Melodee: Solving ODEs with platform-specific code generation

DOE COE Performance Portability Meeting

August 24, 2017
Melodee is...

**Modular Expression Language for Ordinary Differential Equation Editing**

<table>
<thead>
<tr>
<th>For users</th>
<th>For developers</th>
</tr>
</thead>
<tbody>
<tr>
<td>A language for describing ordinary differential equations</td>
<td>A code-generation toolkit for ordinary differential equations</td>
</tr>
</tbody>
</table>
Overview

- Motivating problem—cardiac electrophysiology
- Design of the language
- Using Melodee to generate platform specific code
  - GPUs
  - CPUs
  - KNL
- Advantages of DSLs and code generation
Reaction term for cardiac electrophysiology

- Embarrassingly parallel ODEs
- Computation-bound
- Each cell requires
  - ~ 20-60 differential variable updates
  - ~ 60-100 libm evaluations
  - ~ 150-500 equation calculations
- Reaction takes 80% of the flops
It’s hard to port platform-specific code

- Cardioid was a Gordon Bell Finalist

- Exactly one reaction model optimized for BGQ
  - Math functions replaced with hard-coded rational polynomials
  - Thread load balancing based on BGQ architecture
  - Lots of BGQ vector intrinsics
  - 5800 LOC for 173 equations

- Our job: port this to GPUs
Our portability woes will only get worse

- The reaction model changes constantly
  - Reaction models are under constant development
  - Every experiment needs a novel reaction model

- We need platform-specific optimizations for performance

- ...but optimized code is
  - Un-maintainable
  - Platform dependent
  - Man-hour devouring
  - Tedious to write
Melodee: a language for ODEs

- Melodee is a domain specific language for describing ordinary differential equations
  ```
  subsystem lorenz {
    sigma = 10;
    beta = 8/3;
    rho = 28;
  }
  diffvar x,y,z;
  x.init = 1;
  y.init = 1;
  z.init = 1;
  x.diff = sigma*(y-x);
  y.diff = x*(rho-z)-y;
  z.diff = x*y-beta*z;
  }
  ```

- Not Turing complete

- [http://github.com/llnl/melodee](http://github.com/llnl/melodee)
Design goals for Melodee

- No existing language fit our needs, so we made our own

BioNetGen

Goals
- **Separate** math from implementation
- **Compatible** – easy to convert from C, Matlab, and cellML
- **Agnostic** – independent of domain & simulator
- **Extendible** – separate domain knowledge from semantics
- **Modular** – encourage re-use for large ODEs
- **Safe** – unit checking for common mistakes

Would this be useful for your domain?
Hodgkin Huxley in Melodee

```plaintext
integrate time {ms};

shared V {mV};
shared Ion {uA/uF};
shared E_R {mV};

subsystem hodgkin_huxley {
    subsystem leakage_current {
        E_L = E_R+10.613;
        @param g_L = 0.3;
        provides accum Ion += g_L*(V-E_L);
    }
    subsystem potassium_channel {
        diffvar @gate n {1};
        alpha_n = -0.01*(V+65)/expm1(-(V+65)/10);
        beta_n = 0.125*exp((V+75)/80);
        n.init = 0.325;
        n.diff = (alpha_n*(1-n)-beta_n*n);
        E_K = (E_R-12);
        @param g_K = 36;
        provides accum Ion += g_K*n^4*(V-E_K);
    }
    subsystem sodium_channel {
        diffvar @gate h {1};
        alpha_h {1/ms} = 0.07*exp(-(V+75)/20);
        beta_h {1/ms} = 1/(exp(-(V+45)/10)+1);
        h.init = 0.6;
        h.diff = (alpha_h*(1-h)-beta_h*h);
        diffvar @gate m {1};
        alpha_m = -0.1*(V+50)/(exp(-(V+50)/10)-1);
        beta_m = 4*exp(-(V+75)/18);
        m.init = 0.05;
        m.diff = (alpha_m*(1-m)-beta_m*m);
        E_Na = (E_R+115);
        @param g_Na = 120;
        provides accum Ion += g_Na*m^3*h*(V-E_Na);
    }
}
subsystem membrane {
    provides diffvar @interp(-100,100,2e-2) V;
    provides V_init {mV} = -75;
    Cm = 1;
    i_Stim = 0;
    if (time >= 10 && time <= 10.5) {
        i_Stim = 20;
    }
    V.init = V_init;
    V.diff = -Ion+i_Stim;
}
subsystem full_model {
    use hodgkin_huxley {
        export E_R as V_init;
    }
    use membrane;
}
```
Melodee is a code generation toolkit

- Developer writes code generators in python
  - Cardioid generator is only 600 LoC!

- Melodee parses .mel models for you
  - gives you a list of expressions in SSA

- Sympy for symbolic manipulation
  - Free symbolic differentiation.

- Flexible
  - Backwards compatible with cellML
  - Used in 3 different simulators
Reaction model optimizations

- **Rational polynomials** – replace expensive function evaluations with faster functions
- **Kernel fission vs fusion** – separate the ODE into multiple functions or one function
- **Replace exp/log** – variants based on floating point binary representation
- **Intrinsics** – use the compiler to vectorize or do it ourselves
- **SoA vs AoS** – How do we lay out our data structures?

<table>
<thead>
<tr>
<th>Optimization</th>
<th>BGQ</th>
<th>P100</th>
<th>KNL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rational polynomials</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Kernel fission vs fusion</td>
<td>fission</td>
<td>fusion</td>
<td>fusion</td>
</tr>
<tr>
<td>Replace exp/log</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Explicit vectorization with intrinsics</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>SoA vs AoS</td>
<td>SoA</td>
<td>SoA</td>
<td>AoS</td>
</tr>
</tbody>
</table>
Rational polynomials can replace expensive functions

double Afcaf = 0.3+0.6/(1.0+exp((v-10.0)/10.0));

becomes

double Afcaf;
{
    double numerCoeff[]={-9.52275328672 ... }; 
    double denomCoeff[]={2.18001528726e ... }; 
    double numerator=_numerCoeff[0];
    for (int jj=1; jj<8; jj++)
        _numerator = numerCoeff[jj] + v*numerator;
    double _denominator=denomCoeff[0];
    for (int jj=1; jj<6; jj++)
        _denominator = _denomCoeff[jj] + v*denominator;
    Afcaf = numerator/denominator;
}
GPU: Embedding the coefficients is much faster

```c
poly(double *in, int np, double *p, double *out)
{
    int ii = blockIdx.x*blockDim.x + threadIdx.x;
    double z = 0;
    for (int k=np-1; k>=0; k--)
        z = p[k] + z*in[ii];
    out[ii] = z;
}
```

**Memcpy:** 30.940us
**Naïve:** 202.15us
**Embedded:** 40.760us

```c
double *my_p[] = {...};
double z = 0;
for (int k=np-1; k>=0; k--)
    z = my_p[k] + z*in[ii];
out[ii] = z;
```

```assembly
/* 0x2b8 */
{ IADD32I R3, R3, -0x1;
  LDG.E.64 R10, [R6]; }
ISETP.GT.AND P0, PT, R3, RZ, PT
IADD32I R6.CC, R6, -0x8;
IADD32I.X R7, R7, -0x1;
DFMA R4, R8, R4, R10;
@P0 BRA 0x2b8;

DFMA R8,R2,R8,c[0x2][0x68];
DFMA R8,R2.reuse,R8,c[0x2][0x60];
DFMA R8,R2.reuse,R8,c[0x2][0x58];
...
Unrolling with a duff’s device

```
Unrolled

__constant__ double c_p[];
...
double z = 0;
switch (np) {
    case 8: c_p[7] + z*in[ii];
    case 7: c_p[6] + z*in[ii];
    case 6: c_p[5] + z*in[ii];
    case 5: c_p[4] + z*in[ii];
    case 4: c_p[3] + z*in[ii];
    case 3: c_p[2] + z*in[ii];
    case 2: c_p[1] + z*in[ii];
    case 1: c_p[0] + z*in[ii];
    default:
}
out[ii] = z;
```

- On CPUs, this is
  - just as fast as embedding
  - uses run-time coefficients

- On GPUs
  - c_p must be constant memory
  - c_p must be a constexpr
  - ptxas doesn’t emit indirect branches
    - Still have to pay for performance
      - Memcpy: 30us
      - Embedded: 40us
      - Unrolled: 46us
Polynomial code on CPUs

- Ran 1M points for 2k iterations for 15 degree polynomial
  - All times in seconds (lower is better)

- Vec means using explicit vector intrinsics

<table>
<thead>
<tr>
<th>Description</th>
<th>gcc</th>
<th>icc</th>
<th>clang</th>
</tr>
</thead>
<tbody>
<tr>
<td>embedded</td>
<td>6.81</td>
<td>16.56</td>
<td>6.94</td>
</tr>
<tr>
<td>naïve</td>
<td>21.94</td>
<td>23.95</td>
<td>22.27</td>
</tr>
<tr>
<td>unrolled</td>
<td>14.46</td>
<td>14.67</td>
<td>14.239</td>
</tr>
<tr>
<td>embedded+vec</td>
<td>6.79</td>
<td>6.86</td>
<td>6.87</td>
</tr>
<tr>
<td>naïve+vec</td>
<td>7.35</td>
<td>7.91</td>
<td>10.12</td>
</tr>
<tr>
<td>unrolled+vec</td>
<td>6.83</td>
<td>6.82</td>
<td>6.86</td>
</tr>
</tbody>
</table>

Explicit vectorization must be used for performance on CPUs
Rational polynomial summary

- BGQ, Haswell: need manual vectorization
- GPU: coefficients must be known at compile time
- KNL: rational polynomials are slower?
  - I see a 10% slowdown currently
  - KNL has vector intrinsics for exp, etc.
  - Rational polynomials cause L1 spills??
Intrinsics

- **BGQ, Haswell, KNL**
  - Compilers will NOT auto-vectorize this code
  - Must generate vector intrinsics specific to platform

- **GPU**
  - No intrinsics necessary
**Kernel fission or fusion?**

- **BGQ:** required separating the reaction model into decoupled functions
  - Each thread integrated different variables independently

- **KNL, GPUs:** Faster results by putting everything in one monolithic kernel
  - Hide memory latency with computation.
Replacing exp/log

- Use fact that IEEE754 contains a base-2 logarithm in exponent

- BGQ: essential for good performance
- GPU: replacing exp/log is no faster.
- KNL: 50% slowdown when replacing exp/log
  - Faster intrinsics on chip
Data layout

Structure of Arrays
struct state {
    double x[n];
    double y[n];
}

Array of Structures of Vectors
struct stateVec {
    double x[vwidth];
    double y[vwidth];
}
stateVec state[n/vwidth];

- Haswell, KNL: AoSoV is faster
- BGQ, GPU: SoA is faster
Summary

- Domain specific languages make us more productive
- Code generation is essential for portable performance

<table>
<thead>
<tr>
<th>Optimization</th>
<th>BGQ</th>
<th>P100</th>
<th>KNL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rational polynomials</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Kernel fission vs fusion</td>
<td>fission</td>
<td>fusion</td>
<td>fusion</td>
</tr>
<tr>
<td>Replace exp/log</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Explicit vectorization with intrinsics</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>SoA vs AoS</td>
<td>SoA</td>
<td>SoA</td>
<td>AoS</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Performance (% of peak)</th>
<th>BGQ</th>
<th>P100</th>
<th>KNL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>60%</td>
<td>38%</td>
<td>11%</td>
</tr>
</tbody>
</table>

[http://github.com/llnl/melodee](http://github.com/llnl/melodee)
Acknowledgements

- LLNL
  - Dave Richards
  - Tom O’Hara
  - Xiaohua Zhang

- IBM
  - Doru Bercea

- Intel
  - Doug Jacobsen
Use case: Developing a new reaction model

integrate time \{ms\};
shared V \{mV\};
shared I_{ion} \{uA/uF\};
shared V_{init} \{mV\};

subsystem ORd\_with\_newIKr {
    shared ko \{mM\};
    shared E_K \{mV\};
    shared I_{Kr} \{uA/uF\};
    use ORd\_sodium\_current\_rapid\_rectifier\_current {
        export E_{Na};
    }
    use TT06\_fast\_sodium\_current {
        export i_{Na} as I_{Na};
        export E_{Na} as E_{Na};
    }
    use newIKr;
}