CODE-GENERATION FOR DIFFERENTIAL EQUATION SOLVERS

Dániel Berényi

Wigner RCP, GPU Laboratory, Budapest, Hungary

Perspectives of GPU Computing in Physics and Astrophysics
Rome 2014.
INTRODUCTION

• The most often encountered numerical problem in physics is the solution of Differential Equations (DEs).

• Interesting problems require massive calculations on parallel hardware.

• Parallel code needs careful programming moreover much effort
MOTIVATION

- Pair production in strong fields:

A Boltzmann like equations for the quantum one particle distribution function: (QED: 3+3+1 dimension, 16 components, F. Hebenstreit et. al., Phys. Rev. D82 (2010) 105026.)

\[
\begin{align*}
D_t \psi &\quad - \quad 2 \vec{P} \cdot \vec{t}_1 = 0 \\
D_t \rho &\quad + \quad 2 \vec{P} \cdot \vec{t}_2 = 2m_0 \\
D_t \nu_0 &\quad + \quad \vec{D}_x \cdot \vec{\nu} = 0 \\
D_t \alpha_0 &\quad + \quad \vec{D}_x \cdot \vec{\alpha} = 2m_0 \\
D_t \nu &\quad + \quad \vec{D}_x \nu_0 + 2 \vec{P} \times \vec{\alpha} = -2m \vec{t}_1 \\
D_t \alpha &\quad + \quad \vec{D}_x \alpha_0 + 2 \vec{P} \times \vec{\nu} = 0 \\
D_t \vec{t}_1 &\quad + \quad \vec{D}_x \times \vec{t}_2 + 2 \vec{P} \cdot \vec{s} = 2m \nu \\
D_t \vec{t}_2 &\quad - \quad \vec{D}_x \times \vec{t}_1 - 2 \vec{P} \rho = 0
\end{align*}
\]

The differential operators are non trivial operator series: \( D_t = \partial_t + e \vec{E} \vec{\nu}_p + \ldots \)
MOTIVATION

• The solutions are nice smooth functions so we employ pseudo-spectral methods to solve, but one may choose among many possible spectral basis sets…

• The final equations are dependent on the electromagnetic vector field configuration addressed.

• One may not want to rewrite completely the equations each time, when the functional dependence or the basis is changed, especially when some components chosen to be or turn out to be zero…

• Can we generate the final code? Possibly from the equations? Even for GPU-s?
WORKFLOW

• The expected workflow is:

1. Formulation
   (mathematical equations, symbolic manipulation)

2. Spectral expansion into a dense matrix problem
   (maybe inside a finite difference time integrator)

3. Efficient parallel implementation/execution

• Possible tools:

1. Maxima, Mathematica, …

2. Maxima, Mathematica, … with or exported to Host high level language: C++, Fortran…

3. Parallel APIs & libraries: OpenCL, CUDA, … BLAS implementations…
PROBLEM WITH THE WORKFLOW

• It seems to be hard to make the above workflow userfriendly and automate the conversions between the representations.

• Most (academic) users doesn’t want to delve into GPU programming.

• In many cases development resources are more constrained than hardware resources!

• Can we use one tool and less development time?
PROPOSED SOLUTION

- We approached the problem from the viewpoint of embedded domain specific languages (EDSL):
  - Take a host language (e.g. C++) and create a restricted sub-language, that can express solely the operations you need.

One may choose different paths, of which two will be presented here:

- Static (compile time) approach via expression templates (See Máté Nagy-Egri’s talk)
- Dynamic/Staging (runtime) approach with direct Abstract Syntax Tree manipulations.
ABSTRACT SYNTAX TREES

- Mathematical formulas and equations can be represented by trees:

\[ ax + b = c \]
Imperative programming constructs can also be represented by trees:

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

Not to mention functional programming...
AST MANIPULATIONS

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

- **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

\[ a = \text{dot(vector<2, double>(2., 9.), vector<2, double>(1., 0.))} \rightarrow a \text{ is scalar double} \]

Parallelization from data dependency (consider matrix operations):

```plaintext
A = Z \times X;
B = Y \times X;
C = A \times Y;
```


D. BERÉNYI (WIGNER RCP)
AST MANIPULATIONS

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!

Calls are generated as:

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

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

OpenCL kernel C, OpenGL GLSL, DirectX HLSL
CODE-GENERATION FLOW

- Generic Model equations
  - symbolic manipulations: rewrite, simplify, partial eval, expansions
- Specific Model equations
  - math $\rightarrow$ code, specialization of solver templates
- Specialized Numeric Solvers
  - export tree to target language (possibly also an EDSL)
- Source code file

Generic Solver Templates
- written in an EDSL

D. BERÉNYI (WIGNER RCP)
CODE-GENERATION FLOW

1. Generic Model equations
   symbolic manipulations: rewrite, simplify, partial eval, expansions
   Specific Model equations
   math → code, specialization of solver templates
   Specialized Numeric Solvers
     export tree to target language (possibly also an EDSL)
   Source code file

2. Generic Solver Templates
   written in an EDSL

CURRENT IMPLEMENTATIONS

The above workflow has been splitted into two pilot projects:

1. A full spectral solver for 1D / 2D DEs:
   • 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)

Currently limited by the available memory on the GPU
(Further work: partition of the coefficient matrix)
#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(v, 0.5) << equate(t0, 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 );
}
Fokker-Planck equation
Initial Condition: Gauss centered at -1
$D = 0.5$
$\nu = 0.5$
CURRENT IMPLEMENTATIONS

The above workflow has been split into two pilot projects:

2. : Host/Client OpenCL GPU code generator:

- A C-like EDSL was created in C++ with wrapper classes
- An 8th order Runge-Kutta stepper was implemented in the EDSL
- The EDSL is manipulated: (defunctionalization, parallel call construction, host side C++ code and GPU side OpenCL Kernel generation)
- Then exported and compiled runtime into a DLL and loaded back to the program.
- A functor is supplied to the user to be called with data buffers and parameters.

Successfully tested with simple ODEs.
Asynchronous execution based on data dependency is currently being developed.
```cpp
#include "Computation.h"
#include "odes.h"
#include "Ex2.h"

struct RKState{ double x, v, t; double& operator[]( int i ) { return ((double*)&x)[i]; } }

void RK8Test()
{
    using namespace Metaprogramming;

    MetaProgram p;
    ID(a); //identifier for user input
    {
        ID(x); ID(v); ID(t); ID(s); ID(i); ID(ss); //identifiers
        Type Num("double"), State("State"), Int("int"); //type identifiers

        //State struct
        p |= decllist( Num|x, Num|v, Num|t ) | State;

        //RHS of DE
        p |= signature(State, State) | Id("rhs") = $(s, { !(State|ss), ss[~x] = s[~v], ss[~v] = -2.0*s[~x], rt(ss) });

        //Indexer function for State
        p |= signature(Num, State, Int) | Id("indexer") = $( ids(s,i), { rt( Select( i==0, s[~x], s[~v]) ) });

        //Higher order solver function definition imported:
        p |= getRK8(Id("indexer"));

        //Main entry point and solver invoke (translates to kernel call)
        p |= signature(Num::Void(), vec(Num ) |Id("main") = $(a, async_block( !Id("rk8")(domof(a), a, Id("rhs"), 2, Id("indexer")) ) ));
    }

    Namespace ns;
    au state = CreateBuffer<RKState>(200); //user buffer in CPU RAM
    for( int i=0; i<state.ext[0]; i++ ) { state[i].x = 0.0; state[i].v = 2.0*i; state[i].t = 0.0; }

    ns.CreateBuffer(a, state ); //Bind to identifier
    ns.AddCode(p); //Compile metaprogram
    ns.exec( a ); //Compile and Launch with identifier as parameter
    ns.ReadBuffer("a"); //Read back to user buffer
}
```
SUMMARY

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

- Can represent constructs from Mathematics and Programming 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.

This work was supported by the Hungarian OTKA Grants No. 77816, No. 104260, No. 106119.