DEPARTMENT OF CHEMICAL ENGINEERING
COLLEGE OF ENGINEERING | THE UNIVERSITY OF UTAH

Institute for CLEAN AND SECURE ENERGY
THE UNIVERSITY OF UTAH

Flexible, Efficient Abstractions for High Performance Computation on Current and Emerging Architectures

JAMES C. SUTHERLAND
Associate Professor - Chemical Engineering

MATT MIGHT
Assistant Professor - School of Computing

TONY SAAD
Research Associate

CHRISTOPHER EARL
Postdoctoral Researcher

ABISHEK BAGUSETTY
M.S. Student

US DOE award DE-NA0000740

NSF PetaApps award 0904631

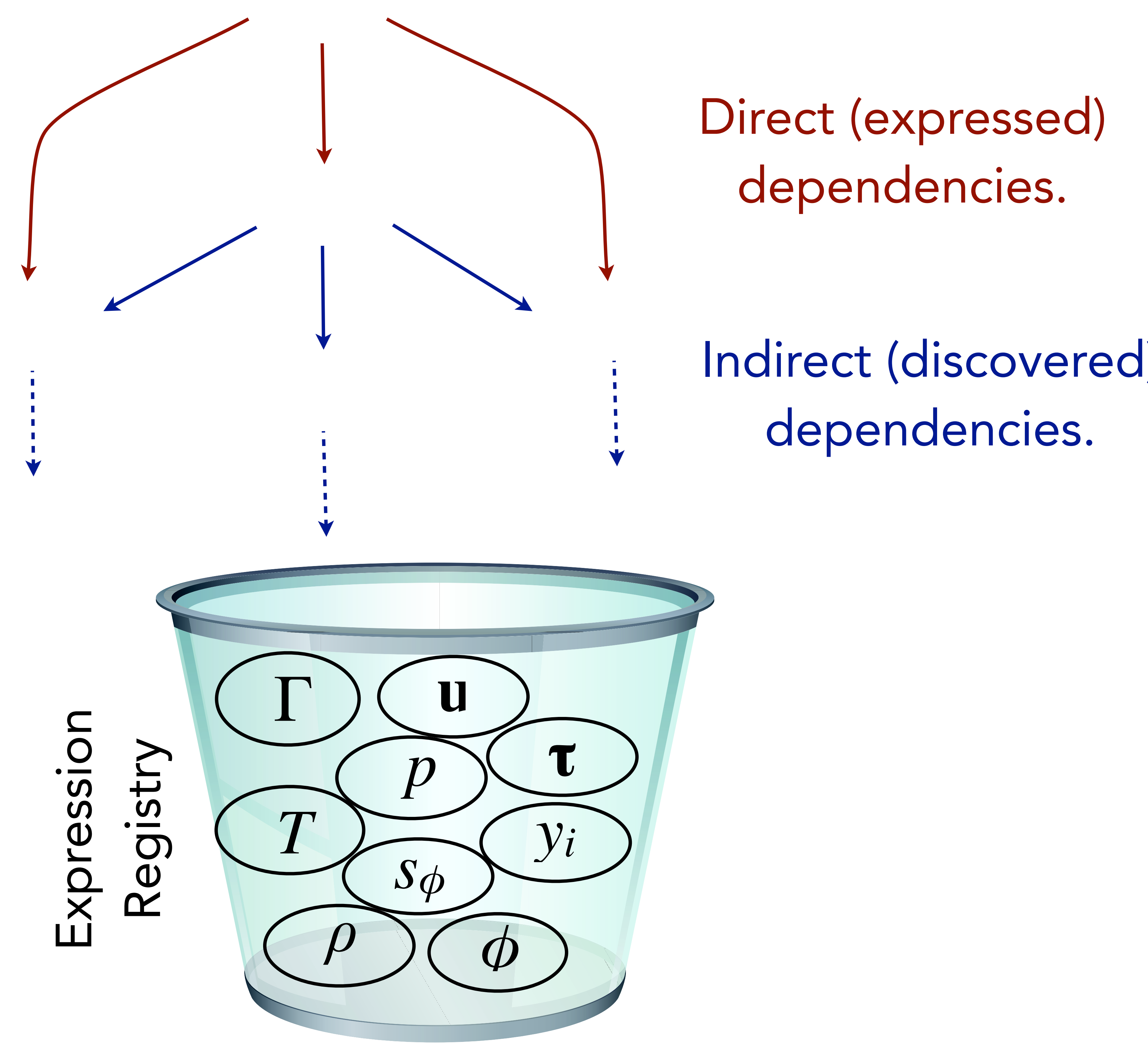
Motivation

- Complex physics ⇒ complex software!
  - *is this necessary, or just a result of looking at the problem in the wrong way?*
- Changing from model "A" to model "B" ...
  - *may require different transport equations*
  - *may introduce different nonlinear coupling*
- Spatial discretization frequently permeates software design
  - *Model developers typically must deal with "mesh loops"*
    - often resort to "copy/paste/modify" tactics that are highly bug-prone
  - *Future proofing:*
    - What if you want to do OpenMP on these loops?
    - What happens when you learn that OpenMP is not the right tool?
    - pthreads, CUDA / OpenCL ... ?
- Questions: Can we write efficient software that...
  - *... naturally handles complexity and allows us to easily extend/replace existing models?*
  - *... allows programmers to easily and robustly express intent while not worrying about "details?"*
  - *... allows us to refactor for different hardware architectures without rewriting the code base?*

Flexible…

- Register all expressions
  - *Each "expression" calculates one or more field quantities.*
  - *Each expression advertises its direct dependencies.*
- Set a "root" expression; construct a graph
  - *All dependencies are discovered/resolved automatically.*
  - *Highly localized influence of changes in models.*
  - *Not all expressions in the registry may be relevant/used.*
- From the graph:
  - *Deduce storage requirements & allocate memory (externally to each expression).*
  - *Automatically schedule evaluation, ensuring proper ordering.*
    - Asynchronous execution is critical! (overlap communication & computation)
  - *Robust scheduling algorithms are key.*

$$\Gamma = \Gamma(T, p, y_i)$$

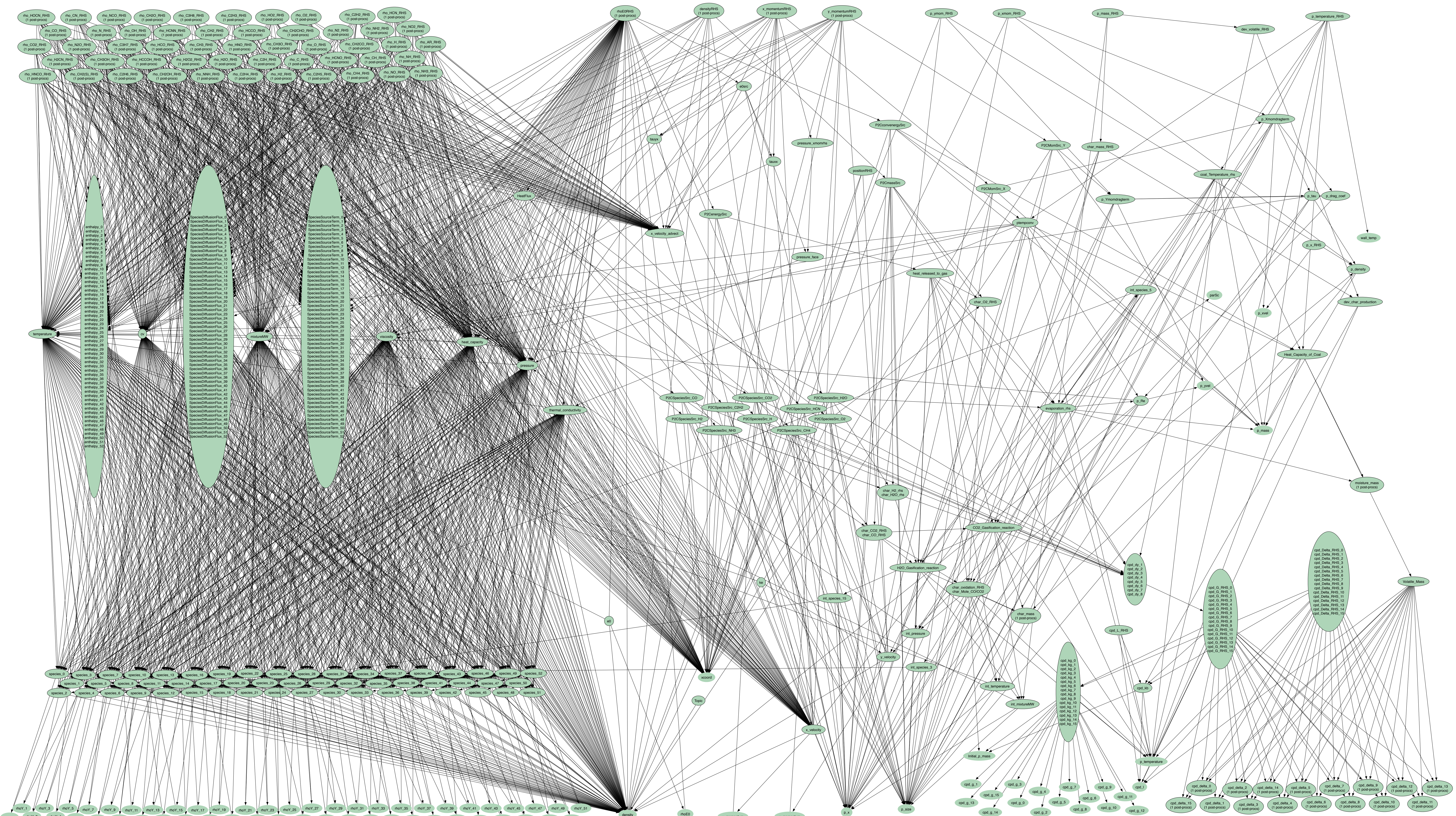


*Notz, Pawlowski, & Sutherland (2012). ACM Transactions on Mathematical Software, 39(1).

Institute for CLEAN AND SECURE ENERGY
THE UNIVERSITY OF UTAH

Example: coal combustion

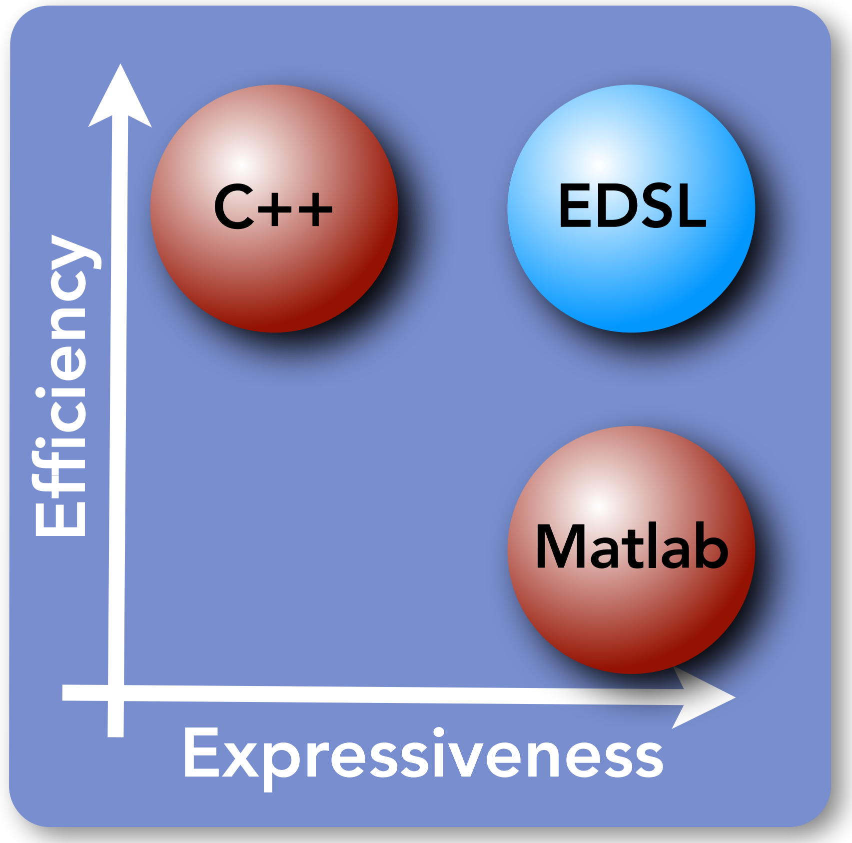
- 55 PDEs
- ~35 ODEs per particle
- Complex interphase coupling

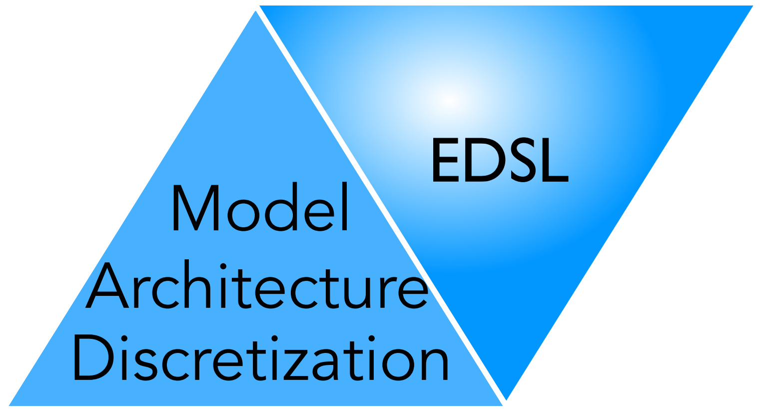


DEPARTMENT OF CHEMICAL ENGINEERING
COLLEGE OF ENGINEERING | THE UNIVERSITY OF UTAH

Efficient…

- Expressive syntax (matlab-style array operations)
  - *Programmer expresses intent (problem structure) - not implementation.*
- High performance
  - *Should match hand-tuned code in performance.*
- Extensible
  - *Insulate programmer from architecture changes (e.g. multicore → GPU → …).*
  - *EDSL "back-end" compiles into code for target architecture.*
- "Plays well with others"
  - *Allow programmer to write in C++ and inter-operate with EDSL.*
  - *Not an "all-or-none" approach: enable incremental adoption.*
  - *Allows concurrent development of EDSL and application codes.*





Field Expressions

$$\vec{c} = \vec{a} + sin(\vec{b})$$

Manual C++

Thread 1

```
Field::const_iterator ia1 = a1.begin();
Field::const_iterator ib1 = b1.begin();
for(Field::iterator ic1 = c1.begin();
    ic1 != c1.end();
    ++ic1, ++ia1, ++ib1) {
  *ic1 = *ia1 + sin(*ib1);
};
...
```

⋮

Thread n

```
Field::const_iterator ian = an.begin();
Field::const_iterator ibn = bn.begin();
for(Field::iterator icn = cn.begin();
    icn != cn.end();
    ++icn, ++ian, ++ibn) {
  *icn = *ian + sin(*ibn);
};
```

Nebo EDSL

```
c <<= a + sin(b);
```

- Data parallel handled internally.
- Thread deployment (resizable threadpool).
- GPU deployment.
- Compile-time guarantee of field compatibility for given operations.

Chained Stencil Operations

$$\phi = -\nabla \cdot \mathbf{q}$$
$$= \nabla \cdot (\lambda \nabla T)$$

```
phi <<= divX( interpX(lambda) * gradX(temperature) )
      + divY( interpY(lambda) * gradY(temperature) )
      + divZ( interpZ(lambda) * gradZ(temperature) );
```

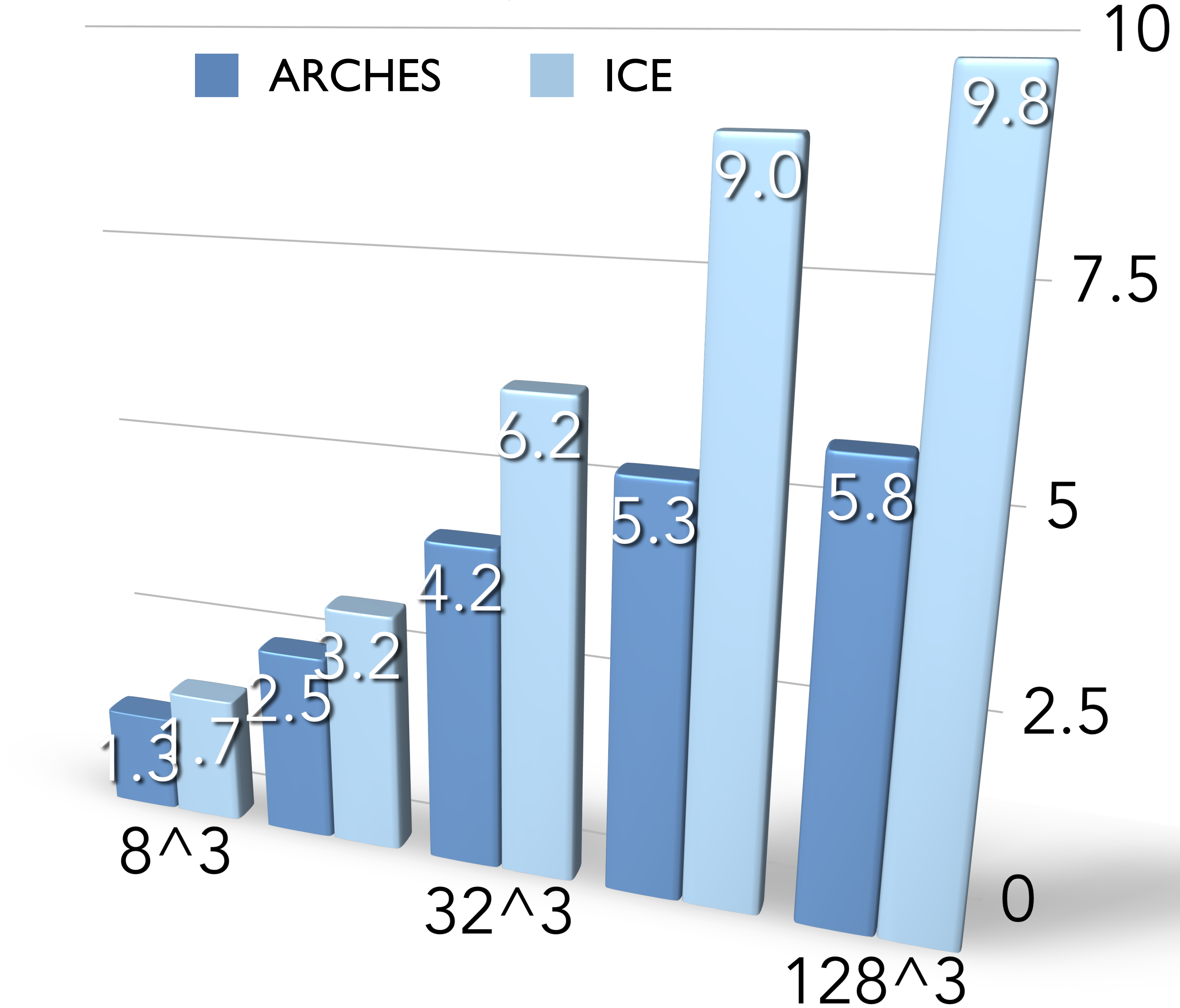
```
// field type inference:
typedef FaceTypes<FieldT>::XFace  XFluxT;
typedef FaceTypes<FieldT>::YFace  YFluxT;
typedef FaceTypes<FieldT>::ZFace  ZFluxT;

// operator type inference:
typedef OpTypes<FieldT>::DivX  DivX;
typedef OpTypes<FieldT>::DivY  DivY;
typedef OpTypes<FieldT>::DivZ  DivZ;
```

- One inlined grid loop, no temporaries.
- Better performance & scalability than without chaining.
- Compile-time consistency checking (field-operator and field-field compatibility).
- Runtime consistency checks for ghost cell validity.

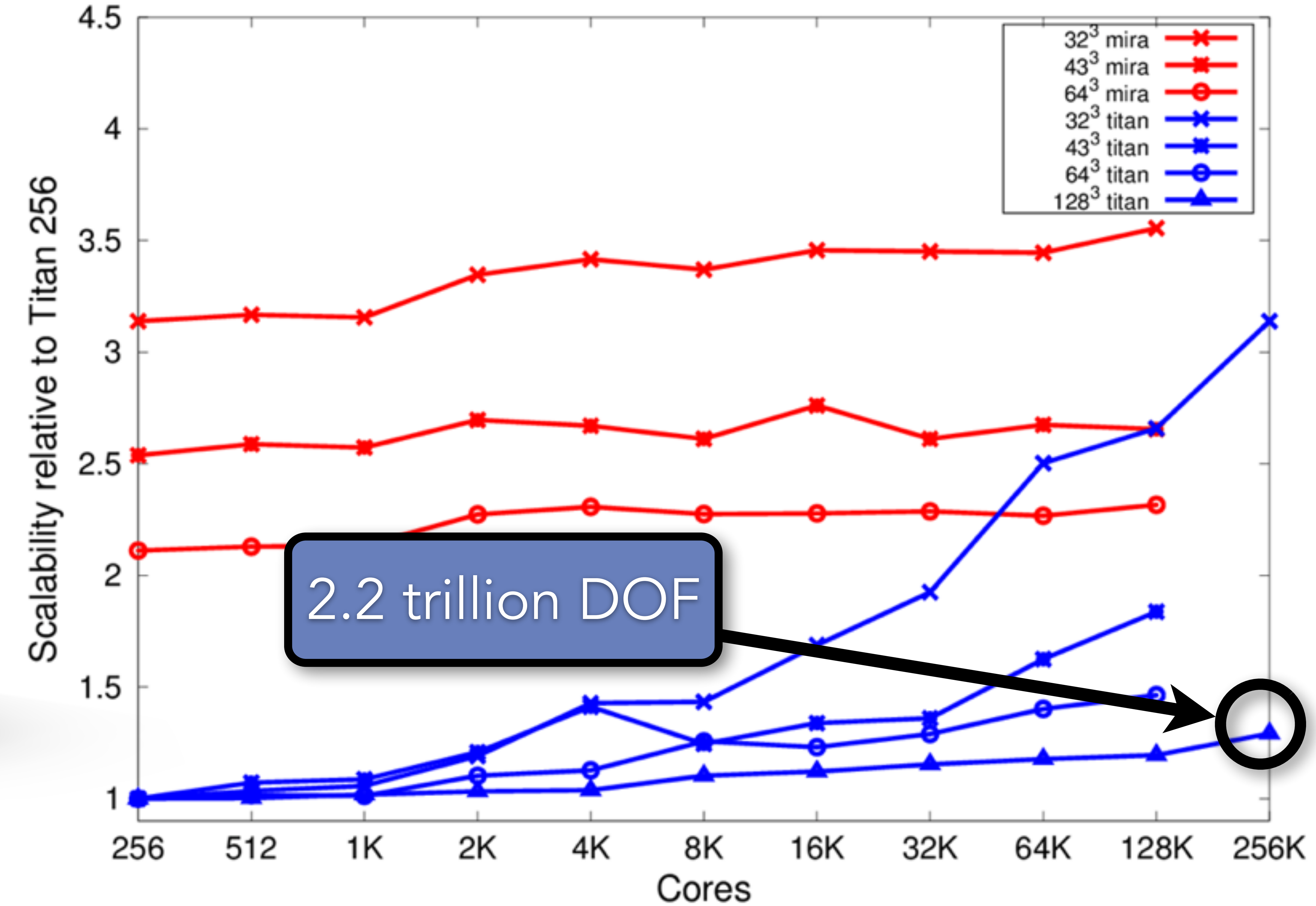
Putting it Together: Performance & Scalability

Speedup using DSL* relative to other Uintah codes



*Comparison to ICE and ARCHES, sister codes in Uintah, on a 3D Taylor-Green vortex problem. Run on a single processor.

Weak scaling on Titan



"1" indicates perfect weak scaling

Multicore & GPU Performance

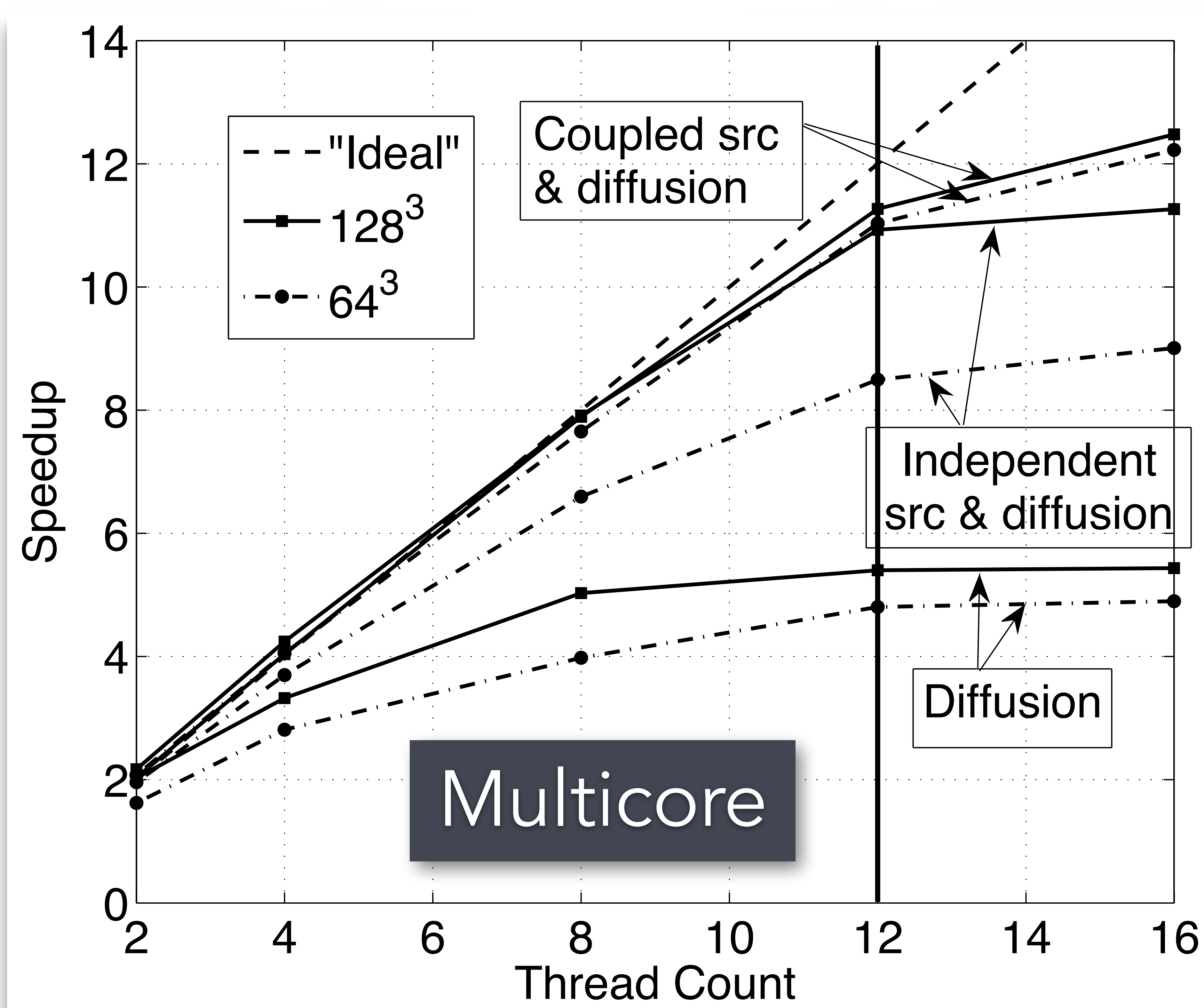
Test: mockup of a diffusion-reaction problem.

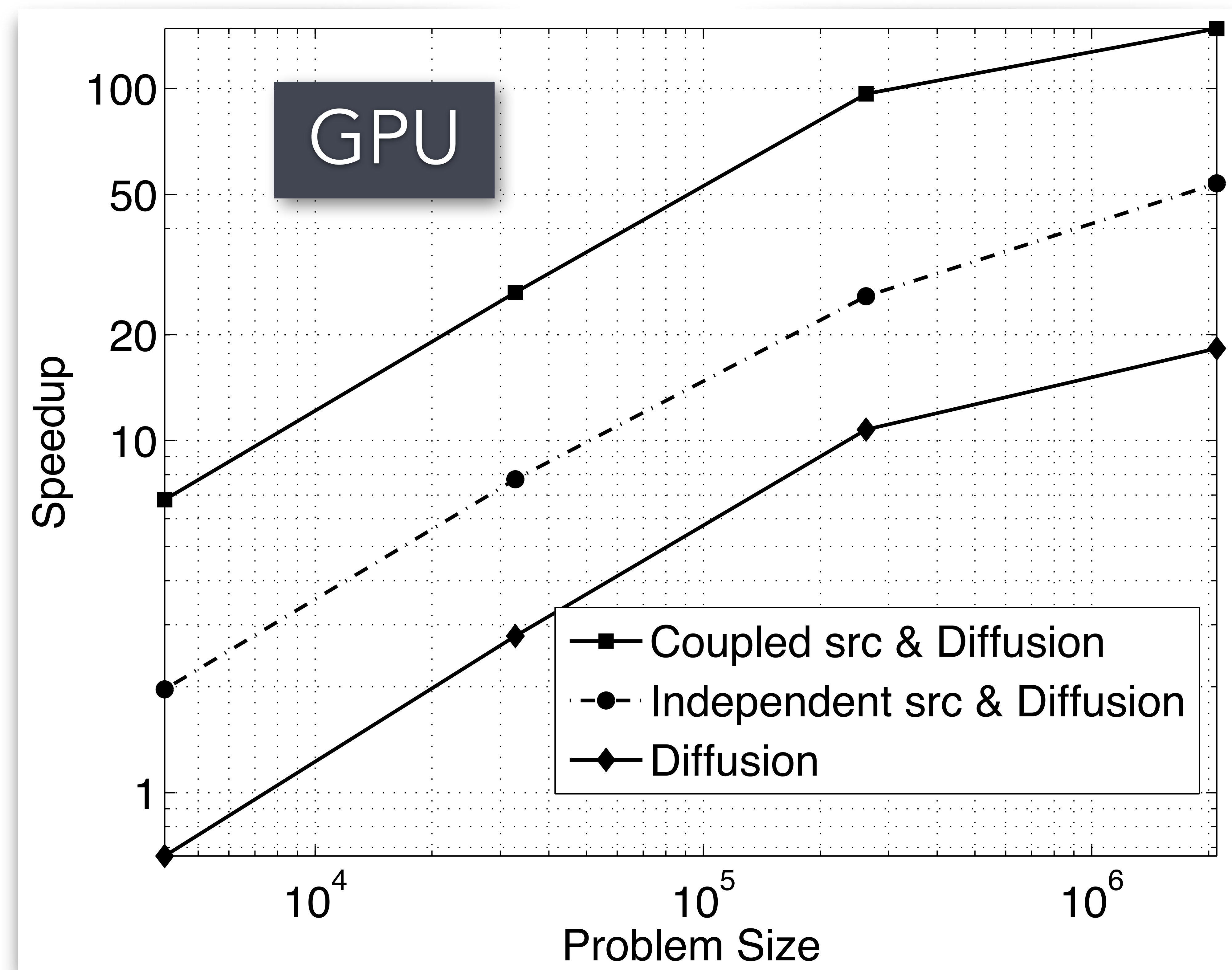
- Easily dial in the number of equations (30 here).
- Diffusion is an inexpensive stencil calculation.
- Reaction is an expensive point-wise calculation.

$$\frac{\partial \phi_i}{\partial t} = -\nabla \cdot \mathbf{J}_i + s_i$$

$$\mathbf{J}_i = -\Gamma \nabla \phi_i$$

$$s_i = f(\phi_i) \text{ or } s_i = f(\phi_j)$$





Parting Thoughts

- Hierarchical parallelization allows for flexible usage of available resources:
  - *Domain decomposition (SIMD)*
    - Should allow a process to do computation on "interior" while waiting on communication from neighbors.
  - *Task decomposition (MIMD)*
    - Decompose the solution into a DAG that can be scheduled asynchronously.
  - *Vectorized parallel (SIMD)*
    - Break grid operations across multicore, GPU, etc.
- DAG representation is a scalable abstraction that:
  - *Handles problem complexity gracefully.*
  - *Provides convenient separation of the problem's structure from the data.*
  - *Allows sophisticated scheduling algorithms to optimize scalability & performance.*
- (E)DSLs are very useful
  - *Future-proofing: separate intent from implementation.*
  - ***E**DSLs allow seamless transition of a code base and leverage existing compilers.*
  - *Template metaprogramming pushes work from run-time to compile-time for more efficiency.*

Institute for CLEAN AND SECURE ENERGY
THE UNIVERSITY OF UTAH
DEPARTMENT OF CHEMICAL ENGINEERING
COLLEGE OF ENGINEERING | THE UNIVERSITY OF UTAH