High speed flow simulation: a look forward to exascale

Current state of the art and software future-proofing

Neil Sandham, Satya Jammy, Christian Jacobs, Markus Zauner, David Lusher
University of Southampton

Funded by EPSRC (grant EP/K038567/1, UK Turbulence Consortium EP/L000261/1)
and H2020 "ExaFLOW: Enabling Exascale Fluid Dynamics Simulations"

UK Turbulence Consortium

- Since 1995, now 46 academics at 21 UK institutions
  - Allocations on national HPC facilities
  - Support of porting, benchmarking and optimisation
- Proliferation of codes
- Since 2018 focus limited resources on a small number of open source codes
  - InCompact3D, CodeSaturne, Nektar++ and OpenSBLI

Outline

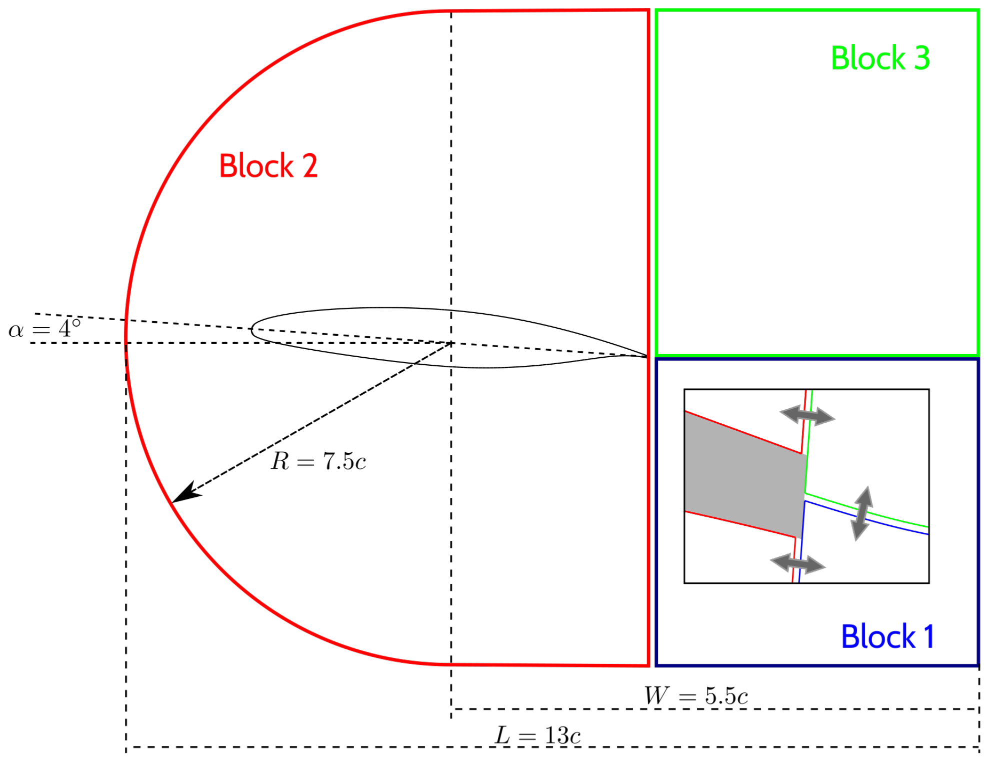
- Legacy SBLI code
  - Methods & sample application
- Future-proofing simulation codes
  - OPS approach (source-to-source translation)
  - Automatic code generation: OpenSBLI
- Performance and outlook towards Exascale
  - Store vs recompute on various hardware platforms
  - Energy consumption

Brief overview of numerical approach

Compressible Navier-Stokes, Newtonian fluid, multi-block curvilinear grids

- Fourth order accurate (central) space differencing,
- Explicit in time RK3 or RK4
- Equation conditioning (entropy splitting, Laplacian formulation of viscous term)
- Avoid filtering for direct numerical simulations (DNS)
  - local oscillations in DNS if flow under-resolved
- Mixed time scale sub-grid model for large eddy simulations (LES)
- Shock capturing (if selected) applied as full-step filter TVD +ACM+Ducros
- Legacy SBLI code (Fortran 95)

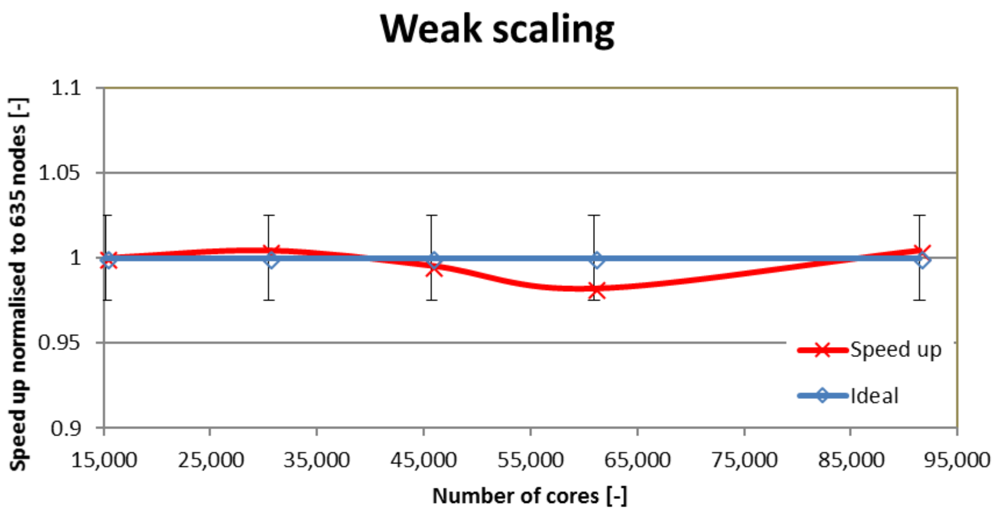
Example: High-fidelity studies on transonic buffet:
Markus Zauner PhD 2019

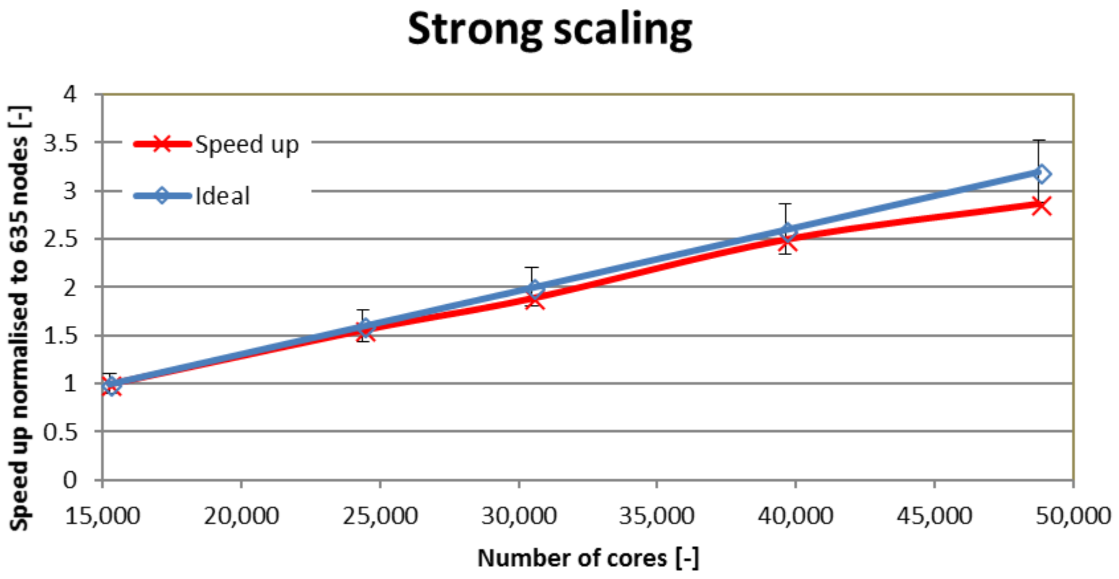


- Mach number: *M=0.7*
- Reynolds number: *Re=500,000*
- Prandtl number: *Pr=0.72*
- Sutherland law: *C_{suth}=0.41*

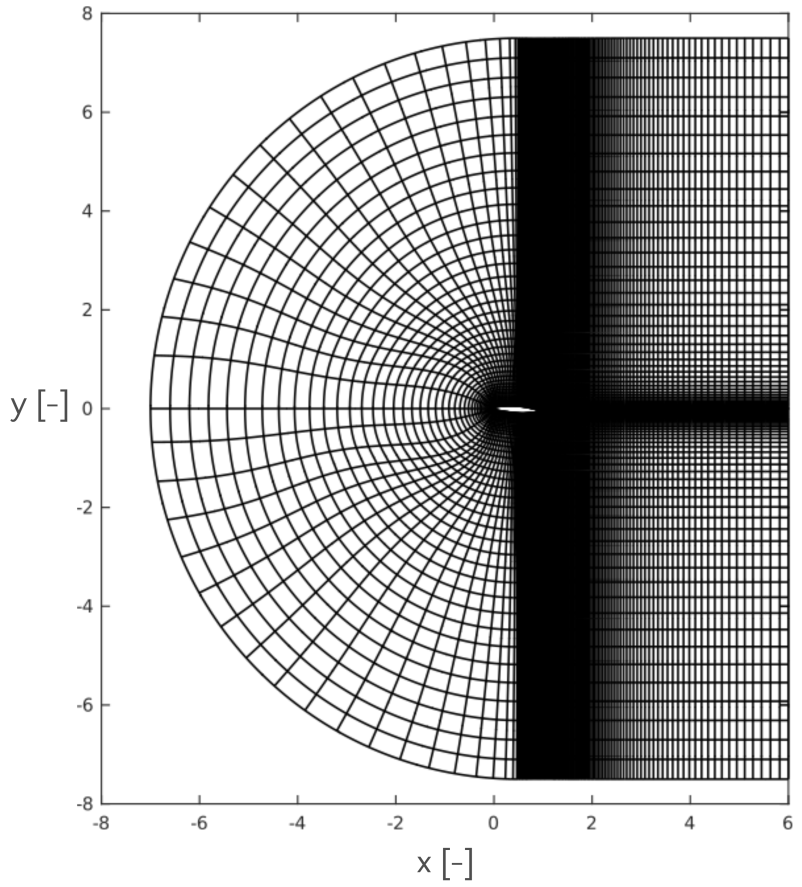
V2C wing profile

Basic code: Scaling on HazelHen (PRACE)





V2C Grid



- Generation: PolyGridWizZ (in-house code)

  ➠ *https://github.com/ZaunerM/PolyGridWizZ*

  Multiblock structured grids for direct numerical simulations of transonic wing sections
  M. Zauner & N. Sandham
  Proceedings of ICCFD10, Barcelona, 2018

  open source

- Gridpoints: Block 2: 3045 x 999 x 150
  Block 1/3: 1999 x 1023 x 150
  Total: $1.07 \cdot 10^9$ points

- Spanwise domain: $L_z = 5\%_c$

- Cell size: Time step = $2 \cdot 10^{-5}$ time units
  Trailing edge: ζ $2.5 \cdot 10^{-4}$ c
  ξ $3.3 \cdot 10^{-4}$ c
  η $1.0 \cdot 10^{-4}$ c

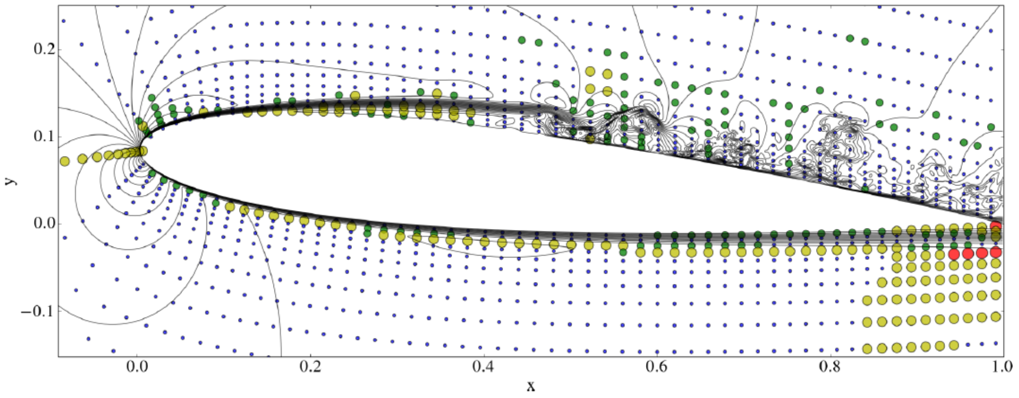
- Costs: $132{,}079$ CPUh per time unit

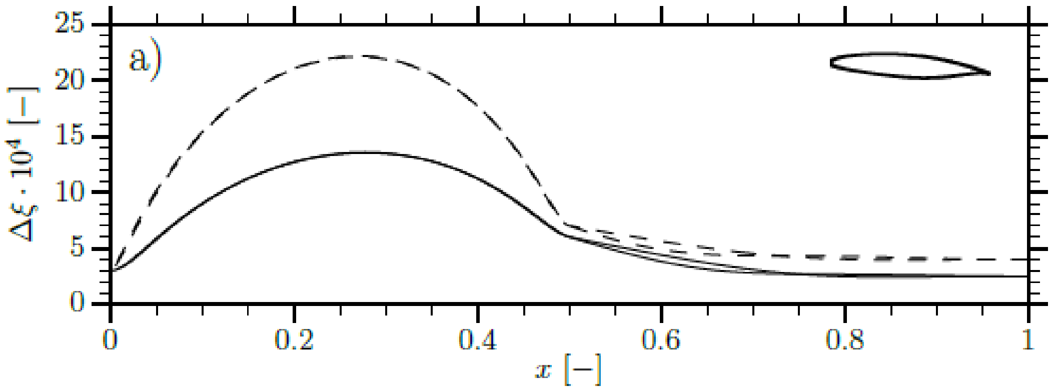
30,000 processors -> ~4 hours for 1 time unit

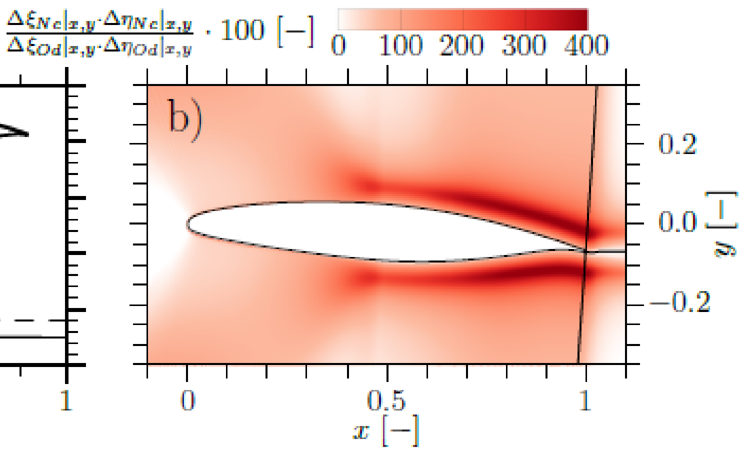
Error indicators

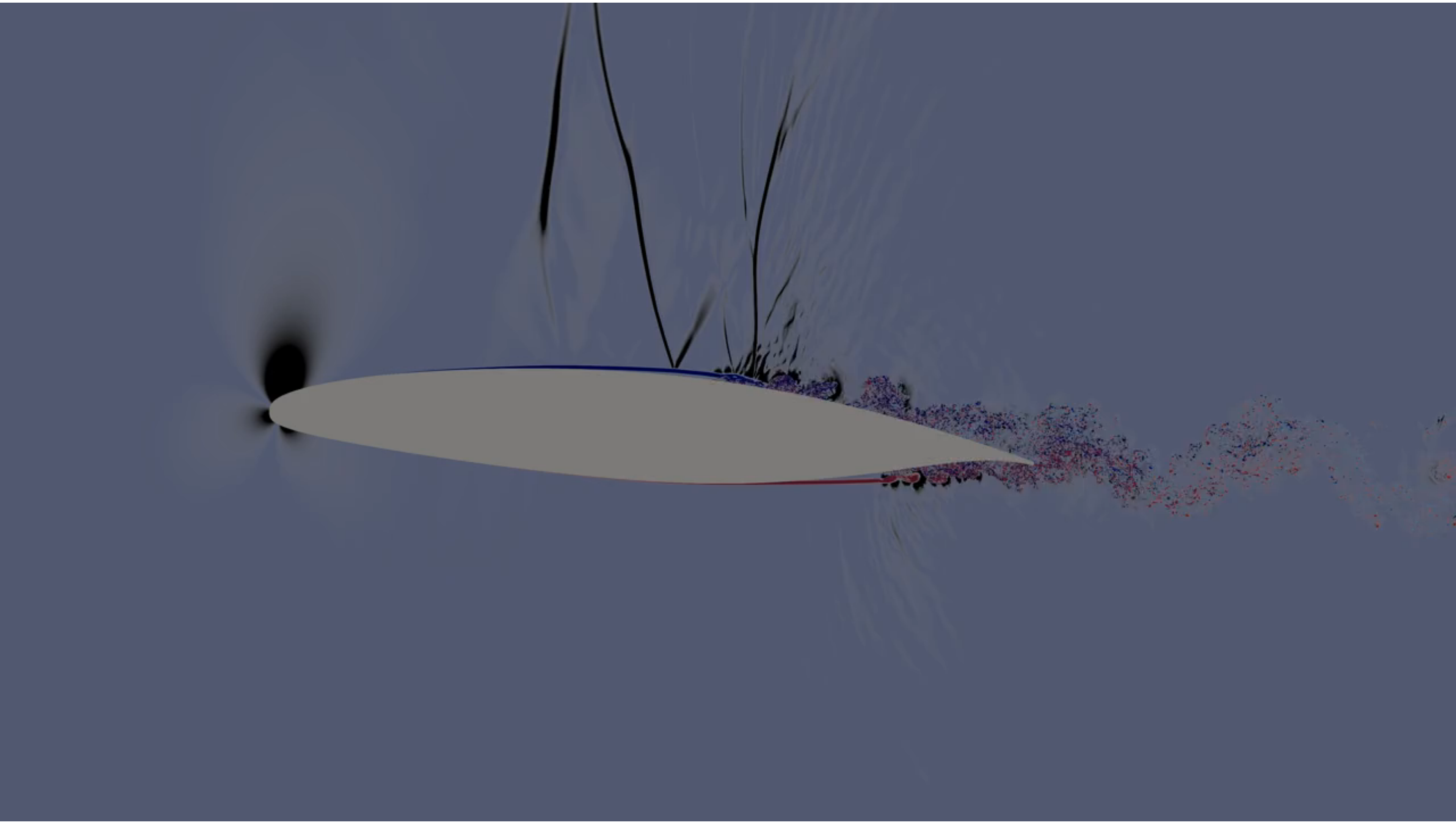
Workflow:

- Parametrised structured grid generation
- Spectral error indicator
- Iterative 2D grid adaptation
- Iterative 3D grid adaptation

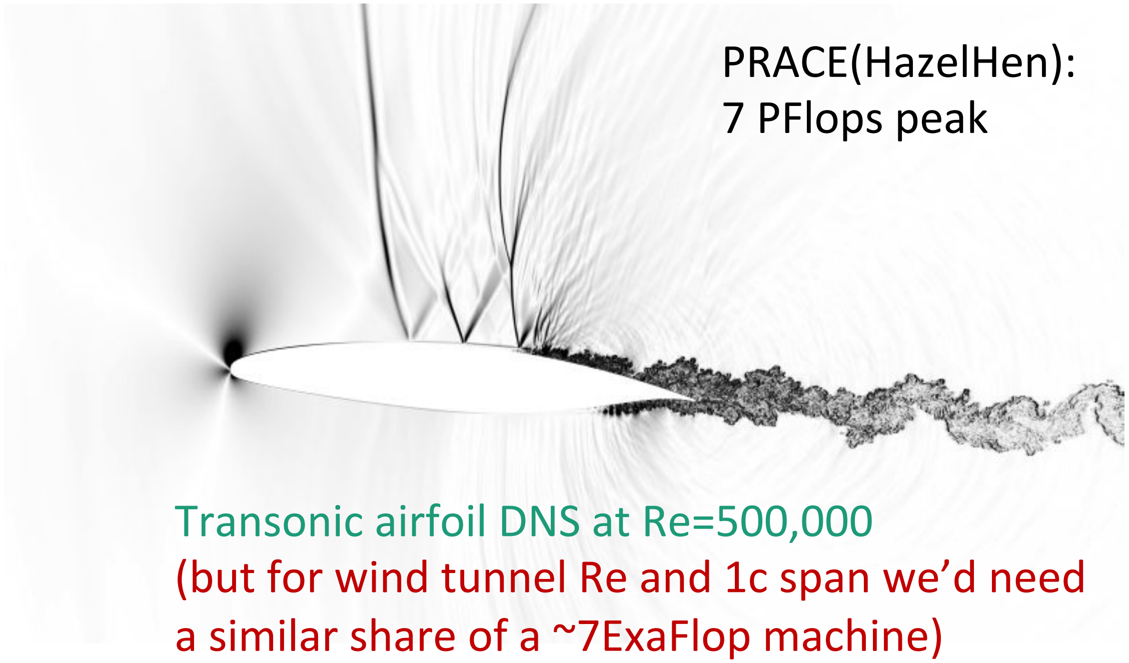








SWBLI: towards exascale



PRACE(HazelHen):
7 PFlops peak

Transonic airfoil DNS at Re=500,000
(but for wind tunnel Re and 1c span we'd need a similar share of a ~7ExaFlop machine)

- Projections and issues:
  - CPU, GPU and potential mixed/novel architectures
  - energy efficiency
  - fault tolerance
  - data compression
  - in-situ graphics
- Porting may require a **non-trivial code rewrite**, requiring expertise in fluid dynamics, numerical methods, and parallel computing paradigms, and their efficient implementation
- …and newer architecture might arrive during porting

Investigation of future-proofing with OPS
(EPSRC project 2014-2016)

- **OPS**: **O**xford **P**arallel library for **S**tructured-mesh computations
  - Key people: Gihan Mudalige, Istvan Reguly, Mike Giles
  - Multi-block structured applications
  - Source-to-source translation for parallel implementations on various architectures
  - Very little overhead with the automation process for hydrodynamic applications e.g. CloverLeaf

Example for simple stencil averaging

ops_par_loop:

```
        int range[4] = {imin,imax,jmin,jmax};
        ops_par_loop(calc, block, 2, range,
        ops_arg_dat(a,S2D_0,"double",OPS_WRITE),ops_arg_dat(b,S2D_1,"double",OPS_READ));
```

Kernel:

```
        void calc(double *a, const double *b) {
         a[OPS_ACC0(0,0)] = 0.5*(b[OPS_ACC1(1,0)] + b[OPS_ACC1(-1,0)];)
        }
```

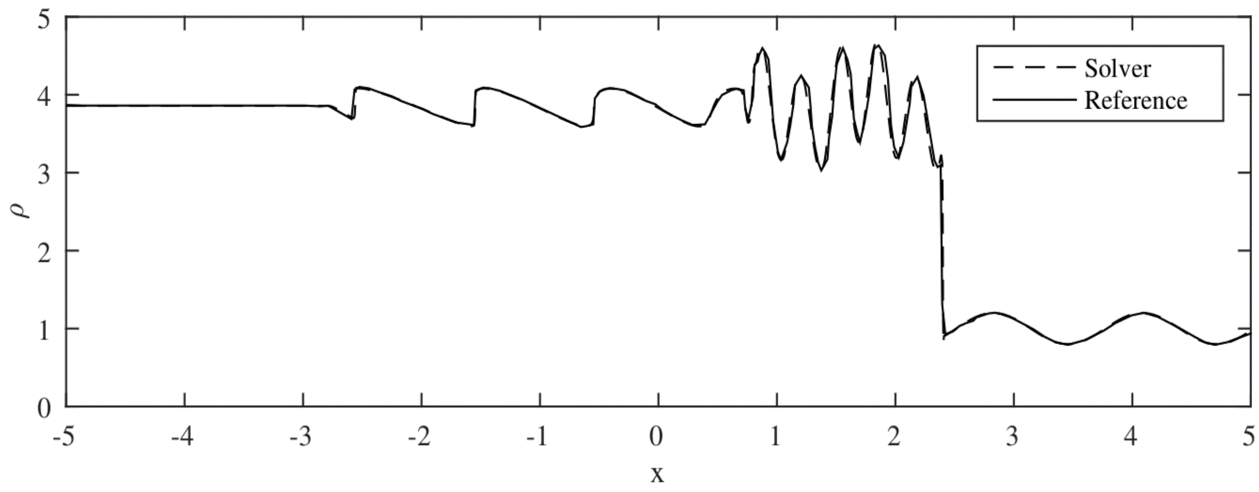
* Substantial coding required, even for simple operations

Proof of concept: Shu-Osher case

Left state($x<=-4$)	Right state($x>-4$)
Density = 3.857143	1+0.2**sin*(x)
Velocity = 2.629369	0
Pressure = 10.3333	1

Validation grid N=2500

Density profile compared with WENO (Pirozzoli) at t=1.8



- Jammy et al

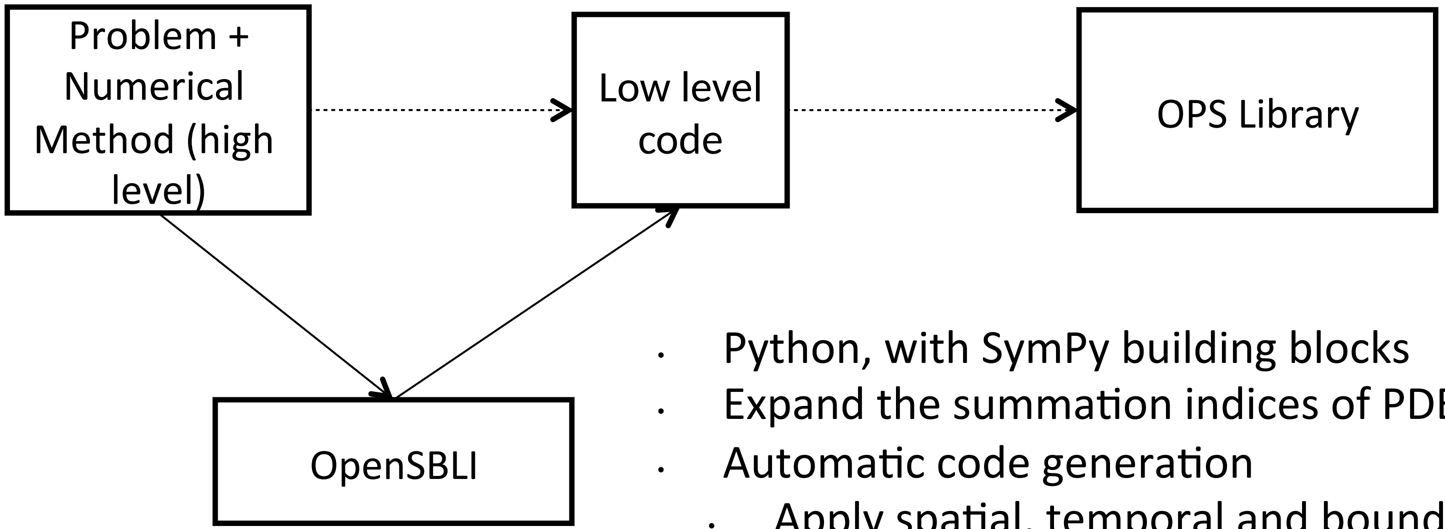
(ParCFD 2015)

Speed ups of up to 6.57x on **GPU** (NVIDIA Tesla K20c 2946 CUDA cores 5GB memory) vs **CPU** (Intel® Xeon®E5-2640 @2.5GHz 12 cores MPI). Also tested OpenCL and OpenMP with no change to code

OpenSBLI: ongoing experiment in automatic code generation

Separation of concerns.

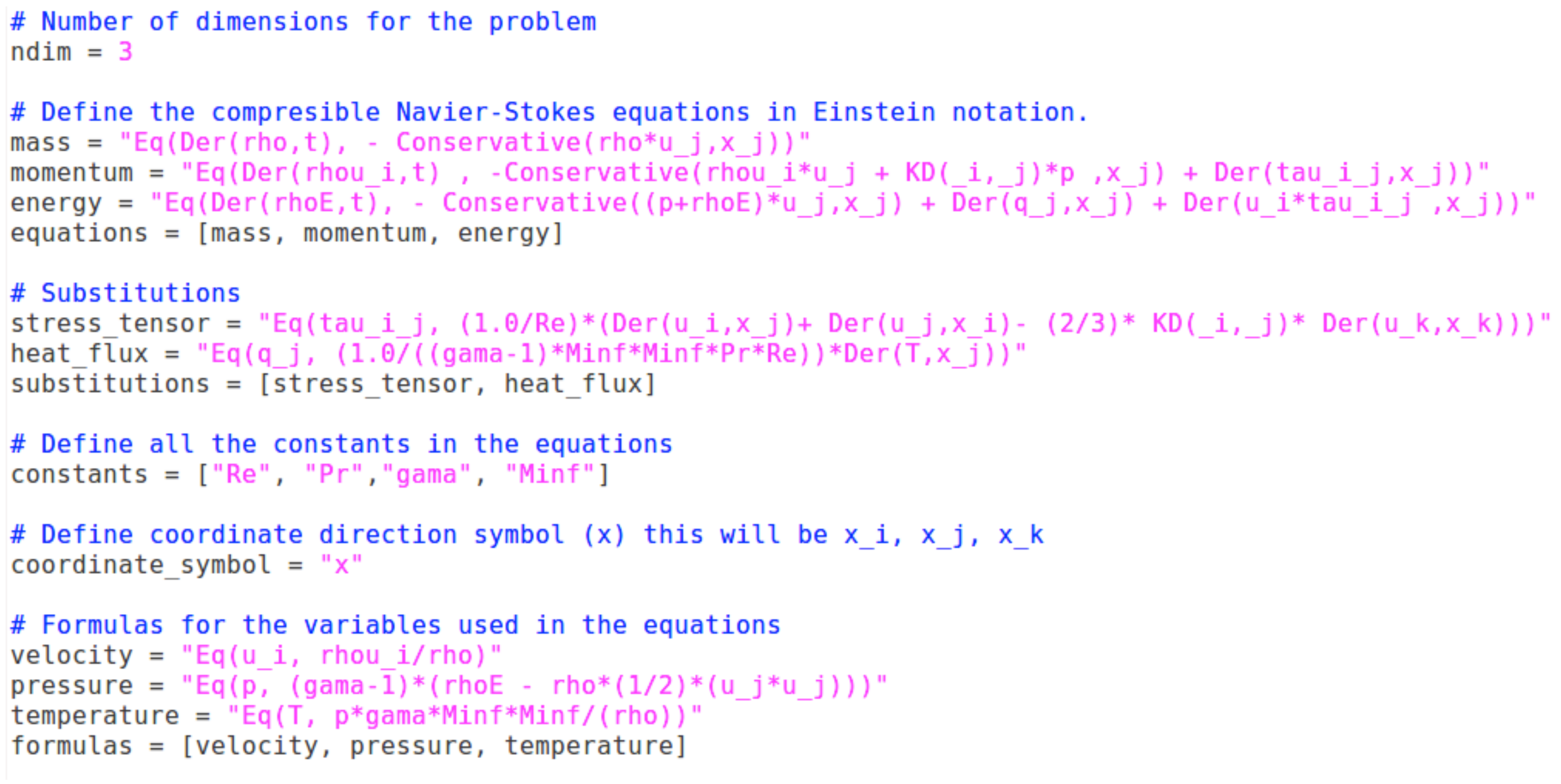
- User describes the problem at a higher level.
- Numerical analyst develops the numerical algorithm which generates
- a sequential model code in OPS-compliant C.
- Computer scientist handles parallel backend implementation.



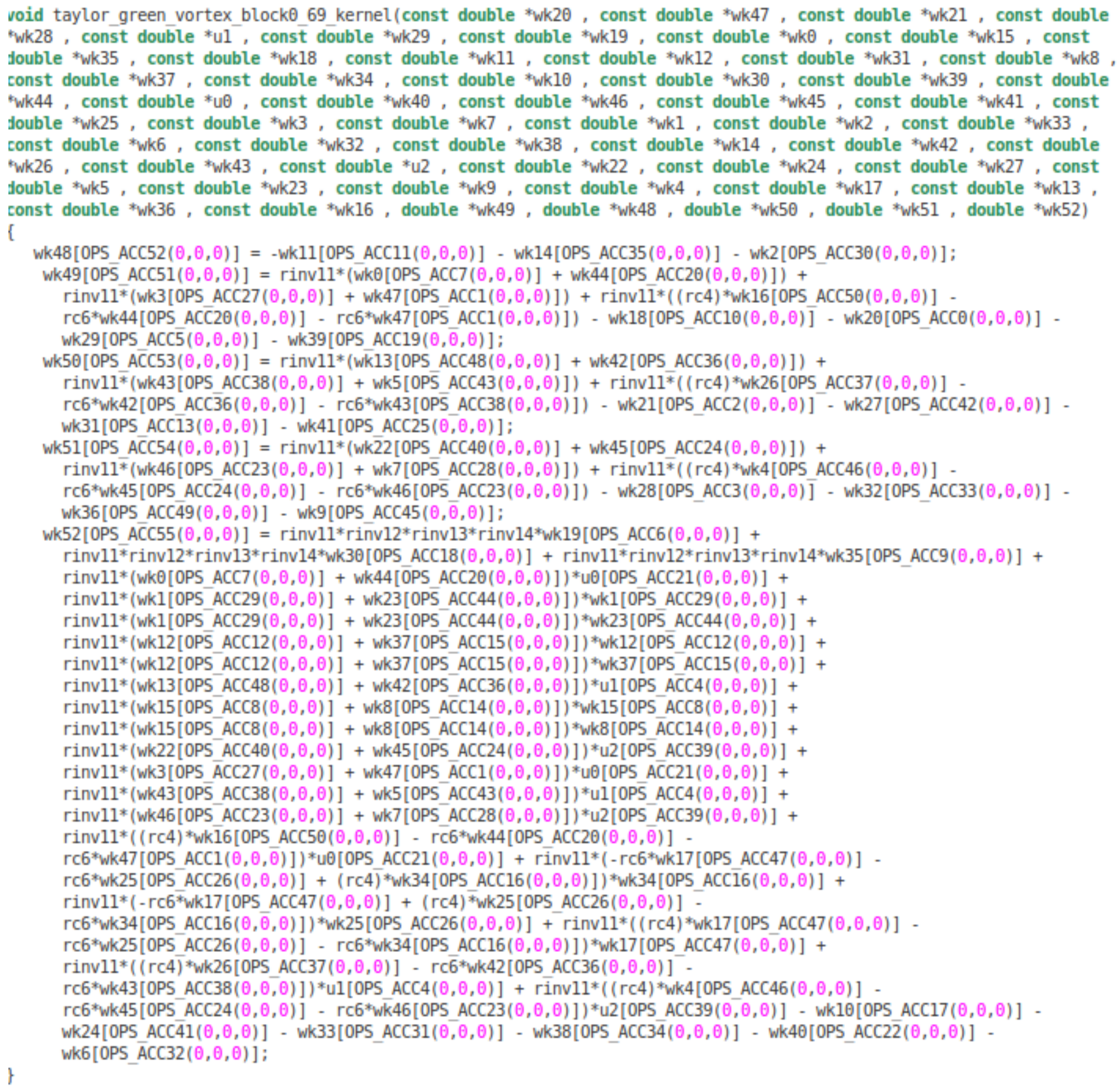
- Python, with SymPy building blocks
- Expand the summation indices of PDEs
- Automatic code generation
  - Apply spatial, temporal and boundary schemes
  - Create computational kernels
  - Generate OPSc code
  - Output LaTeX files of kernels for debugging

Example

- 50 line high-level problem definition for the 3D compressible Navier-Stokes equations
- 2000 line generated sequential OPS C code
- 20K lines of generated code for MPI and CUDA



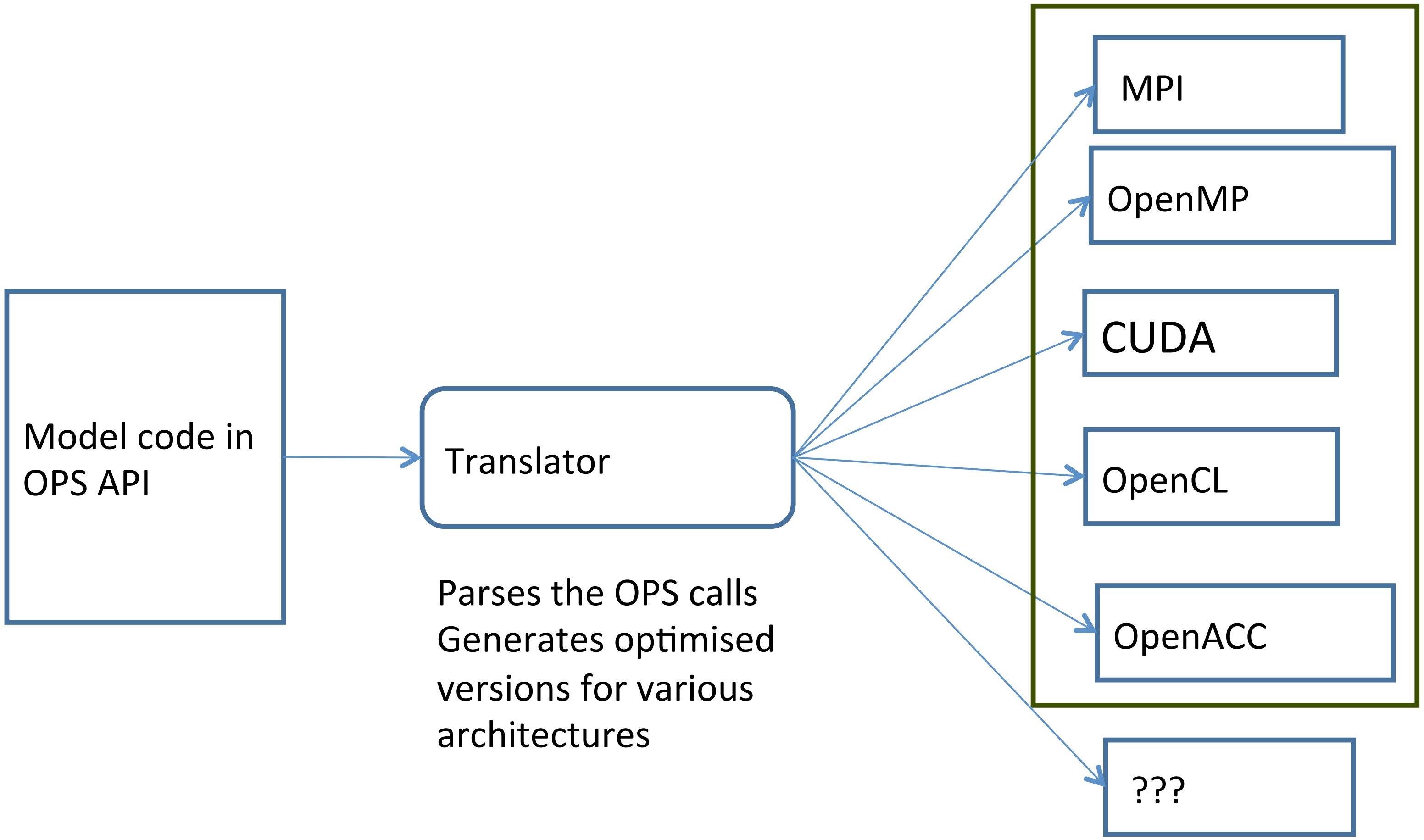
```
# Number of dimensions for the problem
ndim = 3

# Define the compresible Navier-Stokes equations in Einstein notation.
mass = "Eq(Der(rho,t), - Conservative(rho*u_j,x_j))"
momentum = "Eq(Der(rhou_i,t) , -Conservative(rhou_i*u_j + KD(_i,_j)*p ,x_j) + Der(tau_i_j,x_j))"
energy = "Eq(Der(rhoE,t), - Conservative((p+rhoE)*u_j,x_j) + Der(q_j,x_j) + Der(u_i*tau_i_j ,x_j))"
equations = [mass, momentum, energy]

# Substitutions
stress_tensor = "Eq(tau_i_j, (1.0/Re)*(Der(u_i,x_j)+ Der(u_j,x_i)- (2/3)* KD(_i,_j)* Der(u_k,x_k)))"
heat_flux = "Eq(q_j, (1.0/((gama-1)*Minf*Minf*Pr*Re))*Der(T,x_j))"
substitutions = [stress_tensor, heat_flux]

# Define all the constants in the equations
constants = ["Re", "Pr","gama", "Minf"]

# Define coordinate direction symbol (x) this will be x_i, x_j, x_k
coordinate_symbol = "x"

# Formulas for the variables used in the equations
velocity = "Eq(u_i, rhou_i/rho)"
pressure = "Eq(p, (gama-1)*(rhoE - rho*(1/2)*(u_j*u_j)))"
temperature = "Eq(T, p*gama*Minf*Minf/(rho))"
formulas = [velocity, pressure, temperature]
```



```
void taylor_green_vortex_block0_69_kernel(const double *wk20 , const double *wk47 , const double *wk21 , const double
*wk28 , const double *u1 , const double *wk29 , const double *wk19 , const double *wk0 , const double *wk15 , const
double *wk35 , const double *wk18 , const double *wk11 , const double *wk12 , const double *wk31 , const double *wk8 ,
const double *wk37 , const double *wk34 , const double *wk10 , const double *wk30 , const double *wk39 , const double
*wk44 , const double *u0 , const double *wk40 , const double *wk46 , const double *wk45 , const double *wk41 , const
double *wk25 , const double *wk3 , const double *wk7 , const double *wk1 , const double *wk2 , const double *wk33 ,
const double *wk6 , const double *wk32 , const double *wk38 , const double *wk14 , const double *wk42 , const double
*wk26 , const double *wk43 , const double *u2 , const double *wk22 , const double *wk24 , const double *wk27 , const
double *wk5 , const double *wk23 , const double *wk9 , const double *wk4 , const double *wk17 , const double *wk13 ,
const double *wk36 , const double *wk16 , double *wk49 , double *wk48 , double *wk50 , double *wk51 , double *wk52)
{
   wk48[OPS_ACC52(0,0,0)] = -wk11[OPS_ACC11(0,0,0)] - wk14[OPS_ACC35(0,0,0)] - wk2[OPS_ACC30(0,0,0)];
    wk49[OPS_ACC51(0,0,0)] = rinv11*(wk0[OPS_ACC7(0,0,0)] + wk44[OPS_ACC20(0,0,0)]) +
      rinv11*(wk3[OPS_ACC27(0,0,0)] + wk47[OPS_ACC1(0,0,0)]) + rinv11*((rc4)*wk16[OPS_ACC50(0,0,0)] -
      rc6*wk44[OPS_ACC20(0,0,0)] - rc6*wk47[OPS_ACC1(0,0,0)]) - wk18[OPS_ACC10(0,0,0)] - wk20[OPS_ACC0(0,0,0)] -
      wk29[OPS_ACC5(0,0,0)] - wk39[OPS_ACC19(0,0,0)];
    wk50[OPS_ACC53(0,0,0)] = rinv11*(wk13[OPS_ACC48(0,0,0)] + wk42[OPS_ACC36(0,0,0)]) +
      rinv11*(wk43[OPS_ACC38(0,0,0)] + wk5[OPS_ACC43(0,0,0)]) + rinv11*((rc4)*wk26[OPS_ACC37(0,0,0)] -
      rc6*wk42[OPS_ACC36(0,0,0)] - rc6*wk43[OPS_ACC38(0,0,0)]) - wk21[OPS_ACC2(0,0,0)] - wk27[OPS_ACC42(0,0,0)] -
      wk31[OPS_ACC13(0,0,0)] - wk41[OPS_ACC25(0,0,0)];
    wk51[OPS_ACC54(0,0,0)] = rinv11*(wk22[OPS_ACC40(0,0,0)] + wk45[OPS_ACC24(0,0,0)]) +
      rinv11*(wk46[OPS_ACC23(0,0,0)] + wk7[OPS_ACC28(0,0,0)]) + rinv11*((rc4)*wk4[OPS_ACC46(0,0,0)] -
      rc6*wk45[OPS_ACC24(0,0,0)] - rc6*wk46[OPS_ACC23(0,0,0)]) - wk28[OPS_ACC3(0,0,0)] - wk32[OPS_ACC33(0,0,0)] -
      wk36[OPS_ACC49(0,0,0)] - wk9[OPS_ACC45(0,0,0)];
    wk52[OPS_ACC55(0,0,0)] = rinv11*rinv12*rinv13*rinv14*wk19[OPS_ACC6(0,0,0)] +
      rinv11*rinv12*rinv13*rinv14*wk30[OPS_ACC18(0,0,0)] + rinv11*rinv12*rinv13*rinv14*wk35[OPS_ACC9(0,0,0)] +
      rinv11*(wk0[OPS_ACC7(0,0,0)] + wk44[OPS_ACC20(0,0,0)])*u0[OPS_ACC21(0,0,0)] +
      rinv11*(wk1[OPS_ACC29(0,0,0)] + wk23[OPS_ACC44(0,0,0)])*wk1[OPS_ACC29(0,0,0)] +
      rinv11*(wk1[OPS_ACC29(0,0,0)] + wk23[OPS_ACC44(0,0,0)])*wk23[OPS_ACC44(0,0,0)] +
      rinv11*(wk12[OPS_ACC12(0,0,0)] + wk37[OPS_ACC15(0,0,0)])*wk12[OPS_ACC12(0,0,0)] +
      rinv11*(wk12[OPS_ACC12(0,0,0)] + wk37[OPS_ACC15(0,0,0)])*wk37[OPS_ACC15(0,0,0)] +
      rinv11*(wk13[OPS_ACC48(0,0,0)] + wk42[OPS_ACC36(0,0,0)])*u1[OPS_ACC4(0,0,0)] +
      rinv11*(wk15[OPS_ACC8(0,0,0)] + wk8[OPS_ACC14(0,0,0)])*wk15[OPS_ACC8(0,0,0)] +
      rinv11*(wk15[OPS_ACC8(0,0,0)] + wk8[OPS_ACC14(0,0,0)])*wk8[OPS_ACC14(0,0,0)] +
      rinv11*(wk22[OPS_ACC40(0,0,0)] + wk45[OPS_ACC24(0,0,0)])*u2[OPS_ACC39(0,0,0)] +
      rinv11*(wk3[OPS_ACC27(0,0,0)] + wk47[OPS_ACC1(0,0,0)])*u0[OPS_ACC21(0,0,0)] +
      rinv11*(wk43[OPS_ACC38(0,0,0)] + wk5[OPS_ACC43(0,0,0)])*u1[OPS_ACC4(0,0,0)] +
      rinv11*(wk46[OPS_ACC23(0,0,0)] + wk7[OPS_ACC28(0,0,0)])*u2[OPS_ACC39(0,0,0)] +
      rinv11*((rc4)*wk16[OPS_ACC50(0,0,0)] - rc6*wk44[OPS_ACC20(0,0,0)] -
      rc6*wk47[OPS_ACC1(0,0,0)])*u0[OPS_ACC21(0,0,0)] + rinv11*(-rc6*wk17[OPS_ACC47(0,0,0)] -
      rc6*wk25[OPS_ACC26(0,0,0)] + (rc4)*wk34[OPS_ACC16(0,0,0)])*wk34[OPS_ACC16(0,0,0)] +
      rinv11*(-rc6*wk17[OPS_ACC47(0,0,0)] + (rc4)*wk25[OPS_ACC26(0,0,0)] -
      rc6*wk34[OPS_ACC16(0,0,0)])*wk25[OPS_ACC26(0,0,0)] + rinv11*((rc4)*wk17[OPS_ACC47(0,0,0)] -
      rc6*wk25[OPS_ACC26(0,0,0)] - rc6*wk34[OPS_ACC16(0,0,0)])*wk17[OPS_ACC47(0,0,0)] +
      rinv11*((rc4)*wk26[OPS_ACC37(0,0,0)] - rc6*wk42[OPS_ACC36(0,0,0)] -
      rc6*wk43[OPS_ACC38(0,0,0)])*u1[OPS_ACC4(0,0,0)] + rinv11*((rc4)*wk4[OPS_ACC46(0,0,0)] -
      rc6*wk45[OPS_ACC24(0,0,0)] - rc6*wk46[OPS_ACC23(0,0,0)])*u2[OPS_ACC39(0,0,0)] - wk10[OPS_ACC17(0,0,0)] -
      wk24[OPS_ACC41(0,0,0)] - wk33[OPS_ACC31(0,0,0)] - wk38[OPS_ACC34(0,0,0)] - wk40[OPS_ACC22(0,0,0)] -
      wk6[OPS_ACC32(0,0,0)];
}
```

OPSc Example of auto-generated kernel for computing residual of Compressible Navier-Stokes solution

Future-proofing with OPS



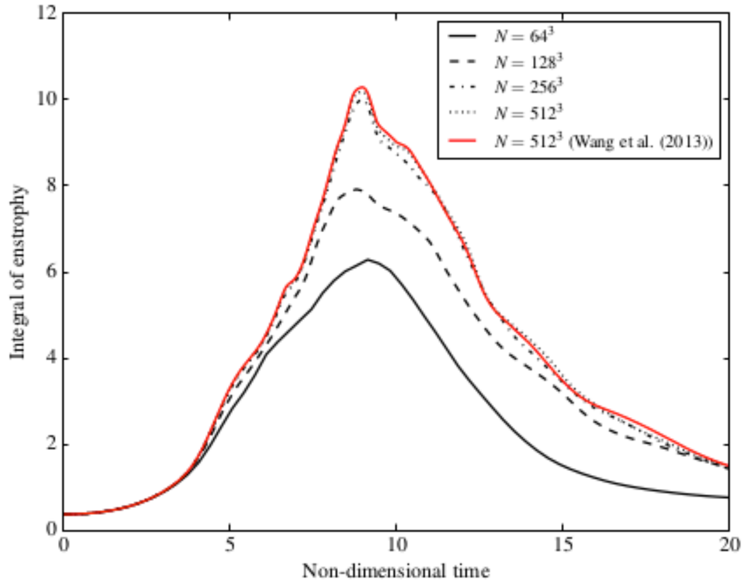
Parses the OPS calls
Generates optimised versions for various architectures

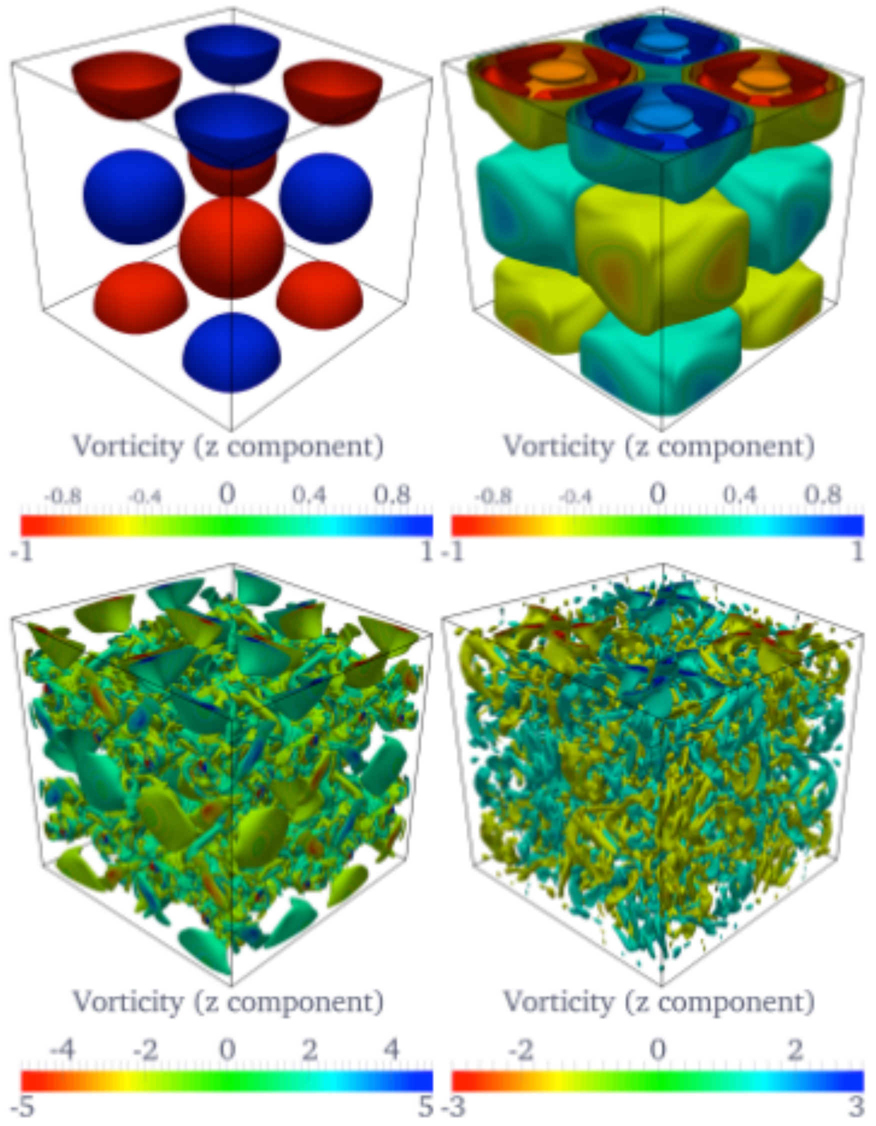
Source code remains unchanged

Newer architectures require backend translator to be written

Verification & Validation (OpenSBLIv1)

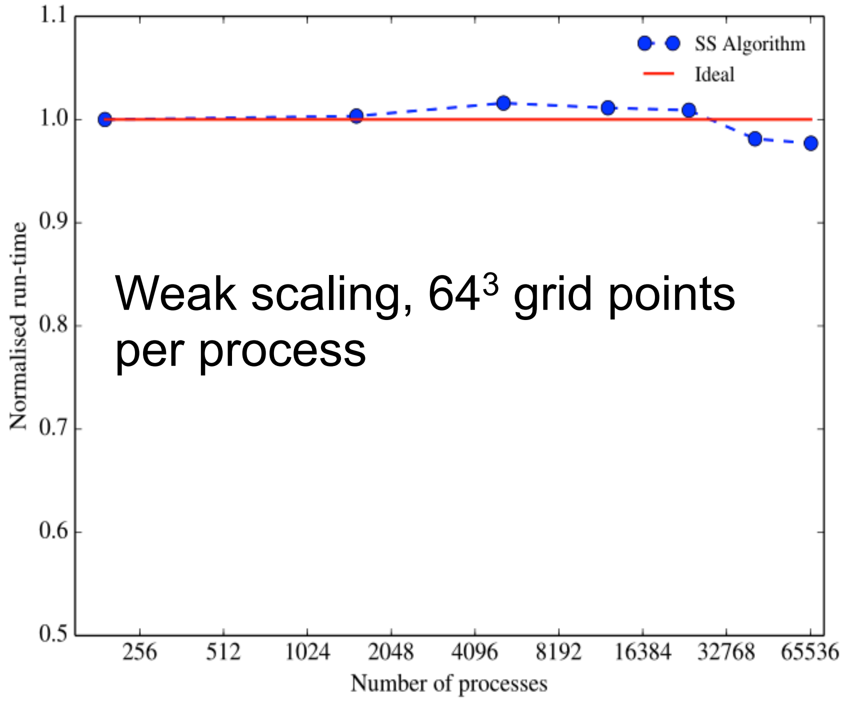
- 3D Taylor-Green vortex
- N-S Equations
- Re = 1600
- CPU(ARCHER), GPU(K40c)



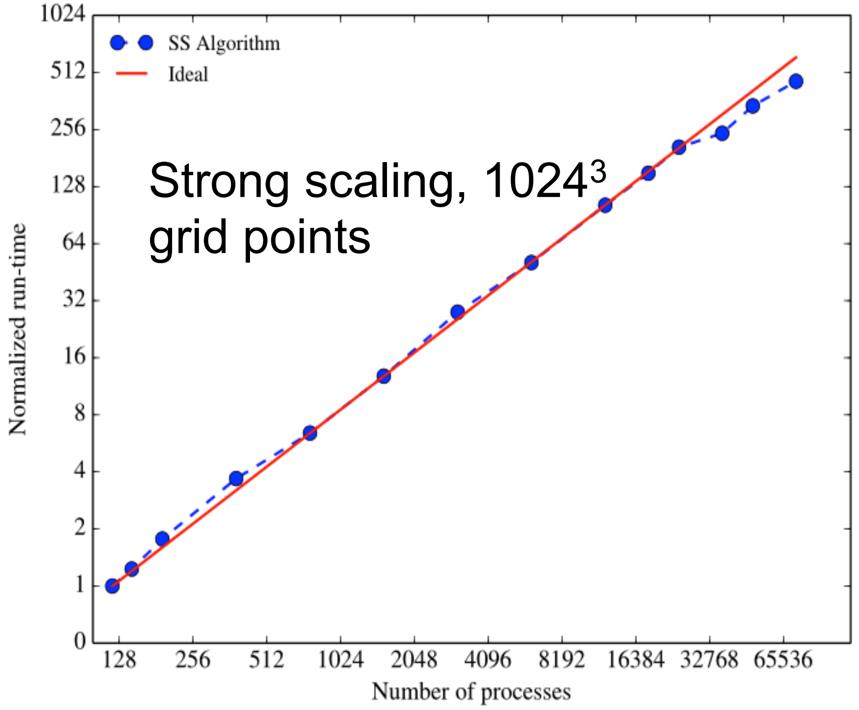


Parallel scaling on CPU

Cray XC-30 (ARCHER)



Weak scaling, 64^3 grid points per process



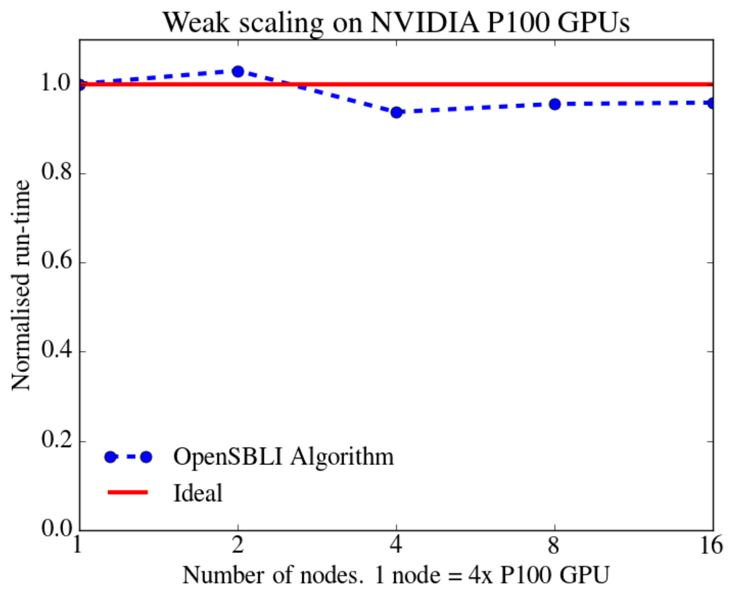
Strong scaling, 1024^3 grid points

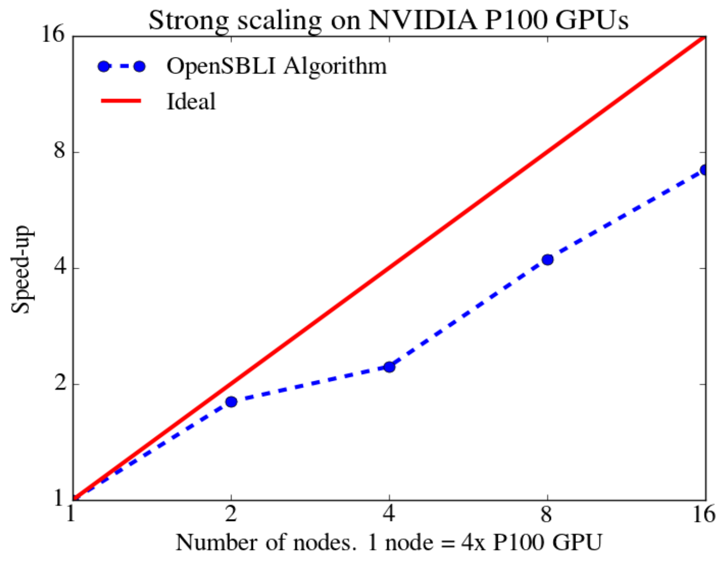
Extrapolation to Exascale?

- Flops constrained by RAM (limited size/bandwidth)?
- Algorithmic changes to exploit the flops by reducing memory usage and data transfers?

Parallel scaling MPI+CUDA

- **Good weak scaling up to 64 P100 GPUs:**
  3D Taylor-Green vortex case, meshes ranging from 277 million to 4.3 billion points.



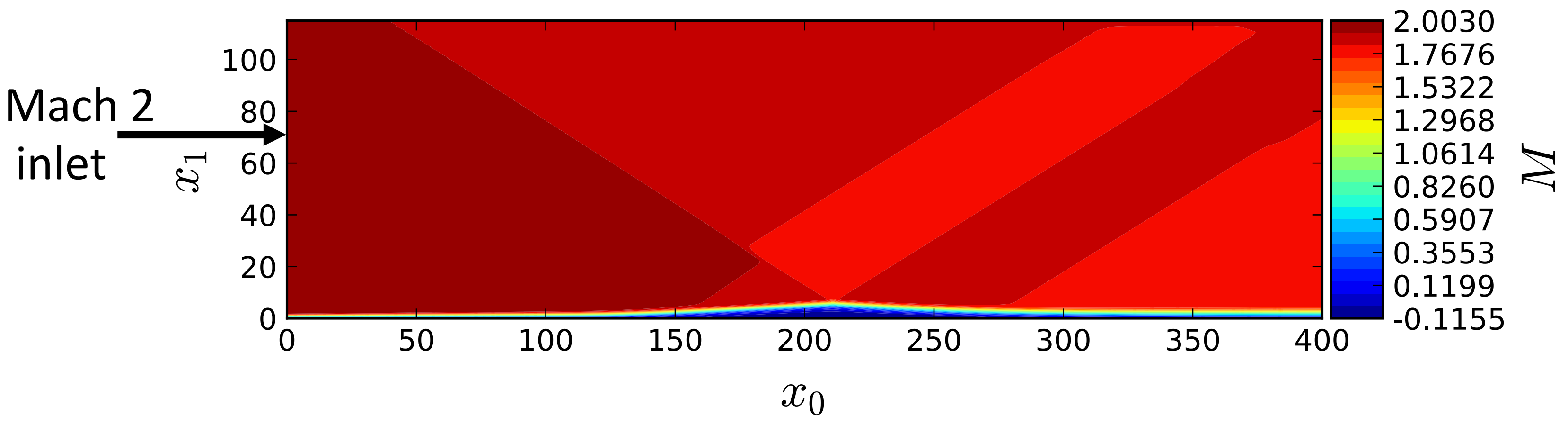


1024^3 grid points per node

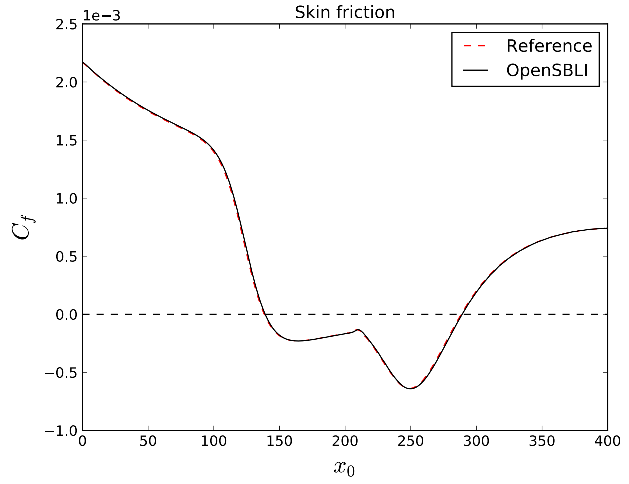
NVIDIA P100, Mellanox EDR Infiniband
(Cambridge CSD3 Wilkes2)

Verification & Validation (OpenSBLIv2)

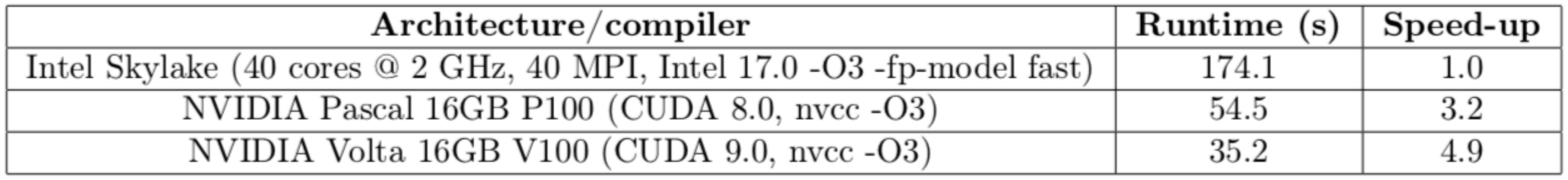
Shock-wave boundary-layer interaction Katzer case



Target architecture (processes/ threads)	Time (s)	Speed-up
ARCHER node CPU – Ivy Bridge (24 MPI)	413.1	1.00
Intel Xeon Phi KNL 7210 (64 MPI)	224.7	1.84
1x GPU NVIDIA Tesla K40	204.6	2.02
1x GPU NVIDIA Tesla P100	44.0	9.39



Architecture performance comparison



Architecture/compiler	Runtime (s)	Speed-up
Intel Skylake (40 cores @ 2 GHz, 40 MPI, Intel 17.0 -O3 -fp-model fast)	174.1	1.0
NVIDIA Pascal 16GB P100 (CUDA 8.0, nvcc -O3)	54.5	3.2
NVIDIA Volta 16GB V100 (CUDA 9.0, nvcc -O3)	35.2	4.9

Table 1: OPS single node runtime comparison on different architectures for 100 iterations. The time for the CPU node (Intel Skylake Xeon Gold 6138) with 40 MPI processes is taken as the baseline.

- Single GPU performance on NVIDIA P100/V100 compared to Intel Skylake CPU node
- 3D SBLI simulation with 16 million points
- NVIDIA V100 is ~5x faster than a 40 core Skylake node

Flexible algorithms

The main limitation of GPUs is the memory capacity (16GB per P100 vs 196GB per CPU node):
Difficult to fit large enough problems on each GPU.
Code-generation gives greater flexibility in how the code is written -> recompute quantities on the fly to reduce work arrays and memory access.

- Exploit the OpenSBLI framework to compare different algorithmic choices without rewriting low-level code

- ❑ **Baseline (BL)** – all the derivatives are stored in work arrays
- ❑ **Recompute All (RA)** – all continuous derivatives in the governing equations are replaced by their discretised formula
- ❑ **Recompute Some (RS)** – only the first derivatives of velocity are stored and the rest recomputed
- ❑ **Store None (SN)** – all the derivatives are evaluated as thread/process local variables
- ❑ **Store Some (SS)** – only the first derivatives of velocity are stored and the rest are evaluated as thread/process local variables

Implementation

- Taylor-Green vortex problem at Re=1600 (64^3 to 256^3)

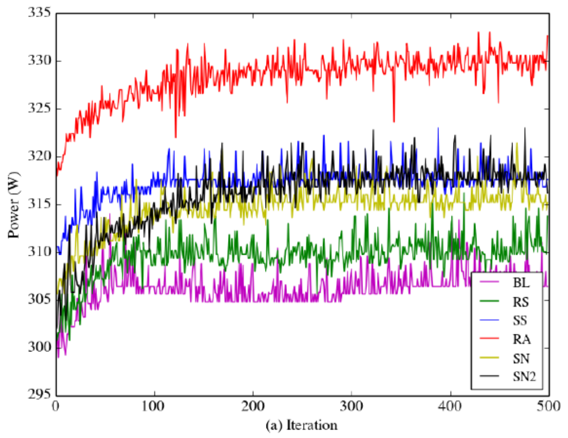
Per RK substep	Baseline	Recompute All	Recompute Some	Store None	Store Some
Kernel calls	87	4	12	4	12
Local variables	0	0	0	63	53
Work arrays	67	5	14	5	14

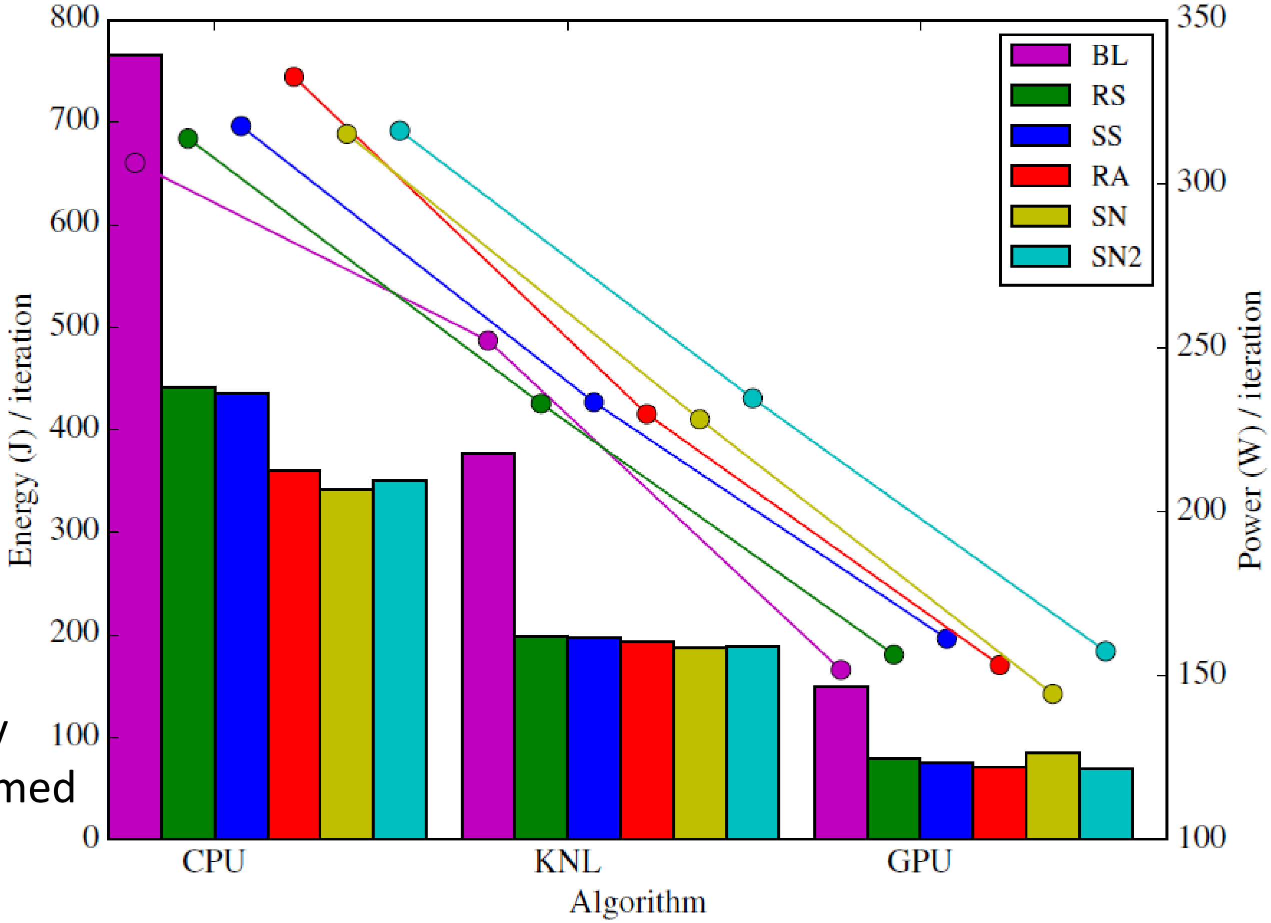
Algorithmic performance

	CUDA Tesla K40c, runtime (s)				
Grid Size (Millions)	Baseline	Recompute All	Recompute Some	Store None	Store Some
0.2	9	6	6	6	5
2.09	57	35	35	41	33
16.77	495	259	256	302	246

	ARCHER node (24 MPI processes), runtime (s)				
Grid Size (Millions)	Baseline	Recompute All	Recompute Some	Store None	Store Some
0.2	16	9	11	8	10
2.09	183	98	97	91	89
16.77	1562	765	803	694	685

Power consumption and energy efficiency





Lines with symbols=power consumed

Bars = energy consumed

Jacobs et al ParCFD, 2017 ECCOMAS 2018

Advantages and limitations of the automated code-generation approach

- ✓ New DSLs can be readily integrated
- ✓ Flexibility of algorithms, methods and equations
  - ✓ Run time and energy efficiency
- ✓ External libraries (e.g. FFT) and implicit solvers need to be implemented in both OpenSBLI and OPS
- ❖ Debugging for errors at different levels may be more difficult (partially mitigated by LaTeX debugging)

- Outlook:
  - Separation of concerns ***should*** enable better software maintainability
  - Some flexibility to match algorithms to architectures, looking towards exascale
  - Open source under GNU GPL: https://opensbli.github.io