GPU CODE-GENERATION FOR DIFFERENTIAL EQUATION SOLVERS

Dániel Berényi

Wigner RCP, GPU Lab
2014 May 29.

INTRODUCTION

The most often encountered numerical problem in physics is

Integration

One of the main sources are

Differential Equations (DEs).

These equations state a relation between an unknown function and its derivatives.

The main goal is to reconstruct this unknown function

or at least some specific values of it

INTRODUCTION

Two main types of DEs to be discussed here:

- Ordinary Differential Equations:
  the unknown function depends on ONE variable.

  $$\frac{df(x)}{dx}$$

  Examples: classical point mechanics, reaction equations, population dynamics, …

- Partial Differential Equations:
  the unknown function depends on MULTIPLE variables.

  Examples: diffusion, waves, hydrodynamics, electrodynamics, …

  $$\frac{\partial f(x,y)}{\partial x}$$

INTRODUCTION

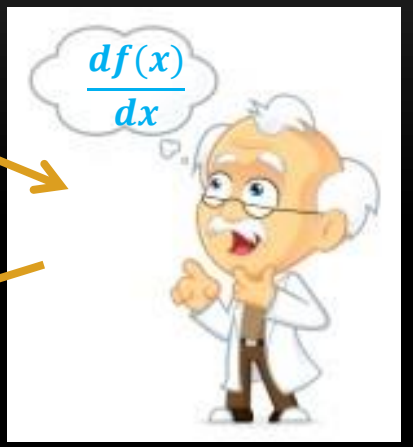
In many cases, especially for coupled equation systems the solutions cannot be determined analitically and numerics must be used…

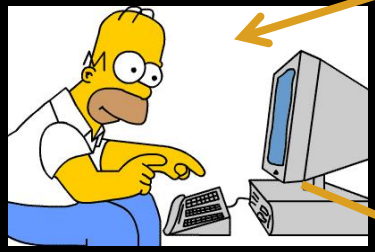
$$\begin{cases} \dfrac{\partial f(x,y)}{\partial x} + \dfrac{\partial f(x,y)}{\partial y} = \cdots \\ \dfrac{\partial g(x,y)}{\partial x} + \dfrac{\partial g(x,y)}{\partial y} = \cdots \\ \dfrac{\partial h(x,y)}{\partial x} + \dfrac{\partial h(x,y)}{\partial y} = \cdots \end{cases}$$

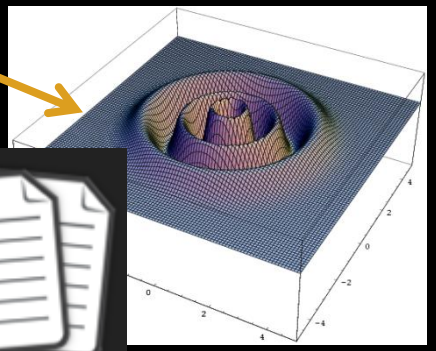
INTRODUCTION

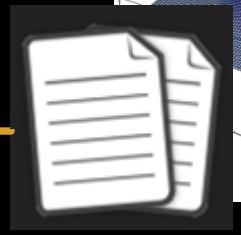
The (simplified) workflow of a physicist:

1. Select the phenomena of interest
2. Formulate a model in terms of equations
3. **Choose or Write a program to solve the equations**
4. Produce article with nice figures
5. Goto 1.





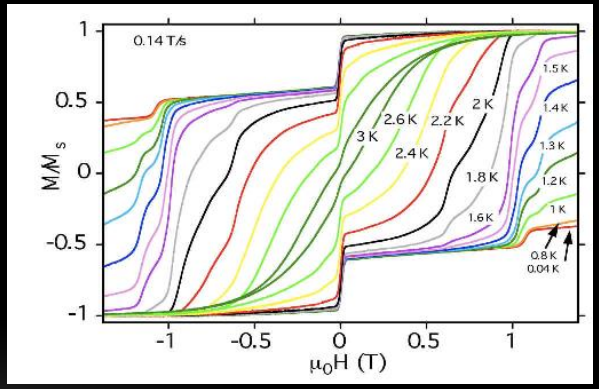




CHALLENGES

- Interesting / realistic models result in very complex equations.
- Some interesting phenomena are only visible after long time or in large systems or in high precision calculations
- Models have free parameters (often function parameters!) whose effect must be investigated over a wide range of possibilities
  → Parameter scans

Most physics problems has large computation cost!



WHY CODE-GENERATION?

- The toughest part today is not the unavailability of potent hardware but the development of fast and reliable codes for specific problems.
- The way from the original statement of the model until the publication of results can be full of mistakes/bugs/misprints etc.
- The software / API landscape is getting wider and hard to choose among them!

Why code-generation?

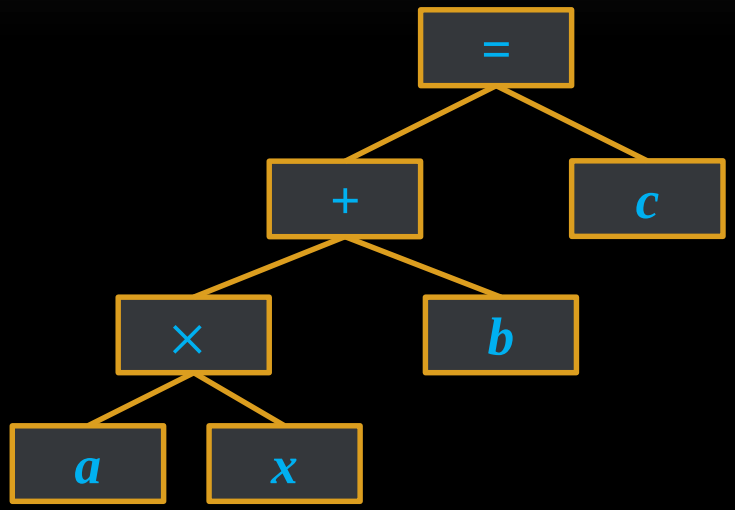
- Automatic specialization from generic to specific problems
- User input is at the first step, consistency is guaranteed during processing
- Can support multiple languages/frontends, easier to change or update

CODE-GENERATION

Proposed solution in terms of Abstract Syntax Trees (ASTs):

- Mathematical formulas and equations can be represented by trees:

$ax + b = c$

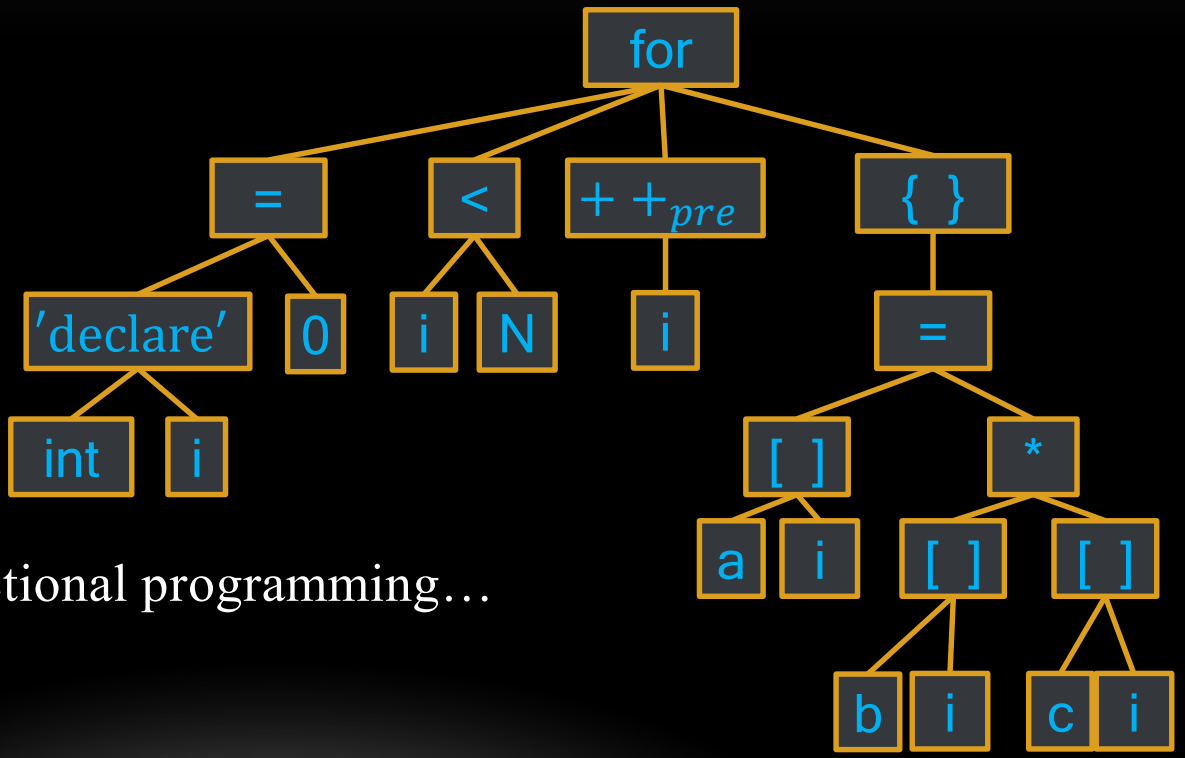


CODE-GENERATION

Proposed solution in terms of Abstract Syntax Trees (ASTs):

- Imperative programming constructs can also be represented by trees:

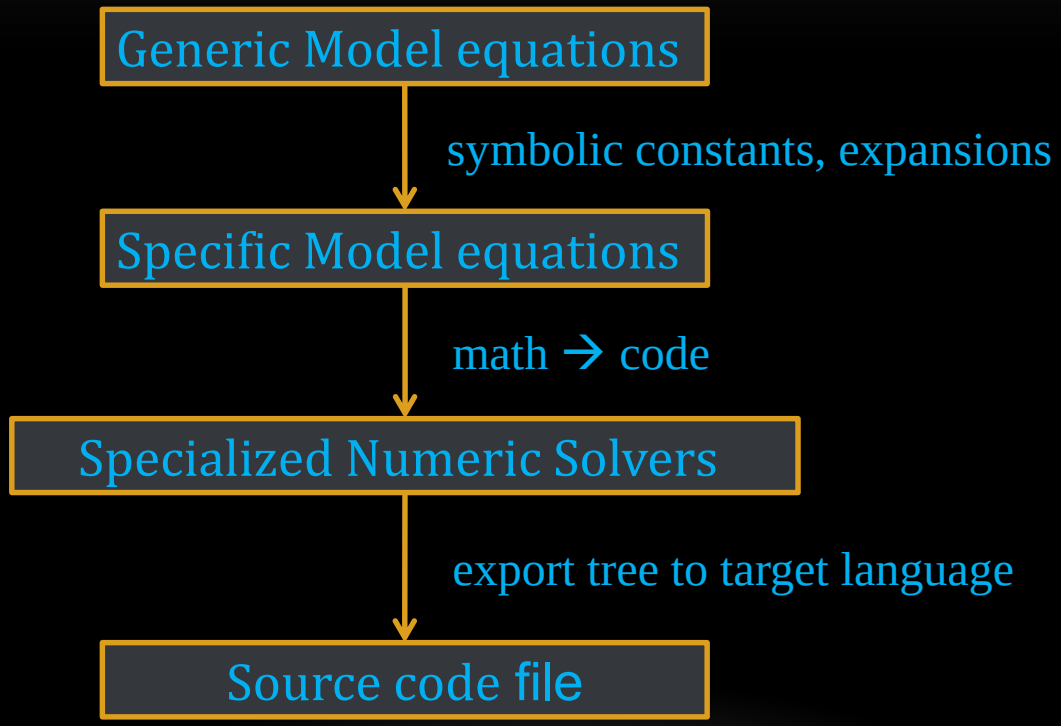
```
for(int i=0; i<N; ++i)
{
    a[i] = b[i] * c[i];
}
```



- Not to mention functional programming…

CODE-GENERATION

Since all of the needed constructs are trees the workflow can be seen as a series of transformations on the ASTs!



CODE-GENERATION

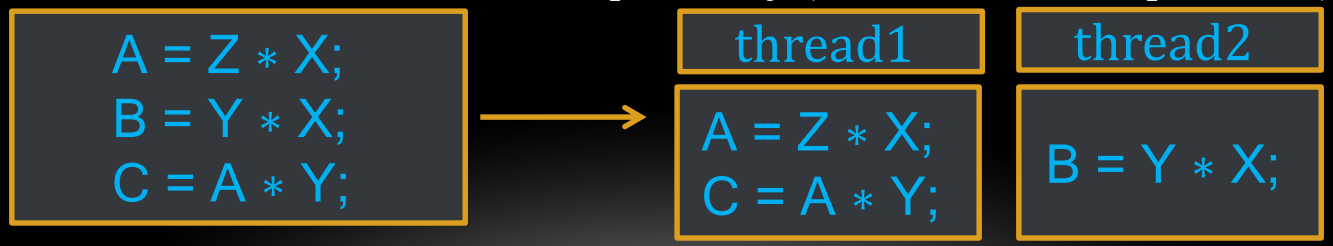
To provide error checking and collect information for optimalizations and parallelization the trees should be processed at the different stages:

- Symbolic (math) stage:
  simplifications ($0 \cdot a \rightarrow a$, $3a + 4a \rightarrow 7a$)
  symbolic differentiation ($\frac{d\sin(x)}{dx} \rightarrow \cos(x)$)

  series expansions ($f(x) \rightarrow \sum_{i=0}^{n} f_i \Phi_i(x)$
  check for argument sanity and rank mismatch
- Programming stage:
  Infer types, further sanity checks

  a = dot(vector<2, double>(2.5, 9.1) , vector<2, double>(1., 0.)) →a is scalar double

  Parallelization from data dependency (consider matrix operations):



SYMBOLIC STAGE

At the symbolic stage a general model is specialized according to user defined constants and parameters and simplified symbolically.
Numerical solvers are just higher-order functions operating on the equations.

Example: Spectral Expansion (like Fourier, Chebyshev series)

1. Equation:

$$\frac{df(x)}{dx} = -af(x)$$

2. Expansion: $f(x) = \sum_{i=0}^{N} f_i \Phi_i(x)$

$$\sum_{i=0}^{N} f_i \frac{d\Phi_i(x)}{dx} = -a \sum_{i=0}^{N} f_i \Phi_i(x)$$

3. Differentiation: $\Phi_i(x) = \cos(iNx)$

$$-\sum_{i=0}^{N} f_i iN \sin(iNx) = -a \sum_{i=0}^{N} f_i \cos(iNx)$$

SYMBOLIC→PROGRAMMING CONVERSION

All Math objects are given a type deduced from the leaves and propagated upwards.

Function definitions, signatures constructed, defunctionalization applied.

One important construct: ParallelFunction created from vector, matrix operations!

$$f1(a, \vec{x}, \vec{y}) := a\vec{x} + \vec{y}$$

$$\vec{z} = f1(a, \vec{x}, \vec{y})$$

```
void f1(range1 i, double a, double* z,
        double* x, double* y)
{
        z[i] = a * x[i] + y[i];
}
```

Calls are generated as:

```
ParallelCall( f1, RangeOf(z), a, z, x, y);
```

FINAL CODE-GENERATION

When all the conversions are ready the program tree is traversed and all the branch operators are converted to their textual equivalents in the selected languages.

Currently C++ / C / OpenCL export is considered.

Why C?
Largest common set of features supported by the compute and rendering APIs:

OpenCL kernel C, OpenGL GLSL,
DirectCompute, DirectX HLSL

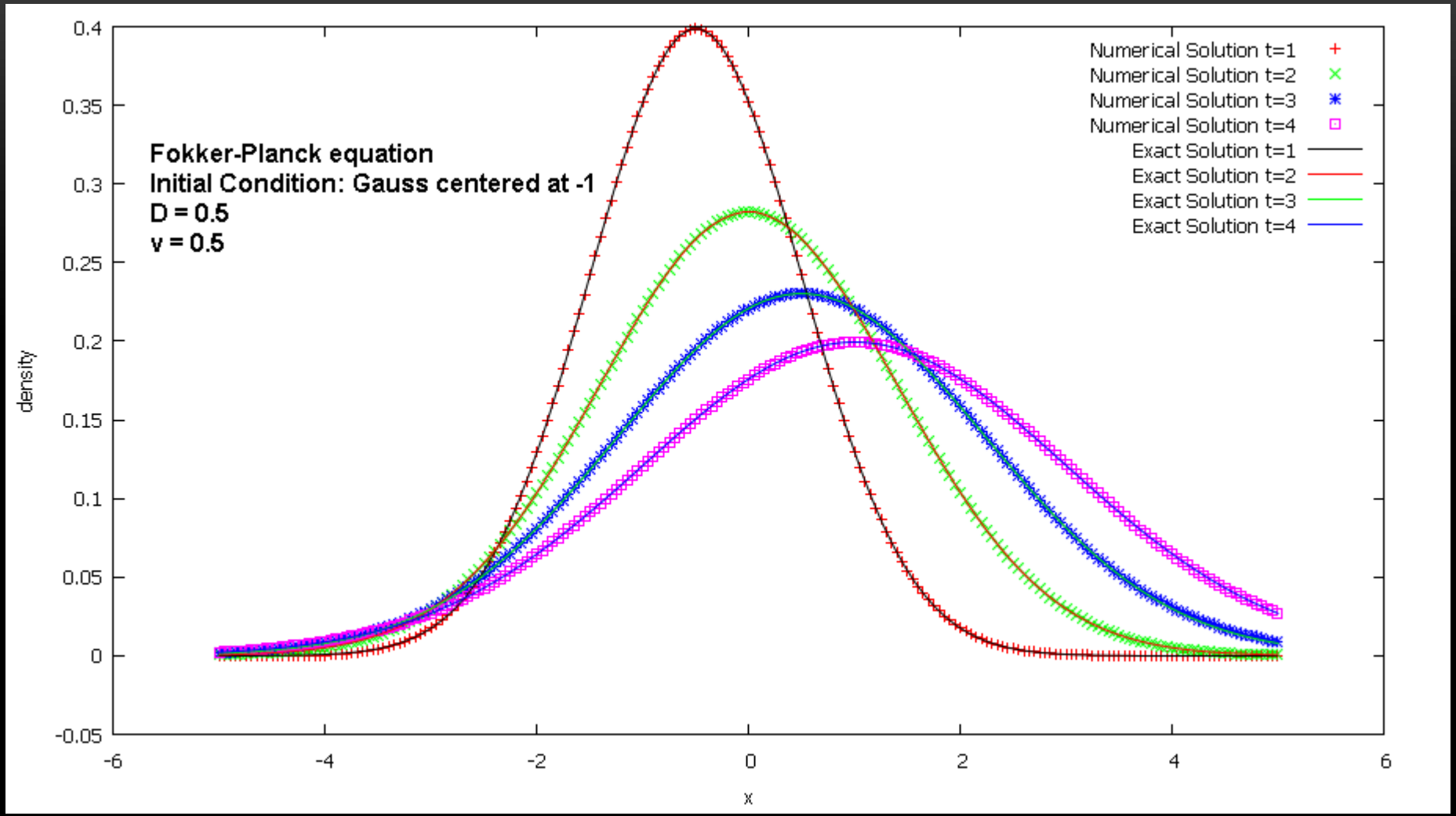
CURRENT IMPLEMENTATIONS

The outlined solutions are tested in two separate projects:

1.: A full spectral solver for 1D / 2D DEs was developed.

- Input of Equations, Variables and ranges in C++ code.
- Automatic symbolic simplification and spectral expansion
- Construction of the spectral coefficient matrix
- Inversion on the GPU (naïve non-generated LU decomposition)

Limited by the available memory on the GPU
(possible workaround: partition of the coefficient matrix)


Fokker-Planck equation
Initial Condition: Gauss centered at -1
D = 0.5
v = 0.5
Numerical Solution t=1
Numerical Solution t=2
Numerical Solution t=3
Numerical Solution t=4
Exact Solution t=1
Exact Solution t=2
Exact Solution t=3
Exact Solution t=4
density
x
0.4
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
-0.05
-6
-4
-2
0
2
4
6

```cpp
#include <Phys/DifferentialEquations.h>

void FokkerPlanckEquation()
{
    SymbolicDE DE;

    MathExpr t(L"t", 1, 1);
    MathExpr x(L"x", 1, 1);
    MathExpr f(L"f", 1, 1);
    MathExpr D(L"D", 1, 1);
    MathExpr v(L"v", 1, 1);
    MathExpr t0(L"t0", 1, 1);
    MathExpr pi(L"PI");

    DE.DimensionSymbols() << t << x;
    DE.UnknownSymbols() << f;
    DE.Equations() << diff(t, f) - diff(x, diff(x, D(x)*f)) + diff(x, v*f);
    DE.Constants() << equate(t0, 0.0);
    DE.Functions() << equate(D, 0.5) << equate(v, 0.5);

    DE.BoundaryConditions()
                  << f(t0, x) - exp(-sq(x+v*t0)/(D*4*t0))/sqrt(pi*4.0*t0*D);


    DE.SpectralBases()  << SpectralExpansion(L"RationalChebyshev", 48, 0.0, 1.0)
                        << SpectralExpansion(L"RationalChebyshev", 48, 0.0, 1.5);

    DE.ProcessAsFullSPectral();

    arr<double> ev; ev << 1.0 << 0.0;
    DE.SampleSolutionToFile1(L"out.txt", -5.0, 5.0, 0.05, 1, ev );
    arr<double> ev2; ev2 << 2.0 << 0.0;
    DE.SampleSolutionToFile1(L"out2.txt", -5.0, 5.0, 0.05, 1, ev2 );
    arr<double> ev3; ev3 << 3.0 << 0.0;
    DE.SampleSolutionToFile1(L"out3.txt", -5.0, 5.0, 0.05, 1, ev3 );
    arr<double> ev4; ev4 << 4.0 << 0.0;
    DE.SampleSolutionToFile1(L"out4.txt", -5.0, 5.0, 0.05, 1, ev4 );

    DE.SampleSolutionToFile2(L"fp.txt", -20.0, 20.0, 0.5, 0, -10.0, 10.0, 0.25, 1, ev4 );
}
```

DEMONSTRATION

CURRENT IMPLEMENTATIONS

The outlined solutions are tested in two separate projects:

2.: Host/Client OpenCL GPU code generator:

- An 8th order Runge-Kutta stepper was implemented
- Full pseudo-source code is written in C++ with wrapper classes
- Pseudo-code is exported and processed runtime (defunctionalization, parallel call construction)
- The host side C++ code and GPU side OpenCL Kernel is generated and compiled runtime into a DLL and loaded back to the program

Successfully tested with simple ODEs.

Asynchronous execution based on data dependency is currently being developed.

SUMMARY

Code generation from Abstract Syntax Trees is a nice versatile tool because:

- Can represent constructs from Mathematics and Programing and easily map to source code
- Symbolic manipulations can be performed
- Reusable, flexible language independent solver templates can be created
- Can generate all the low-level buffer manipulation between the CPU and GPU.

High-performance user friendly DE solvers are almost here.