The Stanford Pervasive Parallelism Lab

Christos Kozyrakis and Kunle Olukotun

http://ppl.stanford.edu

The PPL Team

- Applications
  - Ron Fedkiw, Vladlen Koltun, Sebastian Thrun
- Programming & software systems
  - Alex Aiken, Pat Hanrahan, John Ousterhout, Mendel Rosenblum
- Architecture
  - Bill Dally, John Hennessy, Mark Horowitz, Christos Kozyrakis, Kunle Olukotun (director)

Goals and Organization

- Goal: the parallel computing platform for 2015
  - Parallel application development practical for the masses
    - Joe the programmer…
  - Parallel applications without parallel programming
- PPL is a collaboration of
  - Leading Stanford researchers across multiple domains
    - Applications, languages, software systems, architecture
  - Leading companies in computer systems and software
    - Sun, AMD, Nvidia, IBM, Intel, NEC, HP
- PPL is open
  - Any company can join; all results in the public domain

What Makes Parallel Programming Difficult?

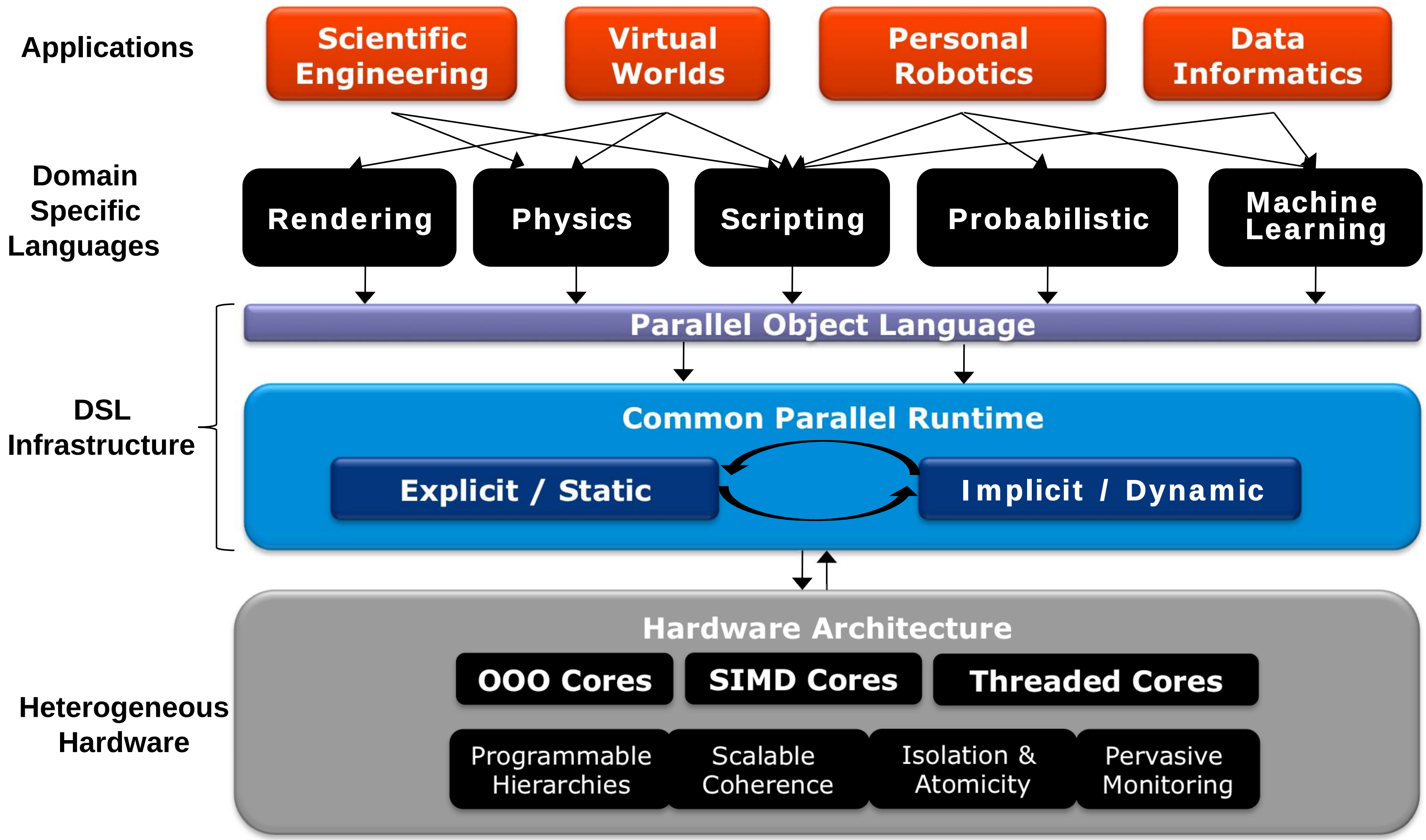
1. Finding independent tasks
2. Mapping tasks to execution units
3. Implementing synchronization
   - Races, livelocks, deadlocks, ...
4. Composing parallel tasks
5. Recovering from HW & SW errors
6. Optimizing locality and communication
7. Predictable performance & scalability
8. ... and all the sequential programming issues

- Even with new tools, can Joe handle these issues?

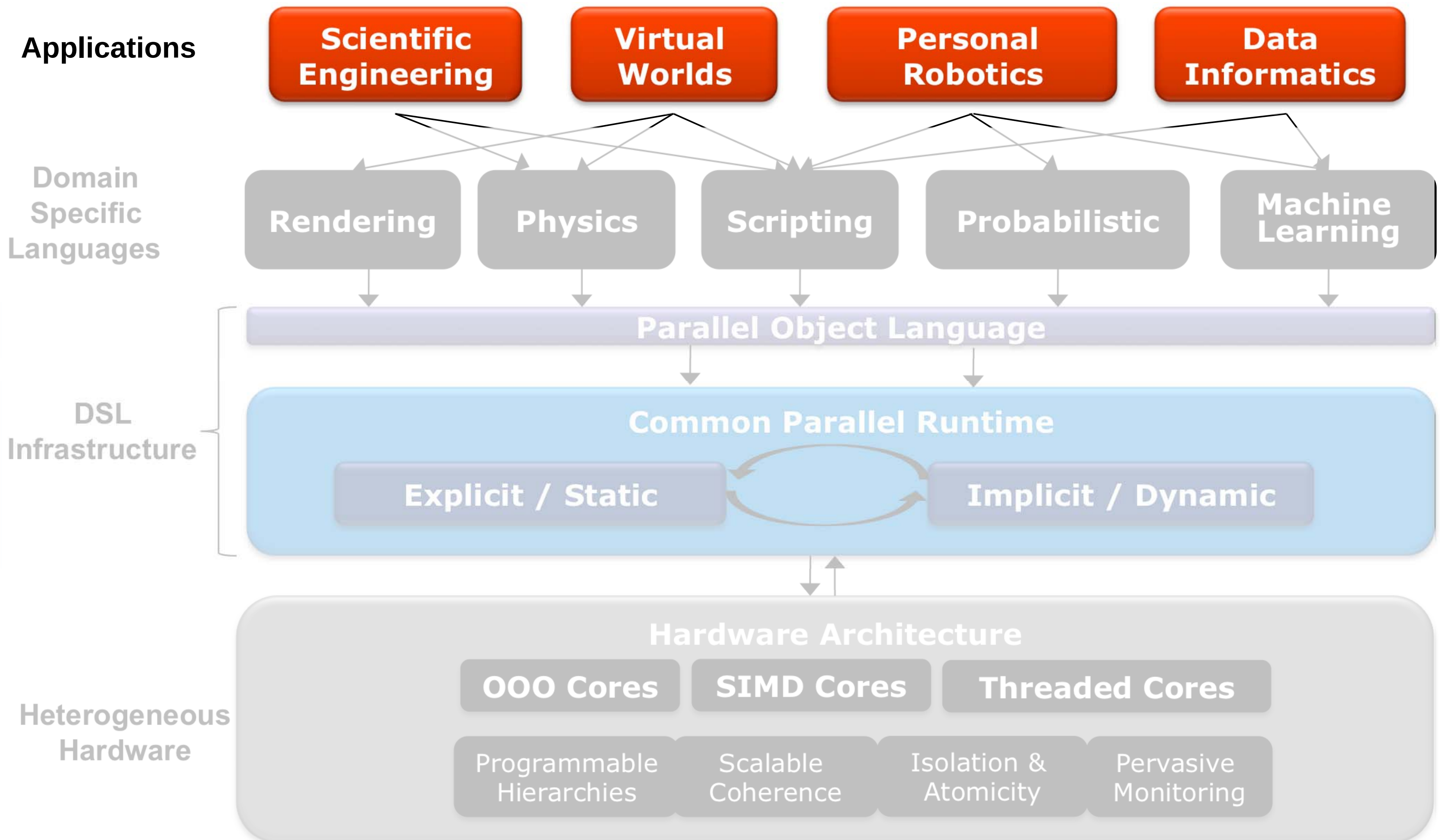
Technical Approach

- Guiding observations
  - Must hide low-level issues from programmer
  - No single discipline can solve all problems
  - Top-down research driven by applications
- Core techniques
  - Domain specific languages (DSLs)
    - Simple & portable programs
  - Heterogeneous hardware
    - Energy and area efficient computing

The PPL Vision

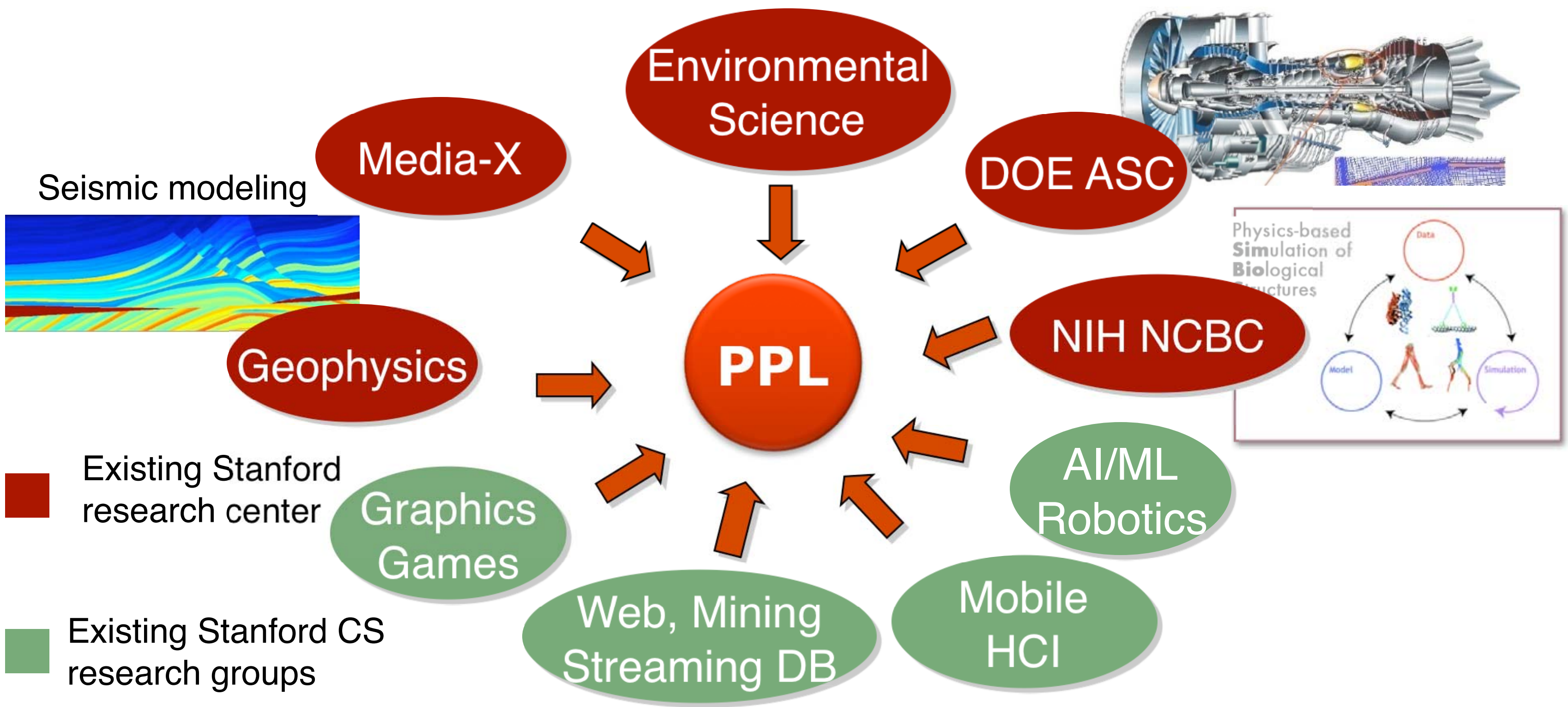


Applications



Demanding Applications

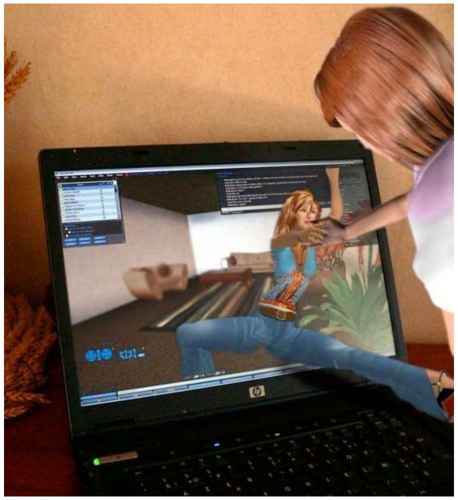
PERVASIVE PARALLELISM LABORATORY PPL



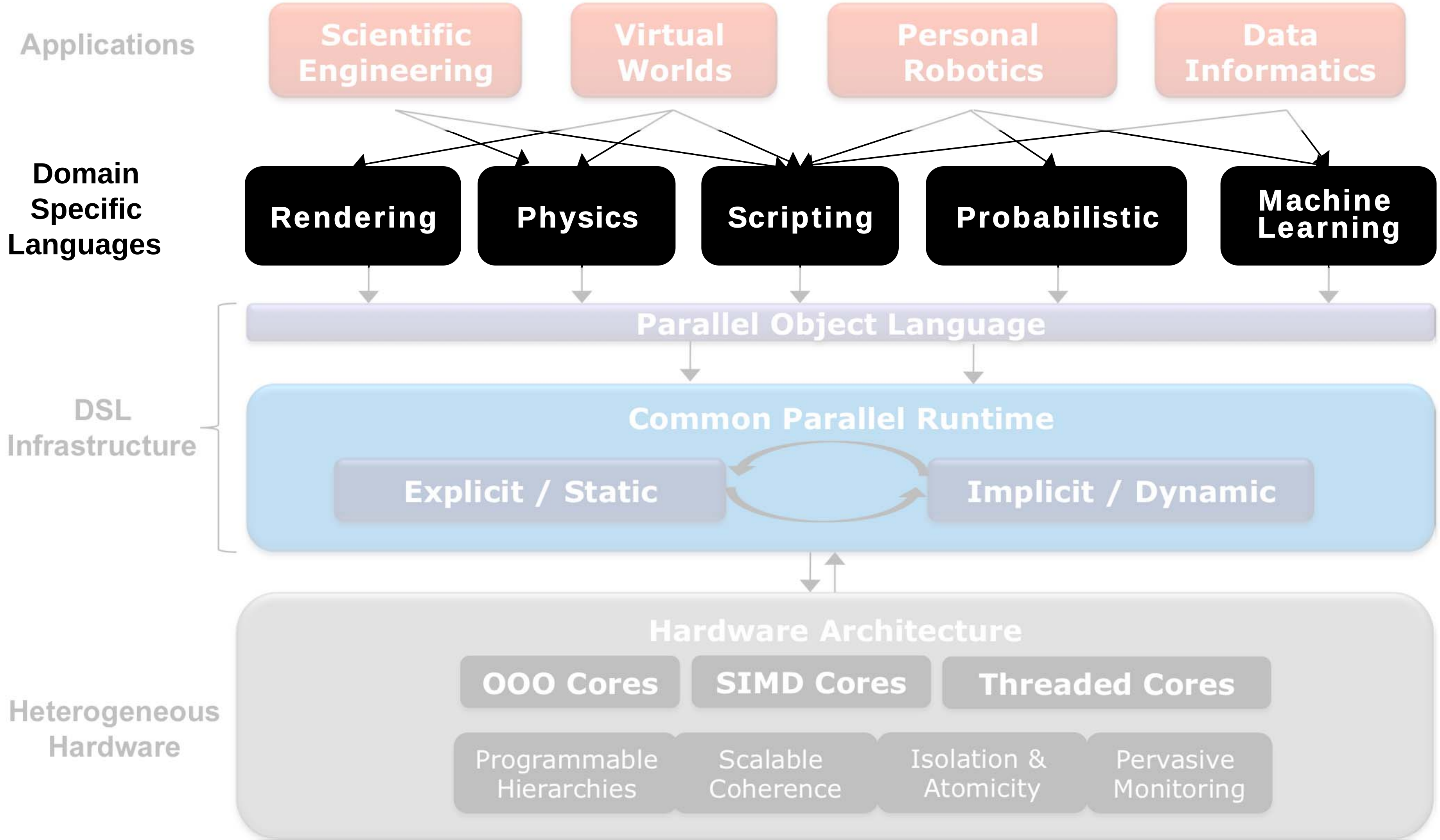
- Leverage domain expertise at Stanford
  - CS research groups, national centers for scientific computing

Virtual Worlds

- Next-generation web platform
  - Millions of players in vast landscapes
  - Immersive collaboration
  - Social gaming
- Computing challenges
  - Client-side game engine
    - Graphics rendering
  - Server-side world simulation
    - Object scripting, geometric queries, AI, physics computation
  - Dynamic content, huge datasets
- More at http://vw.stanford.edu/



Domain Specific Languages

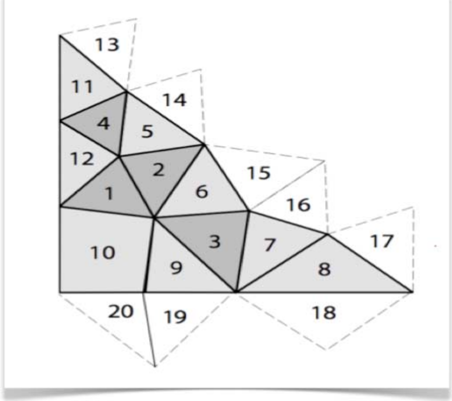


Domain Specific Languages

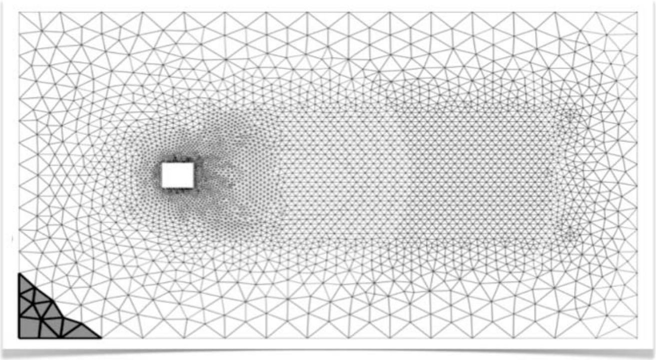
- High-level languages targeted at specific domains
  - E.g.: SQL, Matlab, OpenGL, Ruby/Rails, …
  - Usually declarative and simpler than GP languages
- DSLs ⇒ higher productivity for developers
  - High-level data types & ops (e.g. relations, triangles, …)
  - Express high-level intent w/o implementation artifacts
- DSLs ⇒ scalable parallelism for the system
  - Declarative description of parallelism & locality patterns
    - Can be ported or scaled to available machine
  - Allows for domain specific optimization
    - Automatically adjust structures, mapping, and scheduling

Example DSL: Liszt

- Goal: simplify code of mesh-based PDE solvers
  - Write once, run on any type of parallel machine
  - From multi-cores and GPUs to clusters



- Language features
  - Built-in mesh data types
    - Vertex, edge, face, cell
  - Collections of mesh elements
    - cell.faces(), face.edgesCCW()
  - Mesh-based data storage
    - Fields, sparse matrices
  - Parallelizable iterations
    - Map, reduce, forall statements



Liszt Code Example

```
val position = vertexProperty[double3]("pos")
val A = new SparseMatrix[Vertex,Vertex]

for (c <- mesh.cells) {
    val center = average position of c.vertices
    for (f <- c.faces) {
        val face_dx = average position of f.vertices – center
          for (e <- f.edges With c CounterClockwise) {
               val v0 = e.tail
               val v1 = e.head
               val v0_dx = position(v0) – center
               val v1_dx = position(v1) – center
               val face_normal = v0_dx cross v1_dx
               // calculate flux for face …
               A(v0,v1) += …
               A(v1,v0) -= …
```

Liszt Code Example

```
val position = vertexPr
val A = new SparseMatri

for (c <- mesh.cells) {
    val c      er = average position of c.vertices
    for (f

                                                      nter

              val v0 = e.tail
              val
              val
              val v1_dx =        n(v1) – center
              val face    rmal = v0_dx cross v1_dx
              // calculate flux for face …
              A(v0,v1) += …
              A(v1,v0) -= …
```

High-level data types & operations

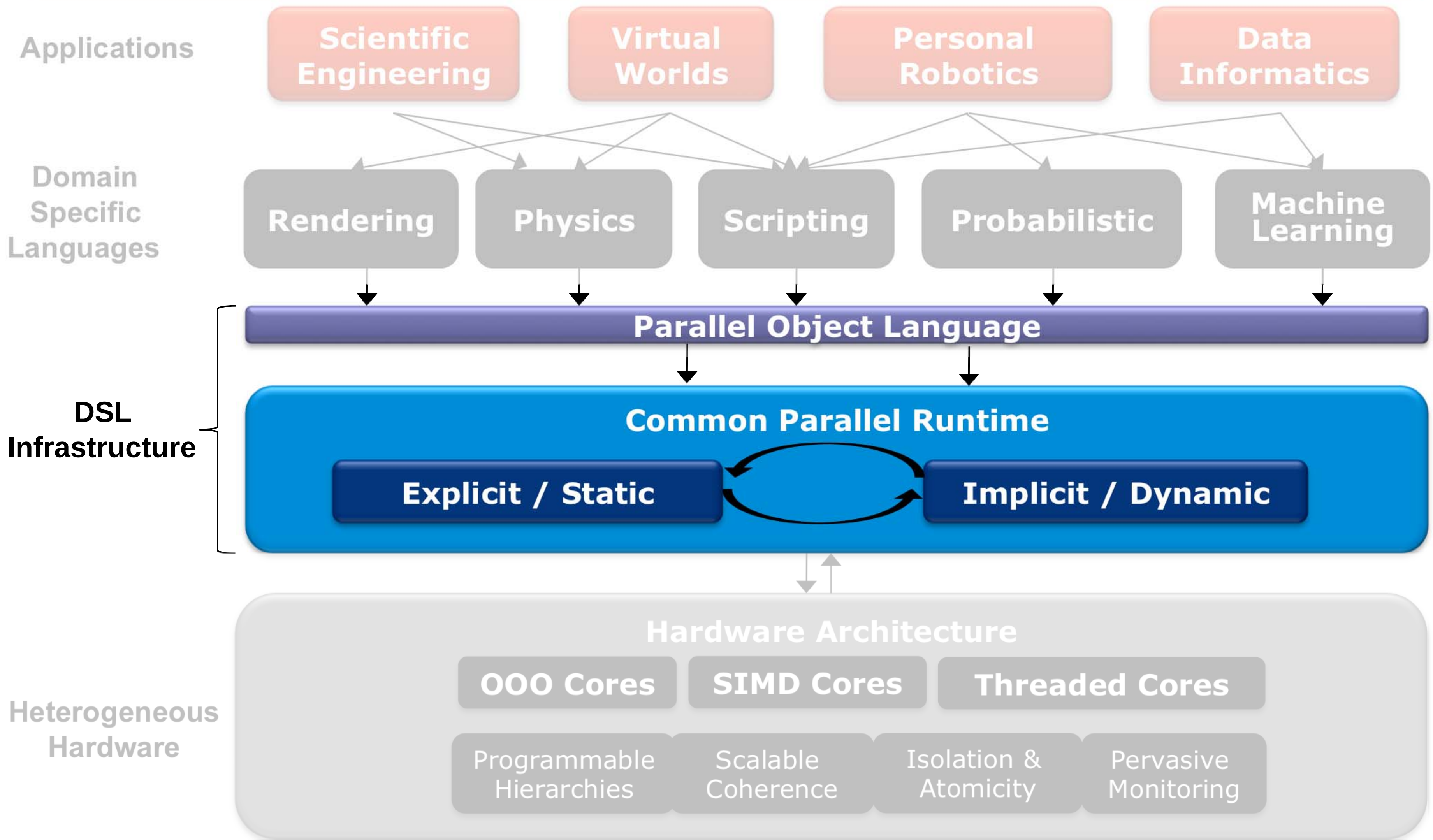
Explicit parallelism using map/reduce/forall
Implicit parallelism with help from DSL & HW

No low-level code to manage parallelism

Liszt Parallelism & Optimizations

- Liszt compiler & runtime manage parallel execution
  - Data layout & access, domain decomposition, communication, …
- Domain specific optimizations
  - Select mesh layout (grid, tetrahedral, unstructured, custom, …)
  - Select decomposition that improves locality of access
  - Optimize communication strategy across iterations
- Optimizations are possible because
  - Mesh semantics are visible to compiler & runtime
  - Iterative programs with data accesses based on mesh topology
  - Mesh topology is known to runtime

DSL Infrastructure



DSL Infrastructure Goals

- Provide a shared framework for DSL development

- Features
  - Common parallel language that retains DSL semantics
  - Mechanism to express domain specific optimizations
  - Static compilation + dynamic management environment
    - For regular and unpredictable patterns respectively
  - Synthesize HW features into high-level solutions
    - E.g. from HW messaging to fast runtime for fine-grain tasks
  - Exploit heterogeneous hardware to improve efficiency

Parallel Object Language

- Required features
  - Support for functional programming (FP)
    - Declarative programming style for portable parallelism
    - High-order functions allow parallel control structures
  - Support for object-oriented programming (OOP)
    - Familiar model for complex programs
    - Allows mutable data-structures and domain-specific attributes
  - Managed execution environment
    - For runtime optimizations & automated memory management
- Our approach: embed DSLs in the **Scala** language
  - Supports both FP and OOP features
  - Supports embedding of higher-level abstractions
  - Compiles to Java bytecode

DSL Execution with the Delite Parallel Runtime

Application

```
def example(a: Matrix[Int],
            b: Matrix[Int],
            c: Matrix[Int],
            d: Matrix[Int]) =
{

  val ab = a * b
  val cd = c * d
  return ab + cd

}
```

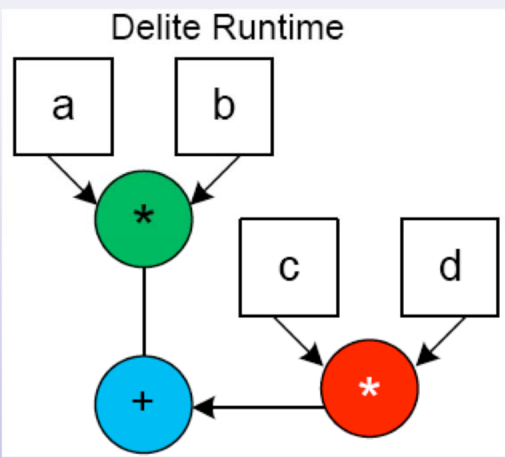
Calls Matrix DSL methods

Matrix DSL

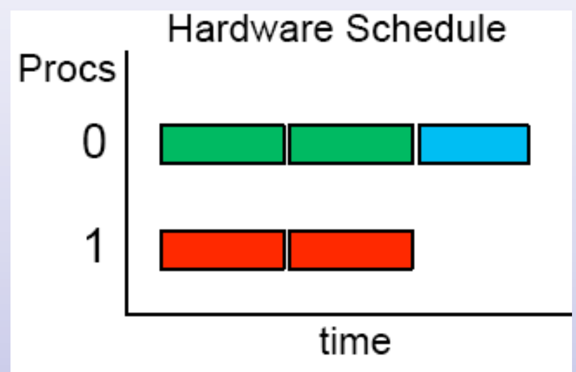
```
def *(m: Matrix[Int]) =
  delite.defer(OP_mult(this, m))

def +(m: Matrix[Int]) =
  delite.defer(OP_plus(this, m))
```

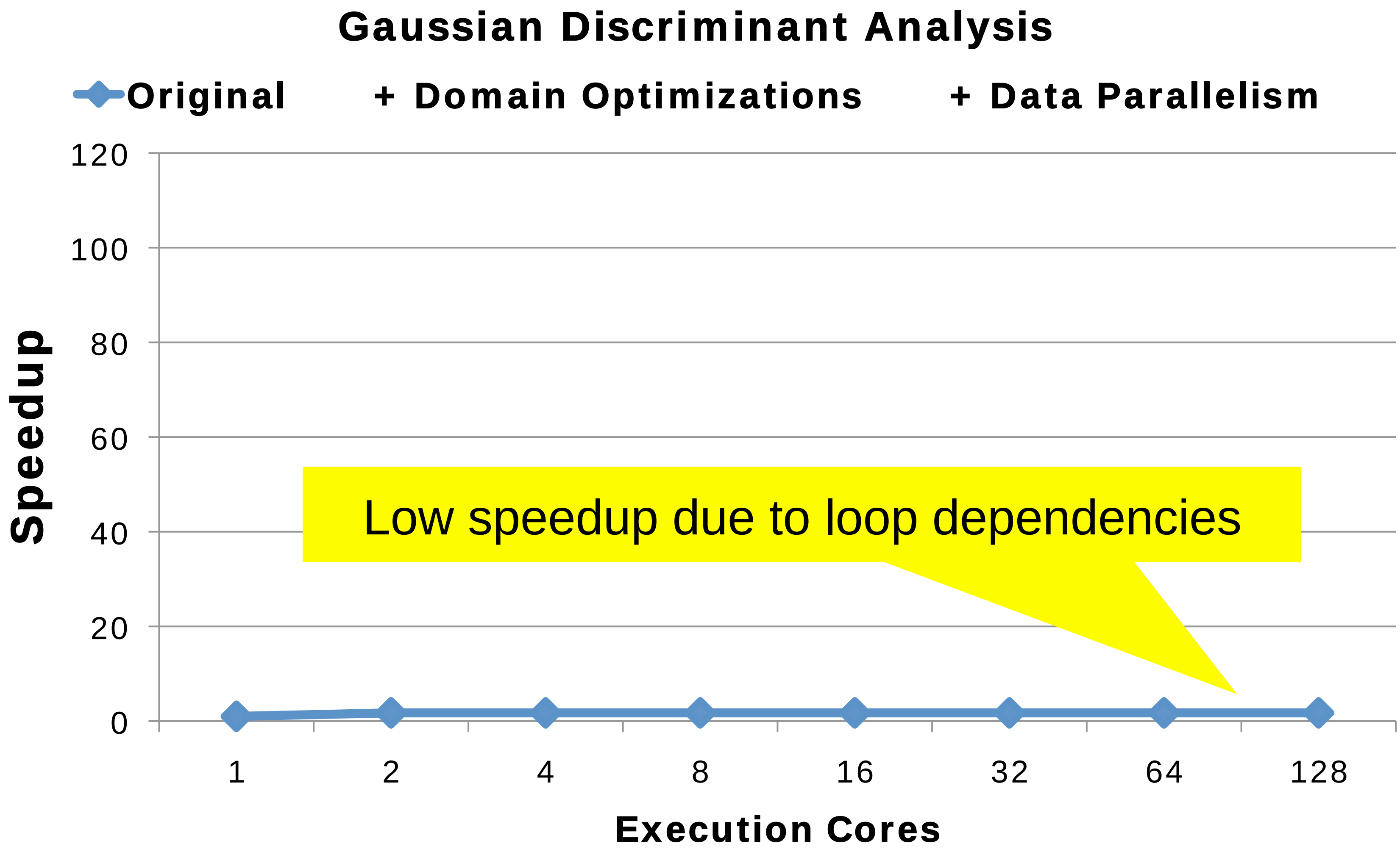
DSL defers OP execution to Delite



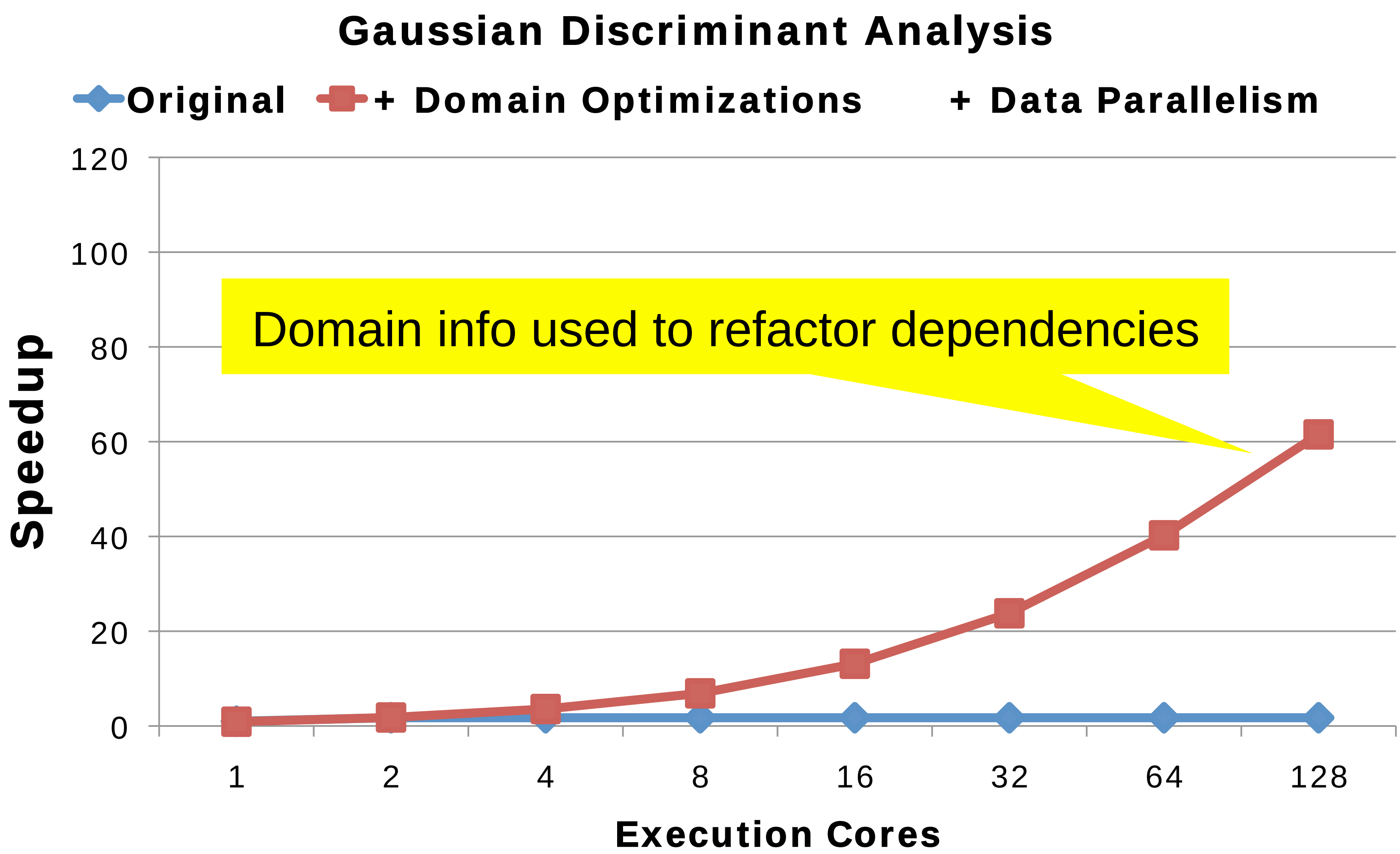
Delite applies generic & domain transformations to generate mapping



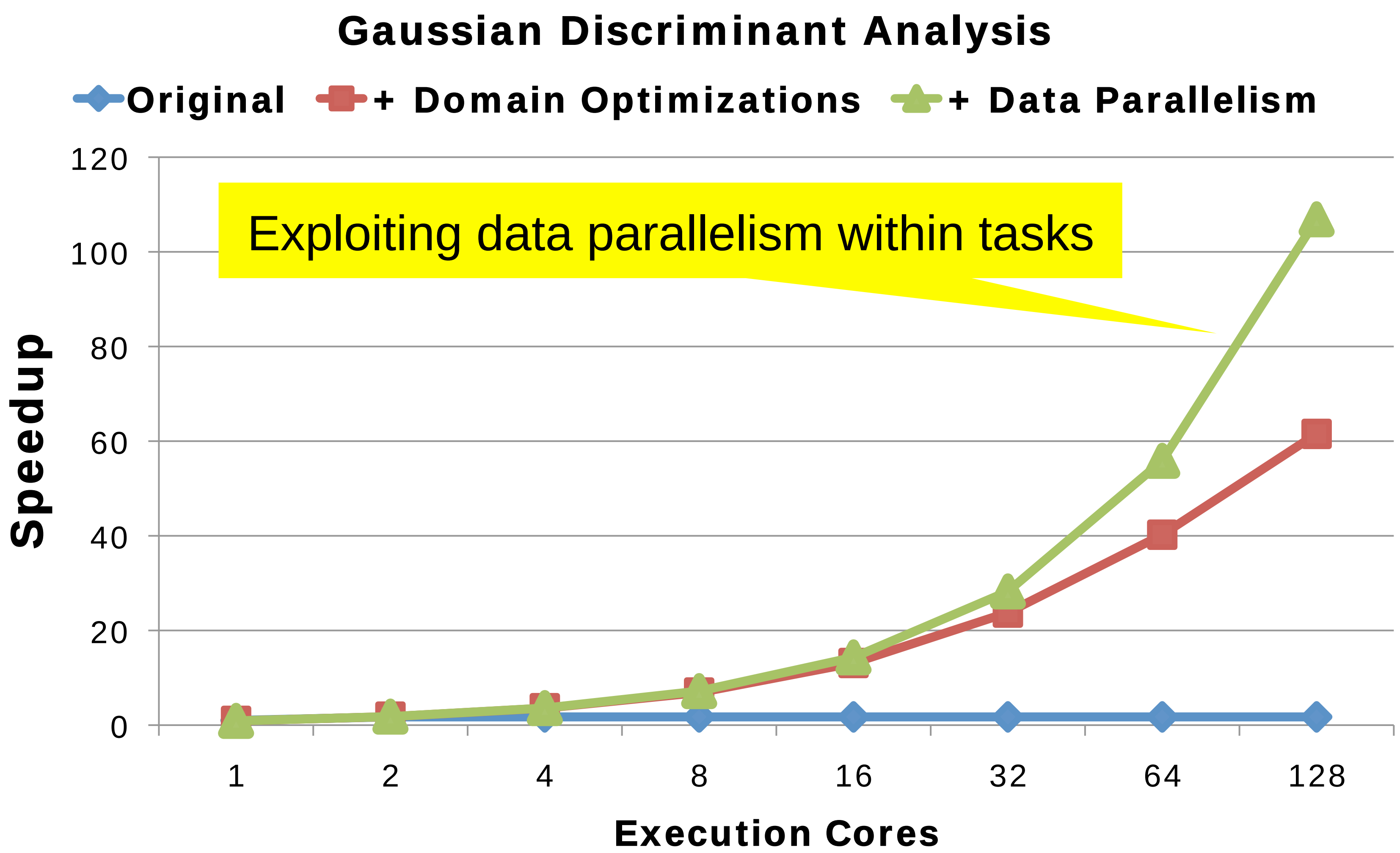
Performance with Delite



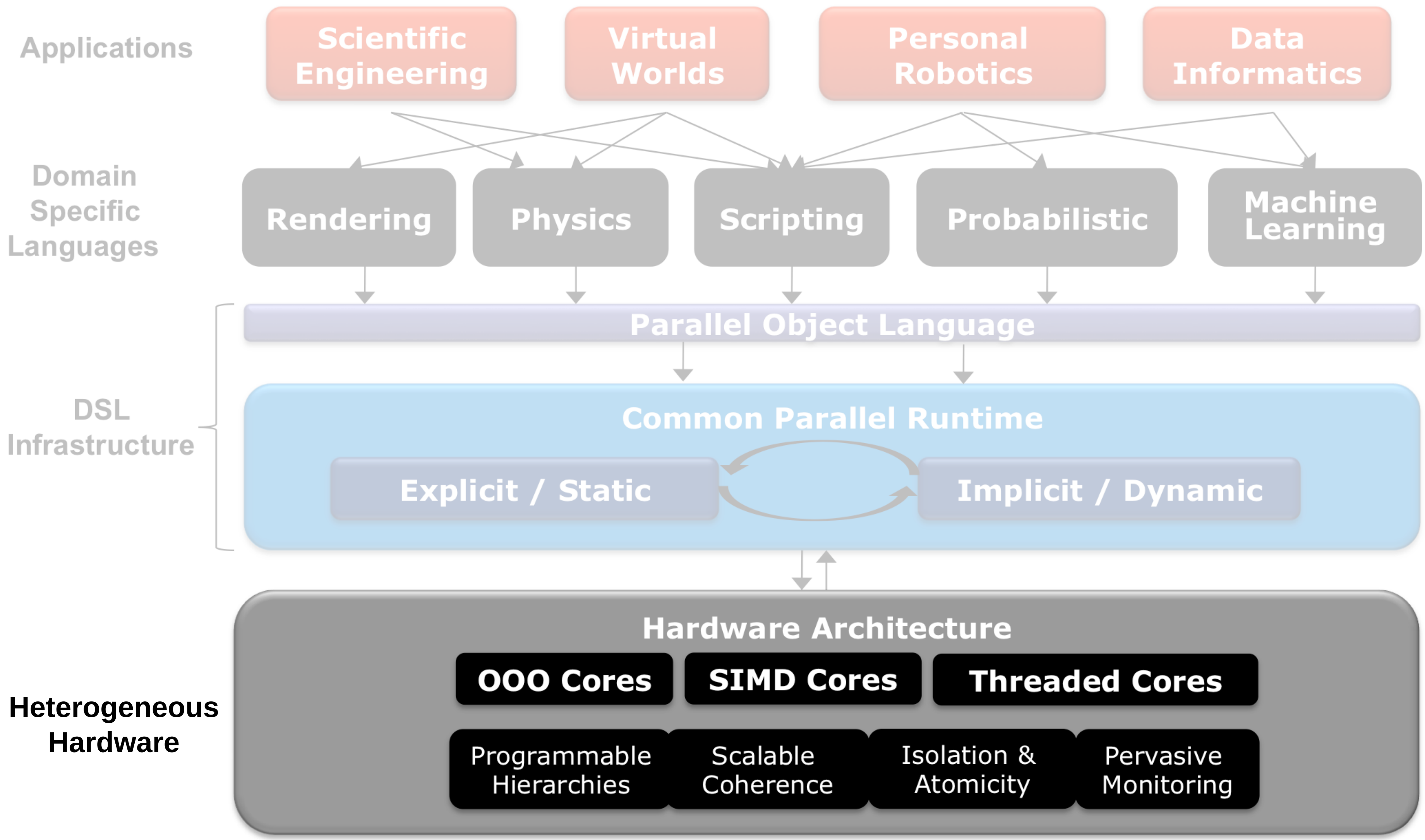
Performance with Delite



Performance with Delite



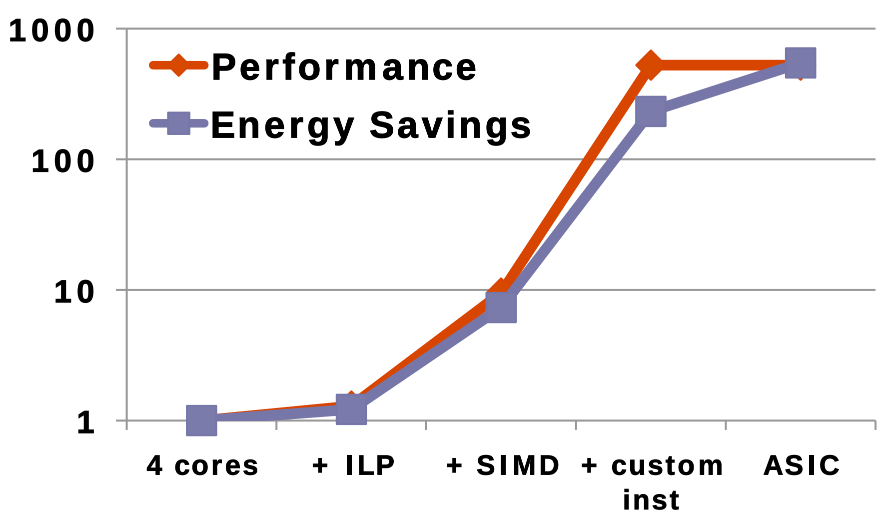
Hardware Architecture

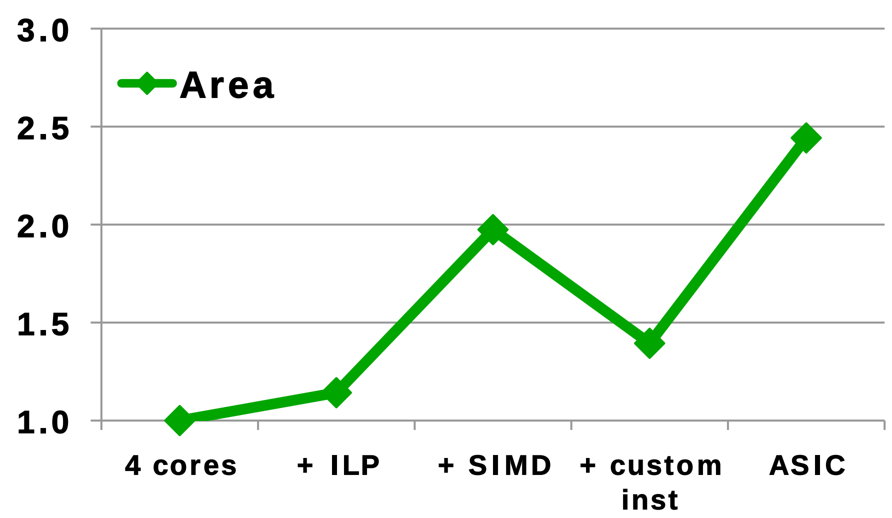


Heterogeneous Hardware

- Heterogeneous HW for energy & area efficiency
  - ILP, threads, data-parallel engines, custom engines
  - Q: what is the right balance of features in a chip?
  - Q: what tools can generate best chip for an app/domain?

- Study: HW options for H.264 encoding



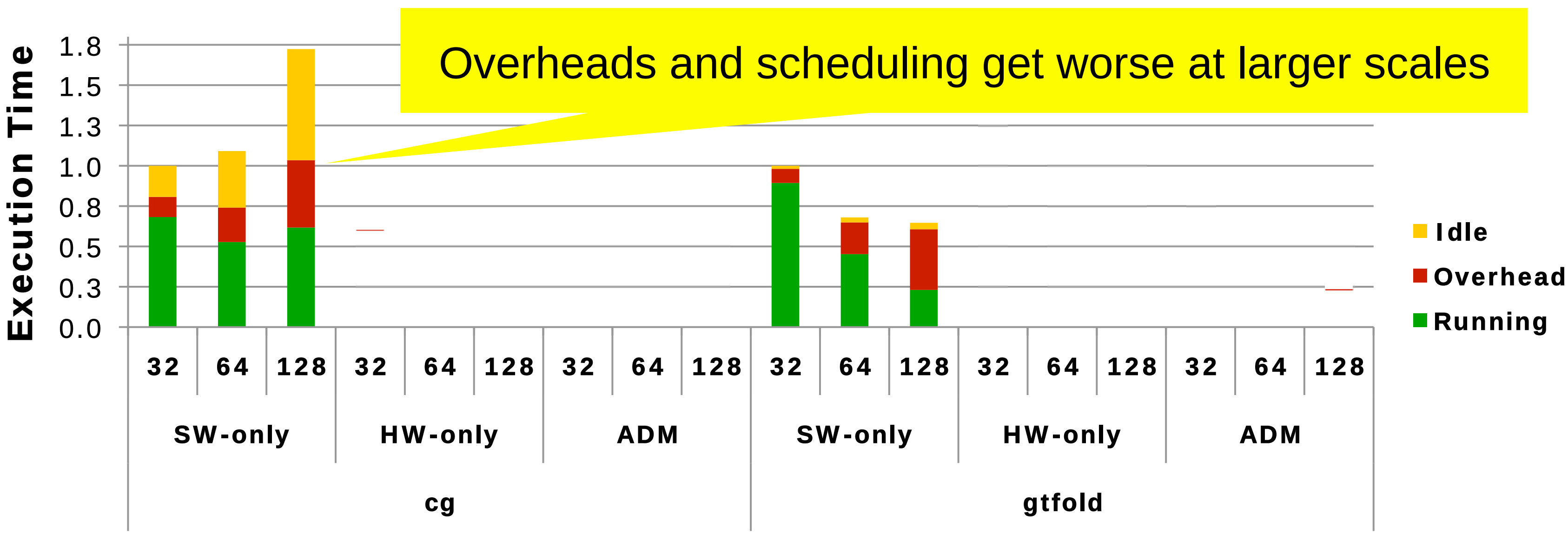


Architectural Support for Parallelism

- Revisit architectural support for parallelism
  - Which are the basic HW primitives needed?
  - Challenges: semantics, implementation, scalability, virtualization, interactions, granularity (fine-grain & bulk), …
- HW primitives
  - Coherence & consistency, atomicity & isolation, memory partitioning, data and control messaging, event monitoring
- Runtime synthesizes primitives into SW solutions
  - Streaming system: mem partitioning + bulk data messaging
  - TLS: isolation + fine-grain control communication
  - Transactional memory: atomicity + isolation + consistency
  - Security: mem partitioning + isolation
  - Fault tolerance: isolation + checkpoint + bulk data messaging

Example: HW Support for Fine-grain Parallelism

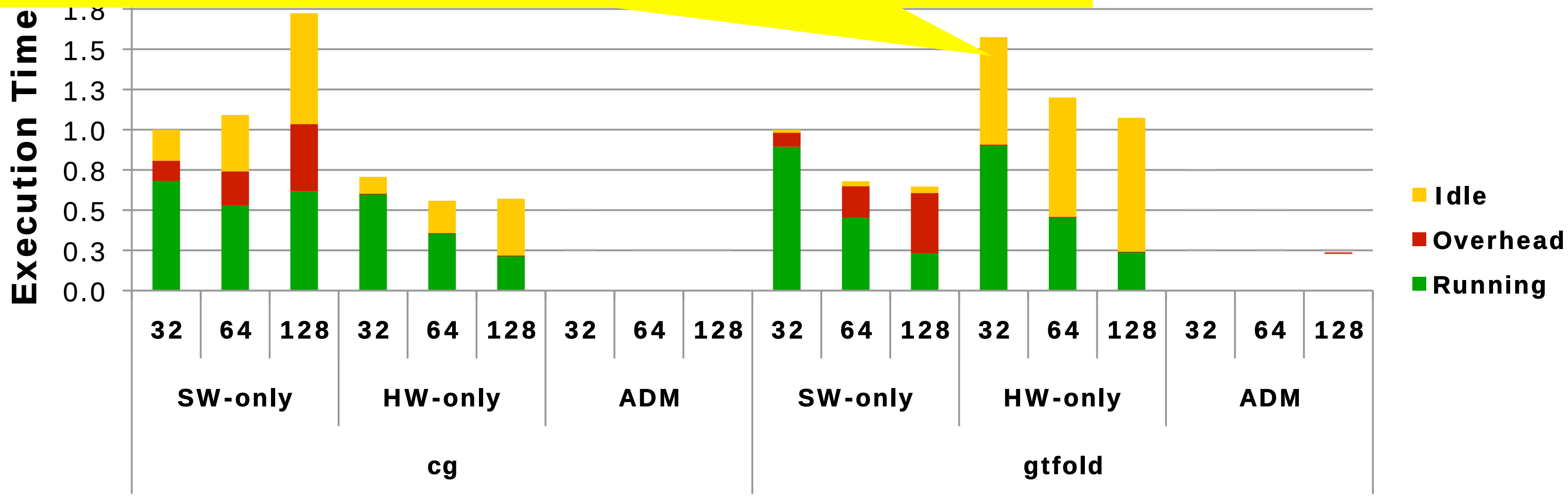
- Parallel tasks with a few thousand instructions
  - Critical to exploit in large-scale chips
  - Tradeoff: load balance vs overheads vs locality
- Software-only scheduling
  - Per-thread task queues + task stealing
  - Flexible algorithms but high stealing overheads



Example: HW Support for Fine-grain Parallelism

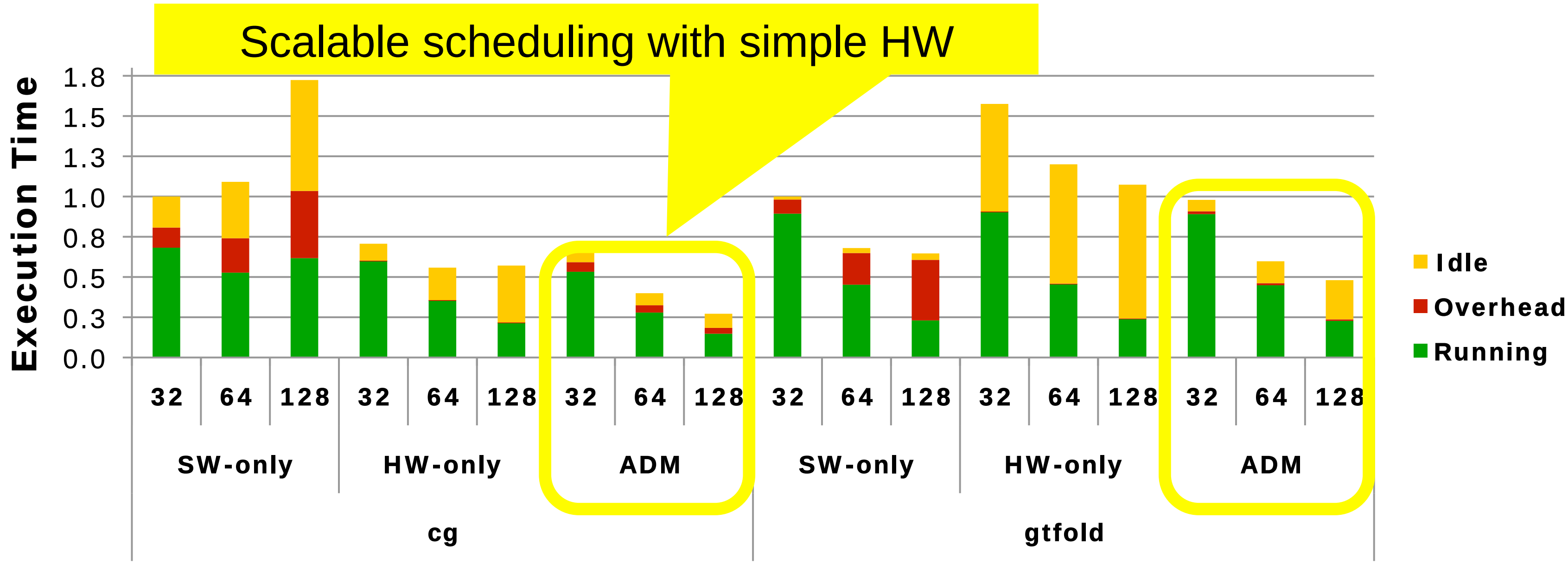
- Hardware-only scheduling
  - HW tasks queues + HW stealing protocol
  - Minimal overheads (bypass coherence protocol)
  - But fixes scheduling algorithm
    - Optimal approach varies across applications
    - Impractical to support all options in HW

Wrong scheduling algorithm makes HW slower than SW



Example: HW Support for Fine-grain Parallelism

- Simple HW feature: asynchronous direct messages
  - Register-to-register, received with user-level interrupt
  - Fast messaging for SW schedulers with flexible algorithms
    - E.g., gtfold scheduler tracks domain-specific dependencies
    - Also useful for fast barriers, reductions, IPC, …
  - Better performance, simpler HW, more flexibility & uses



PPL Summary

- Goal: make parallel computing practical for the masses
- Technical approach
  - **Domain specific languages (DSLs)**
    - Simple & portable programs
  - **Heterogeneous hardware**
    - Energy and area efficient computing
  - Working on the SW & HW techniques that bridge them
- More info at: **http://ppl.stanford.edu**