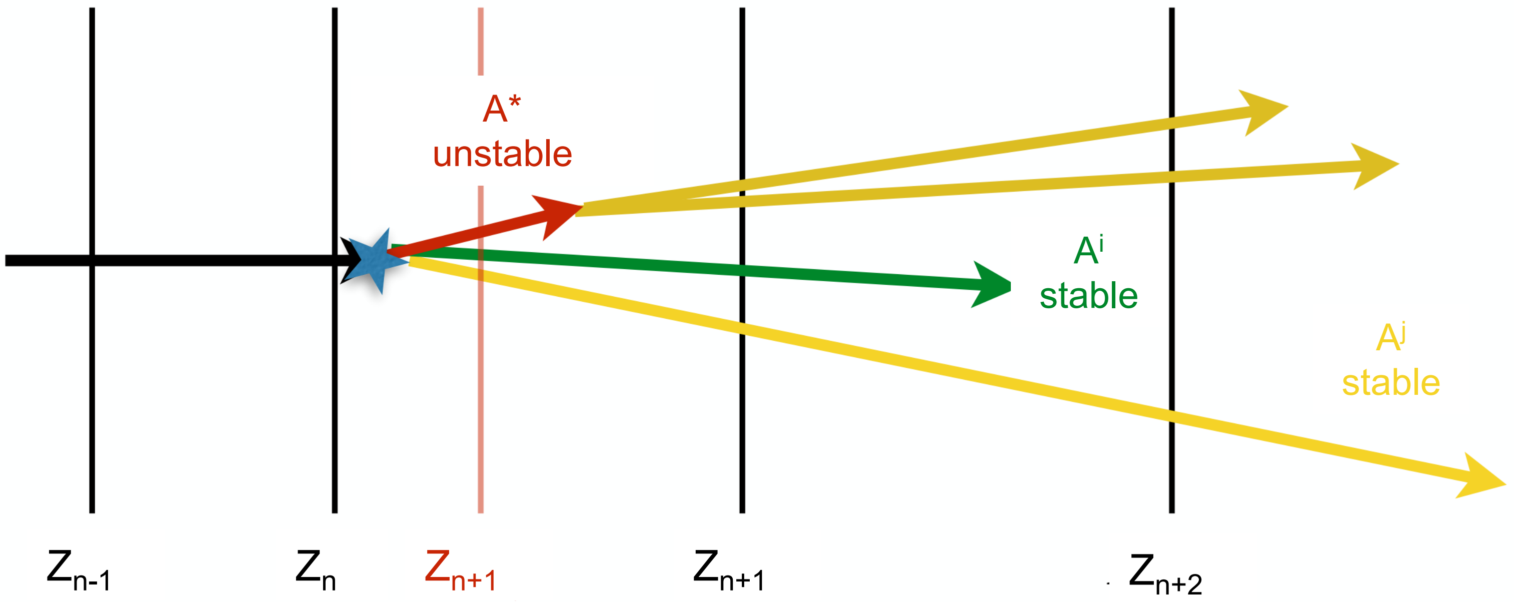
Naïve approach: Many nuclear species (worst case ~400 up to iron) * ~60 energy bins = eqn. system of order 24000

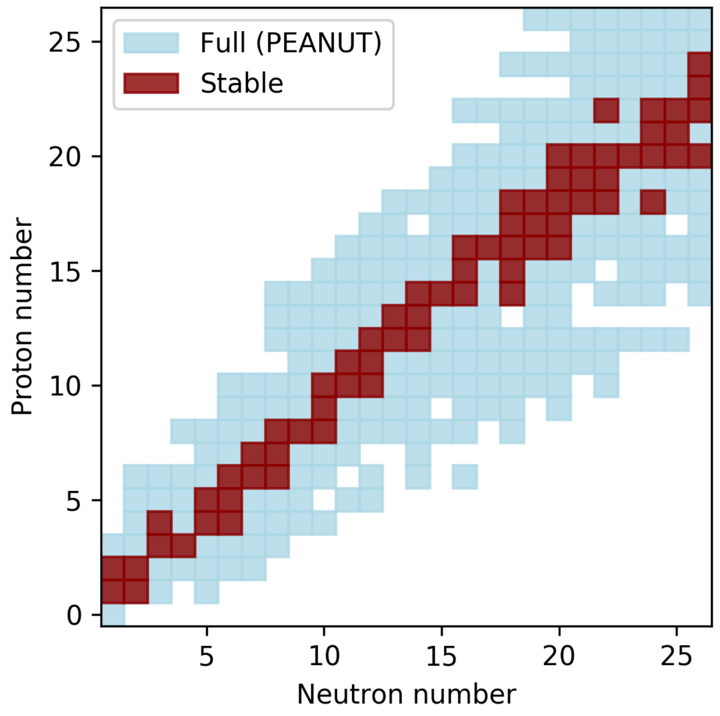
Reduction of order (semi-analytical approximations)

$$-\Gamma_{A_i\gamma}(z)Y^{A_i}(E_N,z)+\sum_{A_j}\int_{E_N}^{\infty}\mathrm{d}E'_N\Gamma_{A\gamma}^{A_j\to A_i}(z)Y^{A_j}(E_N,z)$$

Most species will decay into more stable nuclei during the first integration step in redshift

One origin of stiffness





Parallel, simultaneous computation of rates

$$-\Gamma_{A_i\gamma}(z)Y^{A_i}(E_N,z)+\sum_{A_j}\int_{E_N}^{\infty}\mathrm{d}E'_N\Gamma_{A\gamma}^{A_j\to A_i}(z)Y^{A_j}(E_N,z)$$

Rates Γ have to be recomputed every time the photon density changes:

(84 absorption + 400 inclusive cross sections (channels)) *
* 60 energy bins ~ 30000 double integrals

$$\Gamma_{A_i\gamma}^{A_i\to A_j}(E_i,z)=\frac{1}{2}\frac{m_{A_i}^2}{E_i^2}\int_{\frac{\epsilon_{\rm th}m_p}{2E}}^{\infty}d\epsilon\frac{n_\gamma(\epsilon,z)}{\epsilon^2}\int_0^{2E\epsilon/m_{A_i}}d\epsilon_r\epsilon_r\sigma_{A_i\gamma}^{A_i\to A_j}(\epsilon_r)$$

Use (old QED) trick first and get rid of second integral, g precomputable (NEUCOSMA employs these methods)

$$\Gamma_{A_i\gamma}^{A_i\to A_j}(E_i,z)=\int_{\epsilon_{\rm th}}^{\infty}\mathrm{d}\epsilon\; n_\gamma(\epsilon,z)g_{i\to j}(\epsilon,E_i)=(\mathbf{G}\times\vec{n_\gamma}(z))_i$$

Simple convolution as matrix expression

$$c(E_i)=\int_{E_i}^{\infty}\mathrm{d}E'b(E_i,E')a(E')$$

$$\approx\sum_{j=E_i}^{E_N}\Delta E'_j\; b(E_i,E'_j)a(E'_j)=\sum_j B_{ij}a_j$$

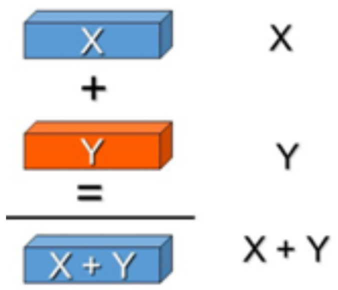
For any order of c $\quad\vec{c}=\mathbf{B}\times\vec{a}$

Well,
matrices … sure …
I write loops
…obviously

Ordinary loops and calls to a Linear Algebra library are not the same

Principle of vectorization

```
double *x, *y, *z;
for (i=0;  i<n;  i++)      z[i] = x[i] + y[i];
```



> Features you might get:
  - 2-8 Float operations per clock instead of 1
  - Addition + multiplication in 1 clock instead of 2
  - Coalesced memory access (higher RAM/Cache FPU bandwidth)
  - SMP (Multicore), easy GPU, …

> We are not computer scientist and we **don't** want to
  - spend a significant fraction of life-time to study all these new technologies/APIs
  - Look at profiler/optimization reports each time we wrote a line of code

> However, it is much easier to accelerate just matrix expressions (most other techniques not worth the additional dev time)

> Many packages available: MKL, Magma, CUBLAS/cuSparse

It's all just marketing!

Some case…

Should be pretty fast, right?

```
SUBROUTINE MATMULOPT(M, N, DATA, VEC, RES)
  INTEGER M, N, I, J
  DOUBLE PRECISION DATA(10000,10000)
  DOUBLE PRECISION VEC(10000), RES(10000)
' intent(out) :: RES

  DO J=1,N
      DO I=1,M
          RES(J) = DATA(I,J)*VEC(I) + RES(J)
      END DO
  END DO

END
```

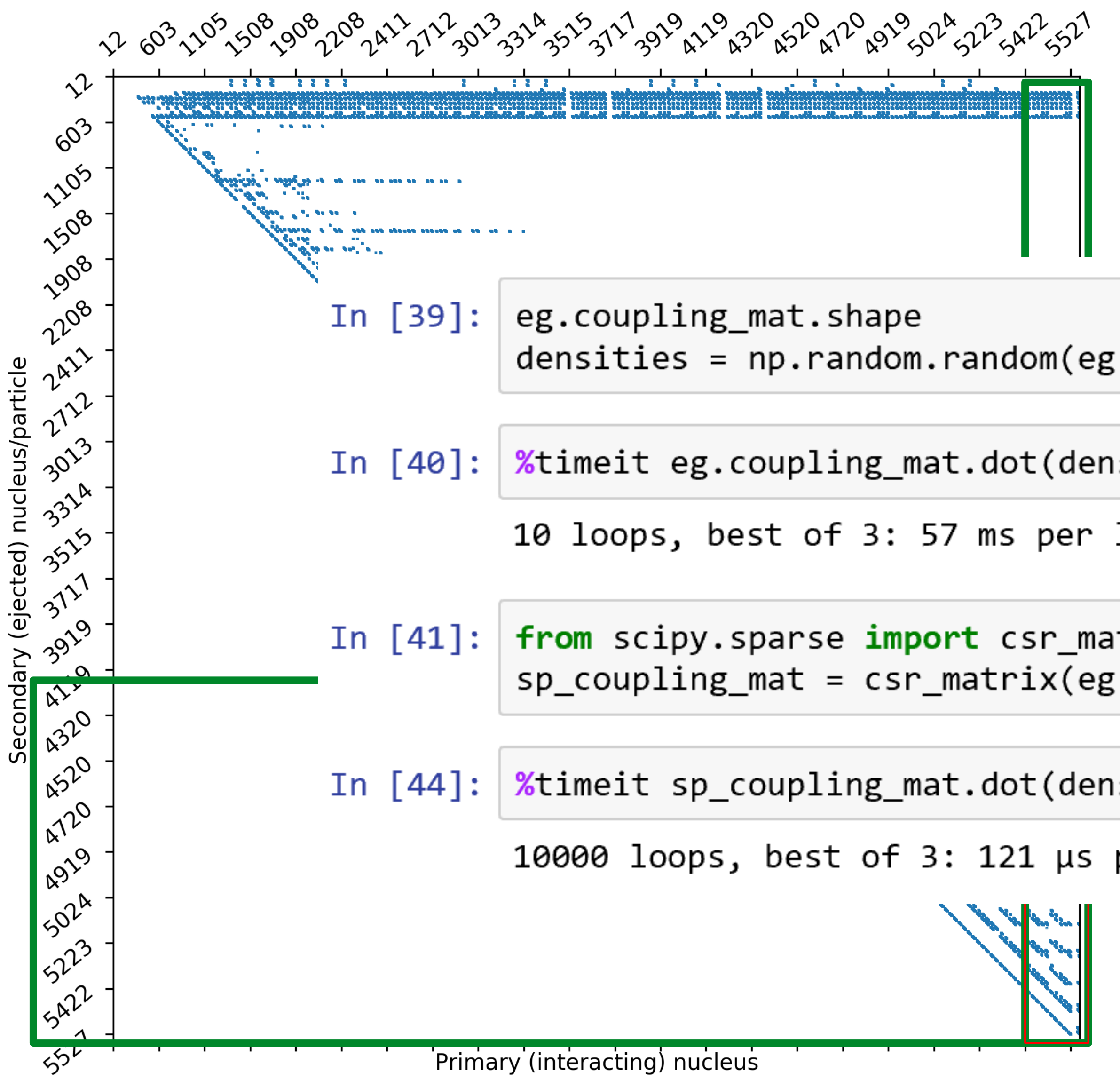
- This example is brute force
- Run on a tablet, workstation typically more
- Linear algebra has many interesting features (sparse matrices, efficient solvers, etc.)

Well,
… great …
but my "matrices" are neither random, nor dense!

```
In [3]: m,n, data, vec = 10000,10000, np.ra                          (10000)

In [4]: dataf = np.asfortranarray(data)

In [5]: vecf = np.asfortranarray(vec)

In [6]: %timeit fortrantest.matmulopt(m,n,dataf,vecf)
10 loops, best of 3: 130 ms per loop

In [7]: %timeit np.dot(data.T, vec)
10 loops, best of 3: 35.4 ms per loop
```

gfortran-7 -O3 vs. numpy linked to Intel MKL

More realistic case: propagation coupling matrix




```
In [39]: eg.coupling_mat.shape
         densities = np.random.random(eg.coupling_mat.shape[0])

In [40]: %timeit eg.coupling_mat.dot(densities)

10 loops, best of 3: 57 ms per loop
```

considered as dense

```
In [41]: from scipy.sparse import csr_matrix
         sp_coupling_mat = csr_matrix(eg.coupling_mat)

In [44]: %timeit sp_coupling_mat.dot(densities)

10000 loops, best of 3: 121 µs per loop
```

converted to sparse

- IDs: A*100 + Z
- represents an injection
- s are rows
- nts are columns
- channels of iron(ish) isotopes (1n, 2n, 1n1p emissions etc.)

Summary

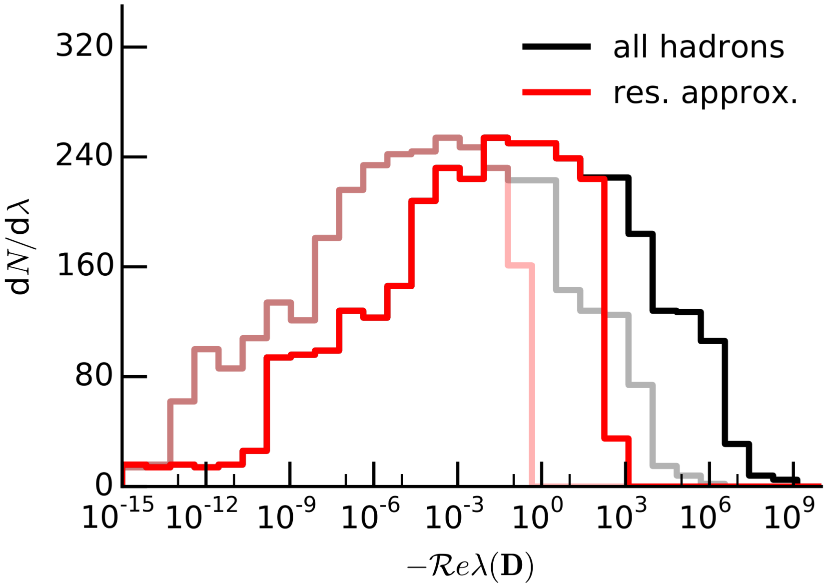
- Since we already write numerical code, we shall consider to directly think in addition and multiplication, and not in integral, derivative
- Radiation transport problems are in most cases **sparse problems**
- Calls to special functions (like pow(x,y)) are very expensive, interpolation is expensive,….
- Formulating the kernel of you problem in algebraic expressions gives you a lot of performance for free, vectorization doesn’t simply become marketing or impossible to afford due to dev time
- You can use GPUs, multi-core, etc., and if you need performance, you probably should, since CPU’s won’t accelerate much in the next decade
- By solving ultra-efficiently (in few seconds) the UHECR propagation problem, we will be able to do some fancy studies (part of the next workshop ;)

Semi-analytical approximations in matrix notations

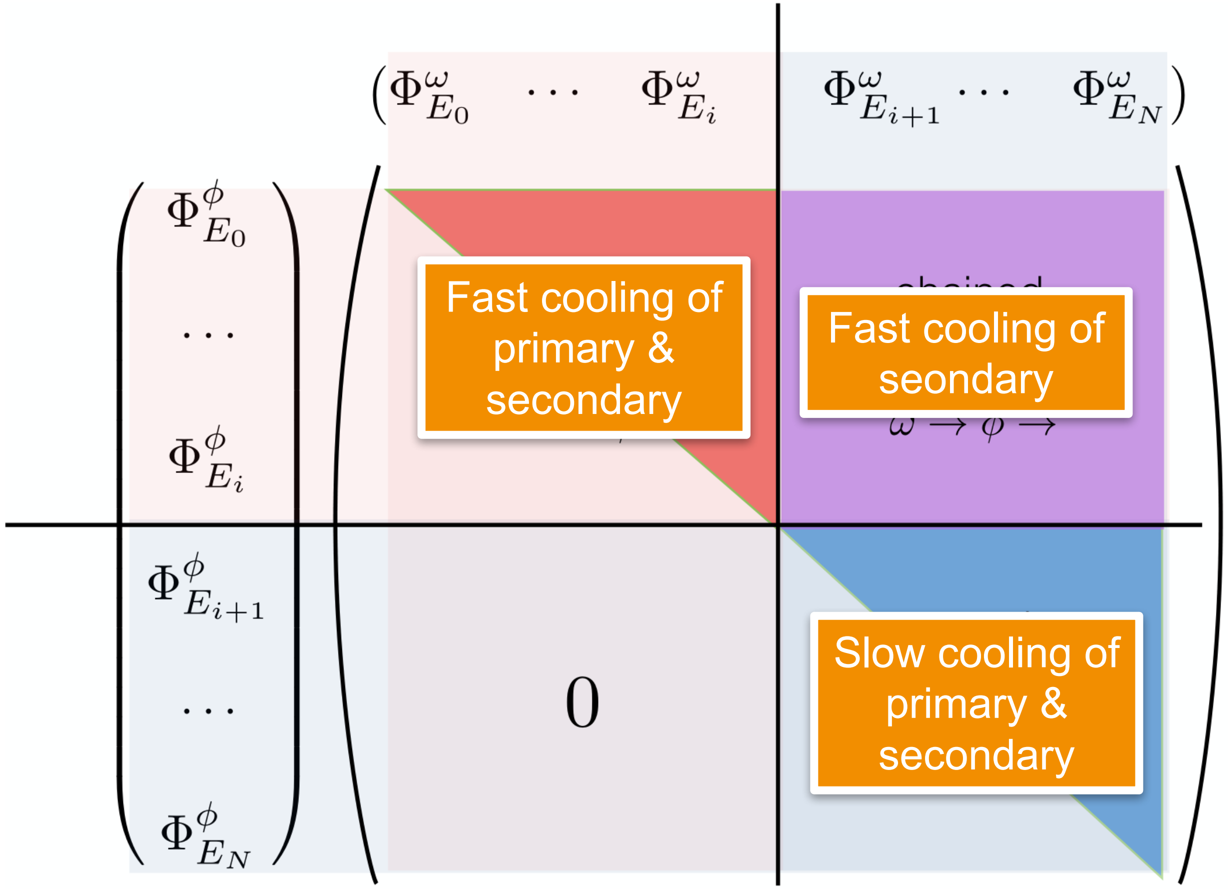
$$\vec{\Phi}^\omega = \left(\Phi^\omega_{E_0} \;\cdots\; \Phi^\omega_{E_i} \;\middle|\; \Phi^\omega_{E_{i+1}} \;\cdots\; \Phi^\omega_{E_N}\right)^T$$

$\lambda_{dec} < t_{mix}\lambda_{int}$	$\lambda_{dec} \geq t_{mix}\lambda_{int}$
$\Phi^\omega_{E_0} \cdots \Phi^\omega_{E_i} \equiv 0$	$\Phi^\omega_{E_{i+1}} \cdots \Phi^\omega_{E_N}$
treat as resonance	transport as particle

Result: removing fast processes from the system -> reduction of stiffness



$$\Delta\vec{\Phi}^{\text{chained}}_{n+1} = \left(\prod_k \boldsymbol{D}^{\text{res}}_k\right) \boldsymbol{C}\boldsymbol{\Lambda}_{int}\vec{\Phi}\Delta X_n$$



DESY