The NAG PDE Toolkit

A code generation framework for PDE solvers

The NAG PDE Toolkit Development Team

Overview of the toolkit

Code generation allows you to write a PDE solver and automatically generate high performance implementations on CPU and GPU, including MPI. It is a necessary tool to achieve performance portability.

NAG has created a Domain Specific Language (DSL) embedded in C++. The DSL allows users to specify **stencil based solvers on structured grids**. A Clang tool parses the code and produces implementations targeting OpenMP+SIMD or CUDA.

Toolkit functionality

- Supports PDEs in up to 6 dimensions (and can go higher)
- Matrix-free iterative solvers (GMRES, CG, BiCGStab)
- Preconditioners, direct solvers and automatic matrix assembly
- Possible to implement finite element solvers on structured grids
- Operator splitting for ADI time steppers
- Highly optimised batch tridiagonal solvers
- Ability to use existing C/C++ code (with restrictions)
- Built-in algorithmic differentiation
- Batch solution of similar PDEs, e.g. risk calculations

A DSL embedded in C++

The toolkit is exposed as a standards compliant C++11 class library, with some additional "language rules." Examples of additional rules:

- stencil accesses are only allowed inside `Operators`
- stencil offsets must be integer constant expressions (ICEs)
- only scalars, `Dats` or aggregates of these may be accessed inside `Operators`

The Clang tool enforces the DSL rules and handles code generation. Below are snippets from an example solving an SLV-type model with Hundsdorfer-Verwer time stepping. The full code is available on request.

```cpp
// Example of an existing user function
template<class FP>
FP myBoundaryValFunc(const FP x, const FP v, const FP K);

template<class FP>
void foo( ... /* user args */ ...)
{
  using namespace pdetk;
  // Cartesian iteration range in X,V inclusive of end points
  IterationRange range({0,M1, 0,M2});
  IterationRange interior({1,M1-1, 1,M2-1});
  // The solution surface. Dats hold data at grid points
  Dat<FP> Ui(range);
  // Operator to set Dirichlet boundary conditions
  Operator Boundary([&]() {
    // Get (i,j) index for 2D operator
    Index idx(2);
    FP x = xmin + idx[0]*dx;
    FP v = vmin + idx[1]*dv;
    if(idx[0]==0 || idx[0]==M1 || idx[1]==0 || idx[1]==M2) {
       Ui(idx) = myBoundaryValFunc(x,v,K);
    }
    });
  // Run operator on full range to set values in Ui
  Boundary(range);

  // Operator for SLV model in interior of uniform grid
  Operator F([&](const TimeIndex &ti, const Dat<FP>& Ui) -> Dat<FP> {
    Index idx(2);
    FP Ux, Uxx, Uv, Uvv, Uxv;
    // Stencil ops for dimension splitting in X
    Ux = ( Ui.dim1(idx[0]+1,idx[1]) - Ui.dim1(idx[0]-1,idx[0]) )/(2*dx);
    Uxx = /* second order central difference */
    // Stencil ops for dimension splitting in V
    Uv = ( Ui.dim2(idx[0],idx[1]+1) - Ui.dim2(idx[0],idx[0]-1) )/(2*dv);
    Uvv = /* second order central difference */
    // Stencil ops for dimension splitting in XV
    Uxv = ( Ui.xdim(idx[0]+1,idx[1]+1) - Ui.xdim(idx[0]-1,idx[1]+1) ...
    // Return value
    Dat<FP> U(2);
    // Leverage surface Sigma(t,x)
     FP sigma = Sigma[ti](idx[0]);
     FP vj = vmin + idx[1]*dv;
     U(idx) = (rate-0.5*sigma*sigma*vj)*Ux + 0.5*sigma*sigma*vj*Uxx
            + kappa*(eta-vj)*Uv + 0.5*volvol*volvol*vj*Uvv
            + sigma*volvol*rho*vj*Uxv;
    return U;
    });

  // Time stepping loop
  for (TimeIndex ti(0) ; ti < N-1; ti++)
  {
    // Do Hundsdorfer-Verwer time stepping
    Dat<FP> RHS(2), Y1(2), Y2(2);
    Dat<FP> Y0 = Ui + dt*F(interior,ti,Ui);
    // Get matrix for operator split in X
    Matrix<FP> Mx = jacobian<FP>(Y1 - theta*dt*F.dim1(interior,ti+1,Y1)
                                                        ,D(), byD(Y1));
    RHS = Y0 - dt*theta*F.dim1(interior,ti,Ui);
    // Solve matrix system Mx*Y1 = RHS
    Mx.solve(RHS, Y1);
    ... /* etc */ ...

    swap(Ui,Uii);
   }
   // Get the data out of Dat into a std::vector
   Ui.getData(soln);
}
```

Project status and early results

The NAG PDE Toolkit is in **early development** with release slated for Q1'20. Results from current development builds solving the SLV problem outlined are shown below. The numbers include a certain amount of hand-tuning by NAG. The goal is to build NAG's expertise into the PDE Toolkit code generation.

Grid size ($x \times v$) is across the top, batch size is down the side. The solver performs 100 ADI time steps with all calculations in double precision. The table reports the number of PDEs per second (PVs/s).

	100x50	200x100	200x200	300x150	300x300	400x400
	Intel Xeon W-2145 (Skylake), 8 cores					
64	350	68	31	27	14	7
96	380	68	31	27	14	7
192	420	68	31	27	14	7
320	450	68	31	27	14	7
576	450	68	31	27	14	7
992	450	68	31	27	14	7
1536	400	60	29	25	13	6
2048	400	60	29	25	13	6
	NVIDIA V100, 16GB RAM					
64	70	60	50	50	39	28
96	110	85	66	65	46	32
192	205	140	100	90	58	37
320	330	180	120	110	66	40
576	550	240	140	126	71	OOM
992	850	280	153	137	OOM	OOM
1536	1150	320	161	OOM	OOM	OOM
2048	1380	340	OOM	OOM	OOM	OOM

"OOM" means the V100 ran out of memory. CPU code is vectorised and performance is stable. To fully load the GPU with small problems requires huge batch sizes. If problem size increases even modestly, GPU performance picks up strongly and continues to scale as the card is given more work.

Early engagement

If the functionality and performance outlined here interests you, then NAG would like to discuss your requirements in more detail to ensure the PDE Toolkit integrates smoothly into your environment.

For further information please contact **support@nag.co.uk**.