中山大學
SUN YAT-SEN UNIVERSITY

高性能任务级运行时平台
Legion软件生态

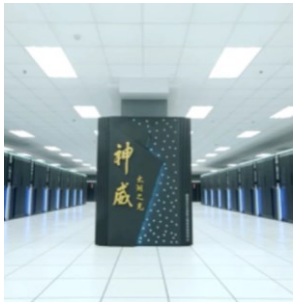
汇报人：肖霖畅

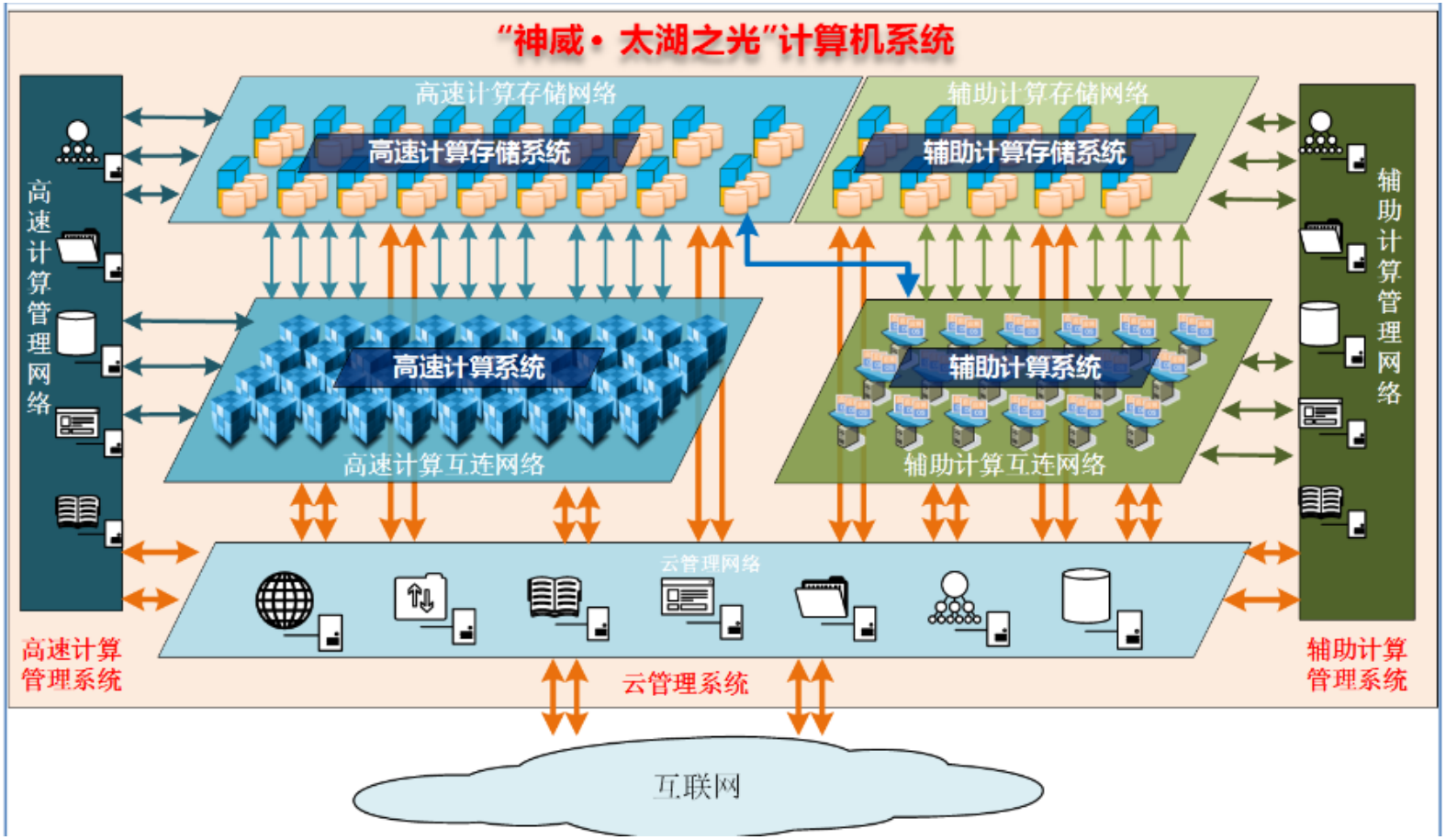
大纲

1. **背景**
2. Legion软件栈层级
3. Legion软件栈详细介绍
4. 总结、未来研究方向和现有问题

背景

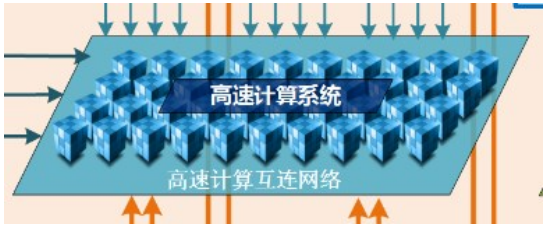
- 现代超级计算机特征
  - 异构性：处理器种类异构、处理器性能异构
  - 分布式内存：存储大小和速度的不一致性





背景

- 复杂的分布式存储层级 —— 神威太湖之光高速计算系统处理器



DDR3 存储控制器

从核存储系统：
1. 一级Cache: 16KB
2. 二级Cache: 核组共享64KB

主核存储系统：
1. 一级Cache: 32KB
2. 二级Cache: 512KB

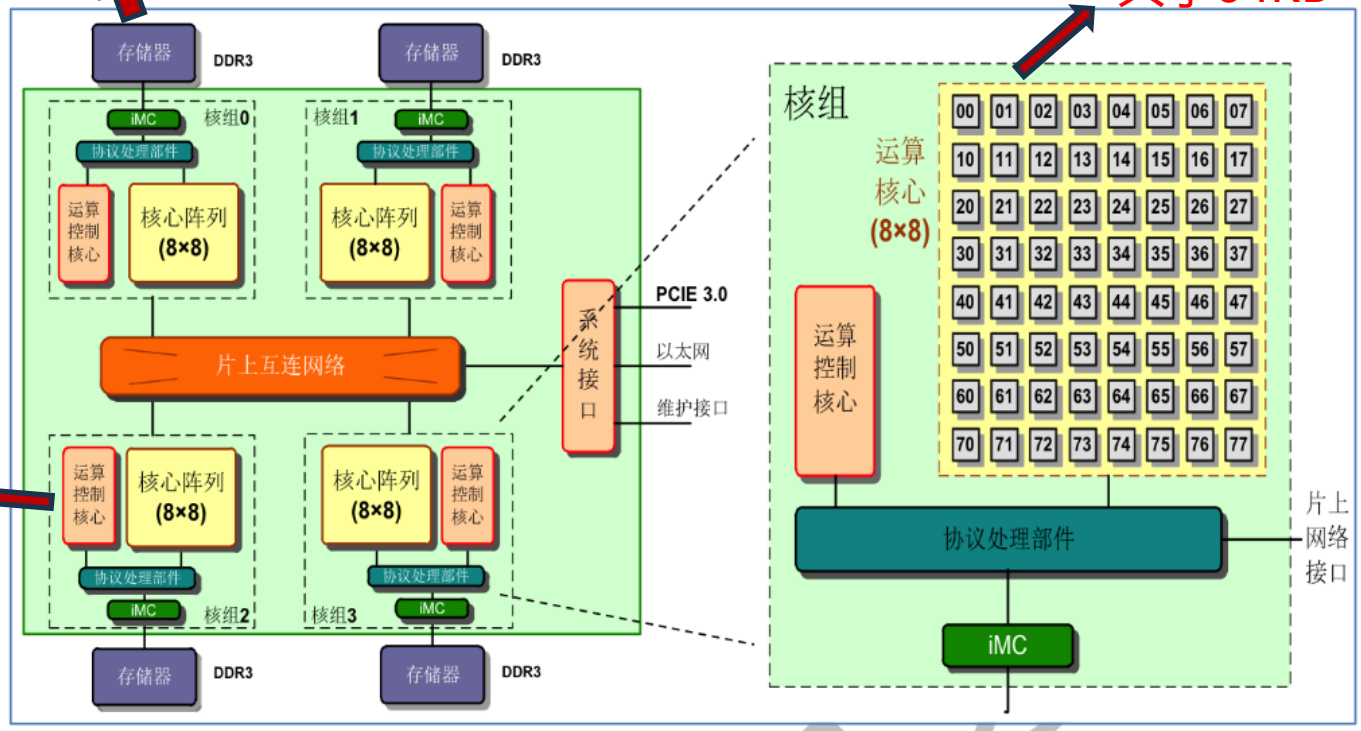
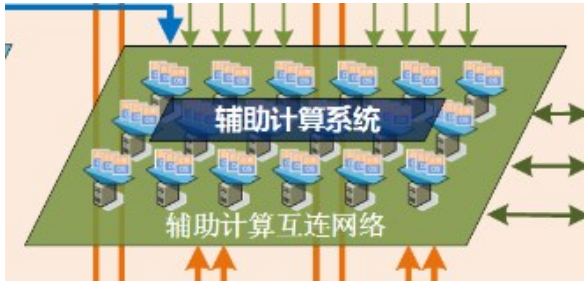


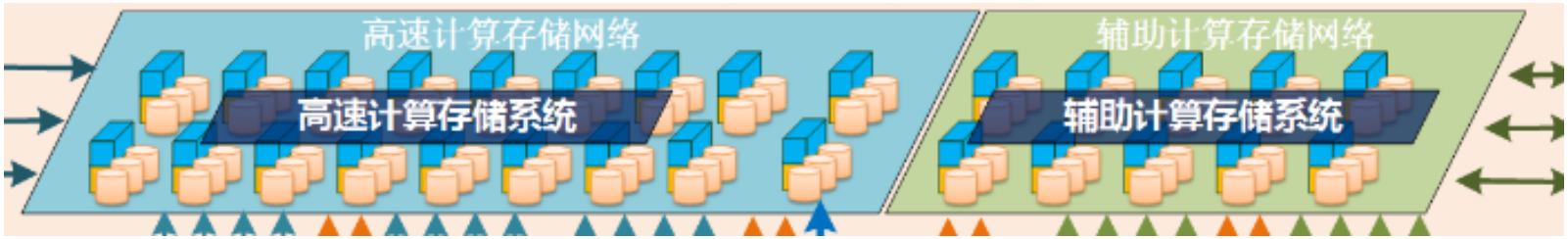
图 1-2“申威 26010”异构众核处理器架构图

背景

- 复杂的分布式存储层级 —— 神威太湖之光辅助计算系统计算节点
  - **普通计算节点**：每节点包含DDR4 内存(128GB)
  - **大内存计算节点**：每节点包含DDR4 内存(1TB)、SAS 本地存储(7TB)、SSD 本地存储(500GB)
  - **GPU 计算节点**：DDR4 内存(128GB)、7KRPM 本地存储(1TB)、SSD 本地存储(1.6TB)



- 复杂的分布式存储层级 —— 神威太湖之光计算存储系统



背景

- 现代计算机架构的成本
  - 现代计算机架构越来越多地由异构处理器和复杂的内存层次结构组成，这些架构中的数据移动成本现在开始主导计算的总体成本[1]。
  - 低效的数据移动模式会导致并行计算的效率受到很大的影响：数据移动的操作即通信在整个训练过程的所有操作中所占据的比例往往是较高的。
  - 异构计算核心和内存层级导致计算工作流以及对数据移动与访问的控制更加复杂，并行编程的正确性变得困难。

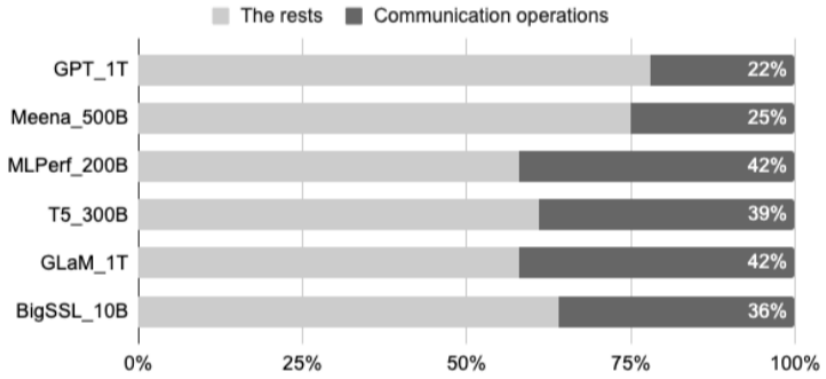
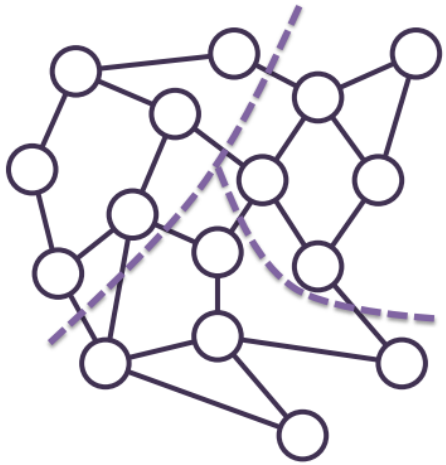


Figure 1: Training step time breakdown of large models. The models run on 128 - 2048 TPU chips depending on the model size. Details on the large models can be found in Section 6.



[1] Bauer M, Treichler S, Slaughter E, et al. Legion: Expressing locality and independence with logical regions[C] //SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. IEEE, 2012: 1-11.

背景

- Legion 应运而生!
  - 项目启动时间：2011年
  - 扩展到Titan超算：2014年
  - 主要参与单位：
    - Stanford University
    - Nvidia
    - Los Alamos National Laboratory
    - SLAC National Accelerator Laboratory

背景

- Legion及其软件栈
  - Legion：基于Task的编程系统
    - **Task是可以异步运行的函数**（难点：Task之间会存在依赖）
    - **Task可以被迁移到分布式系统的任何位置执行**（难点：分布式计算设备异构、数据的存放位置和权限、放到哪里执行？）
  - Legion设计原则：Data, not compute, matters most.
    - Task的特性导致数据处理存在同步问题
    - 数据以Region的形式进行管理，通过Logical Region和Physical Region的设计来管理数据的存放
  - Legion：设计了一套良好的编程抽象和应用接口
    - 支持大量上层任务基于Legion进行超算适配

背景

- Legion成就[2]
  - 形成复杂的、功能完备的软件栈
  - 大量系统顶会论文
  - R&D 100

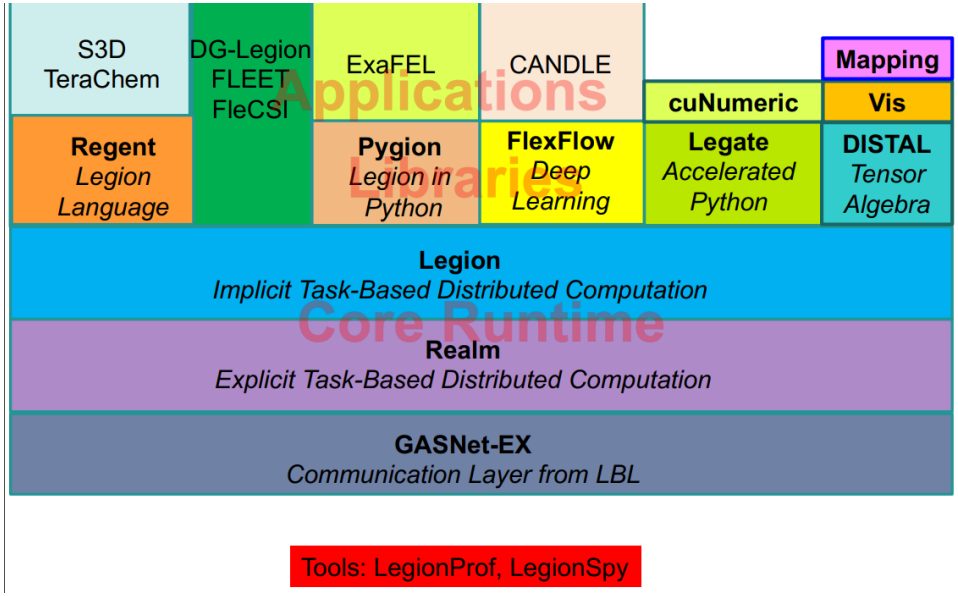


Table of Contents

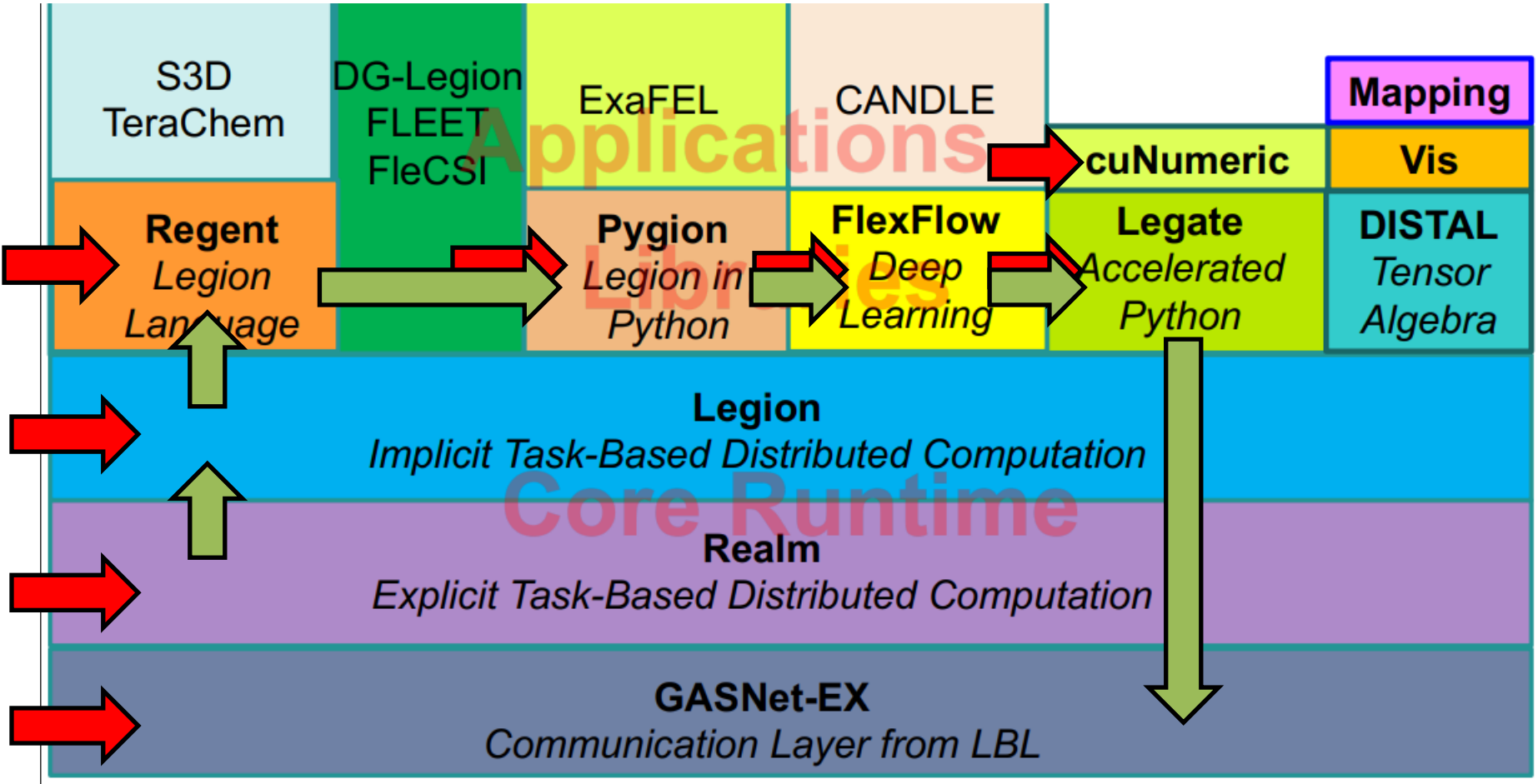
- Legion Runtime:
  - Overview (SC 2012) [PDF]
  - Structure Slicing (SC 2014) [PDF]
  - Tracing (SC 2018) [PDF]
  - Correctness of Dependence Analysis (Correctness 2018) [PDF]
  - Dynamic Control Replication (PPoPP 2021) [PDF]
  - Index Launches (SC 2021) [PDF]
  - Visibility Algorithms (PPoPP 2023) [PDF]
- Programming Model:
  - Partitioning Type System (OOPSLA 2013) [PDF]
  - Dependent Partitioning (OOPSLA 2016) [PDF]
- Realm:
  - Overview (PACT 2014) [PDF]
  - I/O Subsystem (HiPC 2017) [PDF]
- Regent:
  - Overview (SC 2015) [PDF]
  - Control Replication (SC 2017) [PDF]
  - Auto-Parallelizer (SC 2019) [PDF]
- Bindings:
  - Python (PAW-ATM 2019) [PDF]
- Libraries and Techniques:
  - Visualization (ISAV 2017) [PDF]
  - Graph Processesing (VLDB 2018) [PDF, Software Release]
  - Legate NumPy (SC 2019) [PDF]
  - Tensor Algebra (PLDI 2022) [PDF]
  - Distributed Task Fusion (PAW-ATM 2022) [PDF]
  - Sparse Tensor Algebra (SC 2022) [PDF]
  - Legate Sparse (SC 2023) [PDF]
  - AutoMap (SC 2023) [PDF]
- Applications:
  - S3D-Legion (2017) [PDF]
  - Soleil-X (2018) [PDF]
  - HTR Solver (2020) [PDF]
  - Task Bench (SC 2020) [PDF]
  - Meshfree Solver (PAW-ATM 2020) [PDF]
- DSLs:
  - Singe (PPoPP 2014) [PDF]
  - Scout (WOLFHPC 2014) [PDF]
- Theses:
  - Michael Bauer's Thesis (2014) [PDF]
  - Sean Treichler's Thesis (2016) [PDF]
  - Elliott Slaughter's Thesis (2017) [PDF]
  - Wonchan Lee's Thesis (2019) [PDF]
  - Rupanshu Soi's Thesis (2021) [PDF]

[2] Legion Overview – Legion Programming System (stanford.edu)

大纲

1. 背景
2. **Legion软件栈层级**
3. Legion软件栈详细介绍
4. 总结、未来研究方向和现有问题

Legion软件栈层级



Tools: LegionProf, LegionSpy

Legion软件栈层级 – 本次涉及内容

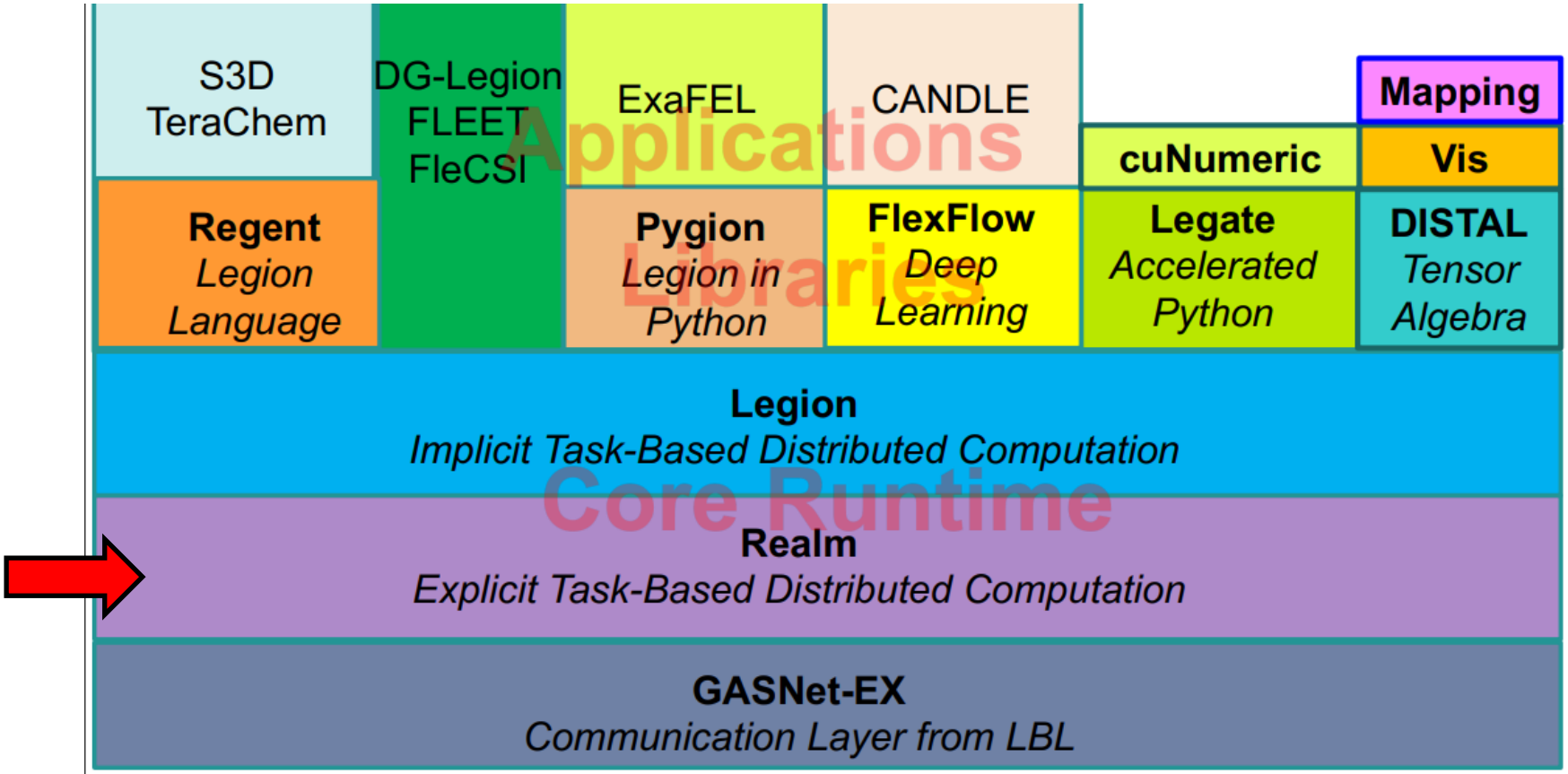
- Commuication Layer级
  - **MPI & GASNet-EX**: Communication Layer
- Core Runtime级
  - **Realm**: Explicit Task-Based Distributed Computation 显式任务分布式计算
  - **Legion**: Implicit Task-Based Distributed Computation 隐式任务分布式计算
- Libraries级
  - **Regent**: 易用的静态类型领域语言
  - **Pygion**: Python动态类型语言支持
  - **Flexflow**: 自动深度学习任务并行优化底层支持库
  - **Legate**: Numpy, Pandas, Scipy等任务并行优化的底层支持库
- Applications级
  - **cuNumeric**: 基于Legion的Numpy实现

大纲

1. 背景
2. Legion软件栈层级
3. **Legion软件栈详细介绍**
4. 总结、未来研究方向和现有问题

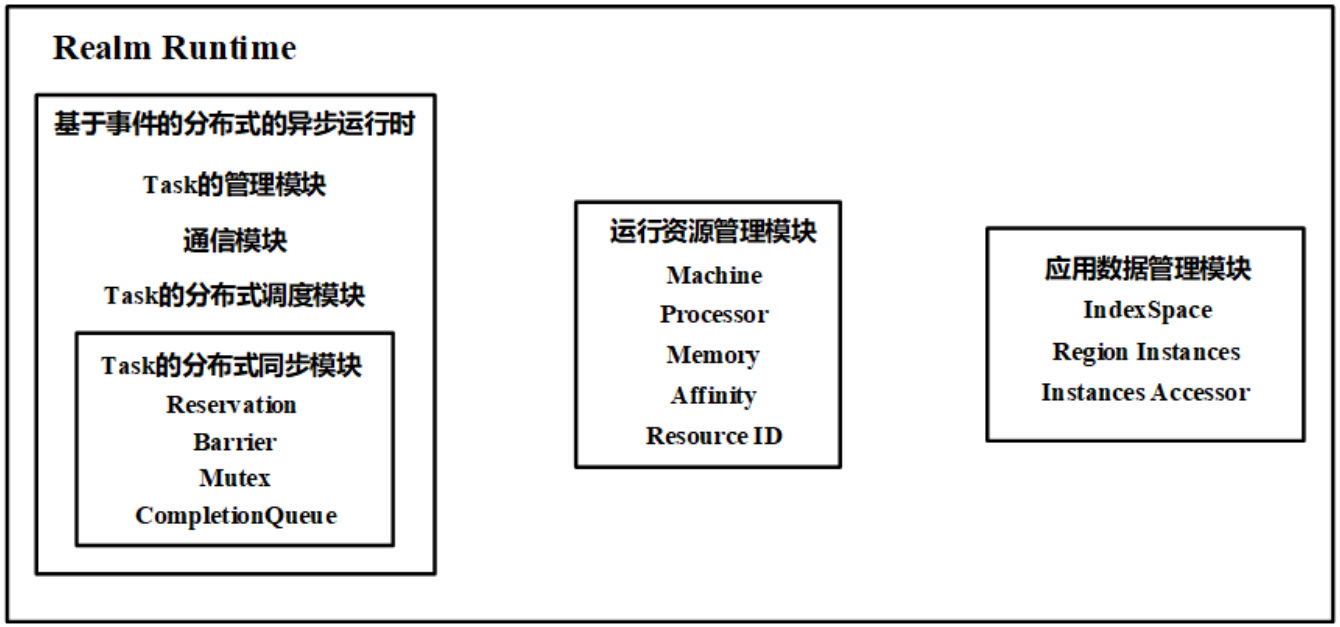
Legion软件栈详细介绍 – Realm

- Legion的核心运行时
  - Realm：显式任务分布式计算运行时系统
    - 2020年起从Legion项目中解耦，成为独立维护的底层模块



Legion软件栈详细介绍 - Realm

- Realm[1]: Explicit Task-Based Distributed Computation
  - **基于事件的的分布式异步运行时**：Task管理、同步、通信、分布式调度
  - **运行资源管理模块**：管理异构资源的统一抽象和接口
  - **应用数据管理模块**：统一管理异构设备内存中数据的创建、访问、销毁和操作(划分、规约等)



[1] Treichler, Sean, Michael Bauer, and Alex Aiken. "Realm: An event-based low-level runtime for distributed memory architectures." Proceedings of the 23rd international conference on Parallel architectures and compilation. 2014.

Legion软件栈详细介绍 - Realm

- Realm运行资源管理模块 - Machine model
  - Realm使用Machine model的抽象来统一地表示与管理所有Task运行时相关的资源，包括计算资源、存储资源等。包含下列几个重要的概念:
    - **Machine**: 最高级别的抽象，表示应用可以使用的所有计算节点，包含所有其中的processor实例、memory实例、affinity信息
    - **Processor**: 表示可以用于运行task的执行资源，比如CPU核心、GPU核心、OpenMP线程池等
    - **Memory**: 描述存放应用数据的位置
    - **Resource ID**: Realm资源实例的唯一标识符（64位id，表示实例类型、所处节点和资源在本地的索引）
    - **Affinity**: 描述Memory实例之间，以及与Processor实例之间的亲和性（类似K8S）

Legion软件栈详细介绍 - Realm

运行资源管理模块
Machine
Processor
Memory
Affinity
Resource ID

- Realm运行资源管理模块 - Machine model
  - Realm Memory 支持的丰富存储类型

```
// Different Memory types
#define REALM_MEMORY_KINDS(__op__) \
  __op__(NO_MEMKIND, "") \
  __op__(GLOBAL_MEM, "Guaranteed visible to all processors on all nodes (e.g. GASNet memory,
  universally slow)") \
  __op__(SYSTEM_MEM, "Visible to all processors on a node") \
  __op__(REGDMA_MEM, "Registered memory visible to all processors on a node, can be a target
  of RDMA") \
  __op__(SOCKET_MEM, "Memory visible to all processors within a node, better performance to
  processors on same socket") \
  __op__(Z_COPY_MEM, "Zero-Copy memory visible to all CPUs within a node and one or more
  GPUs") \
  __op__(GPU_FB_MEM, "Framebuffer memory for one GPU and all its SMs") \
  __op__(DISK_MEM, "Disk memory visible to all processors on a node") \
  __op__(HDF_MEM, "HDF memory visible to all processors on a node") \
  __op__(FILE_MEM, "file memory visible to all processors on a node") \
  __op__(LEVEL3_CACHE, "CPU L3 Visible to all processors on the node, better performance to
  processors on same socket") \
  __op__(LEVEL2_CACHE, "CPU L2 Visible to all processors on the node, better performance to
  one processor") \
  __op__(LEVEL1_CACHE, "CPU L1 Visible to all processors on the node, better performance to
  one processor") \
  __op__(GPU_MANAGED_MEM, "Managed memory that can be cached by either host or GPU") \
  __op__(GPU_DYNAMIC_MEM, "Dynamically-allocated framebuffer memory for one GPU and all its
  SMs")
```

Legion软件栈详细介绍 - Realm

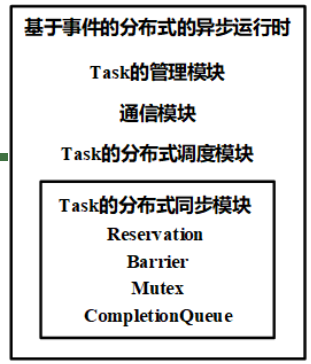
运行资源管理模块
Machine
Processor
Memory
Affinity
Resource ID

- Realm运行资源管理模块 - Machine model
  - Realm Affinity：提供Process和Memory、Memory和Memory的亲和性信息，包括时延和带宽

```
auto machine = Machine::get_machine();
auto p = Machine::ProcessorQuery(machine)
             .only_kind(Processor::LOC_PROC)
             .first();
auto m =
    Machine::MemoryQuery(machine).only_kind(Memory::SYSTEM_MEM).first();

std::vector<Machine::ProcessorMemoryAffinity> pm_affinity;
machine.get_proc_mem_affinity(pm_affinity, p, m, true /*local_only*/);
unsigned bandwidth = pm_affinity[0].bandwidth;
unsigned latency = pm_affinity[0].latency;
log_app.print("bandwidth: %u MB/s, latency: %u ns", bandwidth, latency);
```

```
[0 - 7f4e47328440]    0.001258 {3}{app}: bandwidth: 100 MB/s, latency: 5 ns
```

Legion软件栈详细介绍 - Realm



- Realm 基于事件的的分布式异步运行时
  - 一个完全异步的、基于Event的运行时，是Realm编程模型的核心
    - **Future的编程模型**：所有的Realm操作都由Runtime在后台延迟与非阻塞地运行，用户利用Runtime返回的Event实例来判断操作是否确实完成了
    - 通常用户可以在Event上阻塞地等待，或者将该Event作为其他操作的前置或者后置条件来使用，从而描述不同异步操作之间的依赖关系

```
Runtime rt;
rt.init(&argc, &argv);

// 注册任务
auto print_str = std::string("hello,realm!");
auto register_event = Processor::register_task_by_kind(
    Processor::LOC_PROC, false /*!global*/, MY_TASK,
    CodeDescriptor(my_task), ProfilingRequestSet(), &print_str,
    sizeof(std::string *));
// 显式地在事件上调用wait从而阻塞地等待注册完成
register_event.wait();
```

```
// lanuch任务
auto p = Machine::ProcessorQuery(Machine::get_machine())
                .only_kind(Processor::LOC_PROC)
                .first();
auto task_event = p.spawn(MY_TASK, nullptr, 0);

// 等待task event触发即task执行完毕时，再停止runtime的运行
rt.shutdown(task_event);
// 阻塞地等待runtime确实停止完毕
rt.wait_for_shutdown();
```

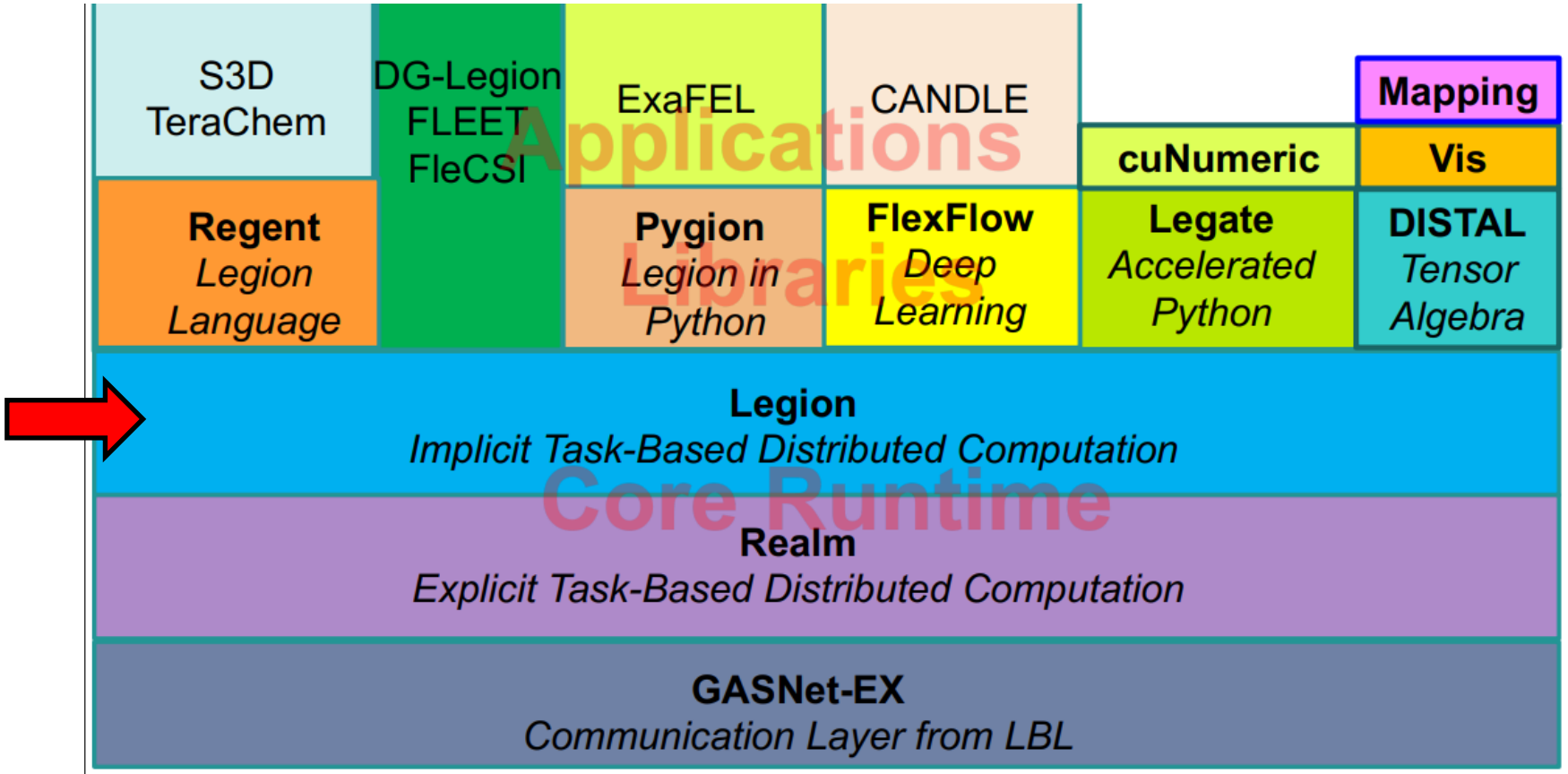
Legion软件栈详细介绍 - Realm

- Realm 应用数据管理模块
  - **核心抽象：Task**
    - 一个拥有特定函数签名与任意函数体的函数，表示用户自定义的异步操作
    - task在运行前也需要注册到Realm的运行时当中
    - 注册了任务后，可以在任意的processor上执行任务
  - **核心抽象：RegionInstance**，用于统一管理存储和应用的数据
  - **核心抽象：Instances Accessor**，用于控制数据的访问权限
  - **核心抽象：IndexSpace**，可以实现对RegionInstance进行拷贝、填充、规约等操作
- 复杂的权限管理，有兴趣同学可以观看文档[3]

[2] Realm Index Spaces – Legion Programming System, Realm Copies and Fills – Legion Programming System, Realm Reductions – Legion Programming System

Legion软件栈详细介绍 – Legion

- Legion的核心运行时
  - Legion[1]：隐式任务分布式计算运行时系统



[1] Bauer, Michael, et al. "Legion: Expressing locality and independence with logical regions." SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. IEEE, 2012.

Legion软件栈详细介绍 - Legion

- Legion: Implicit Task-Based Distributed Computation
  - 基于Realm的提供更多功能的Runtime库

Realm的模块:

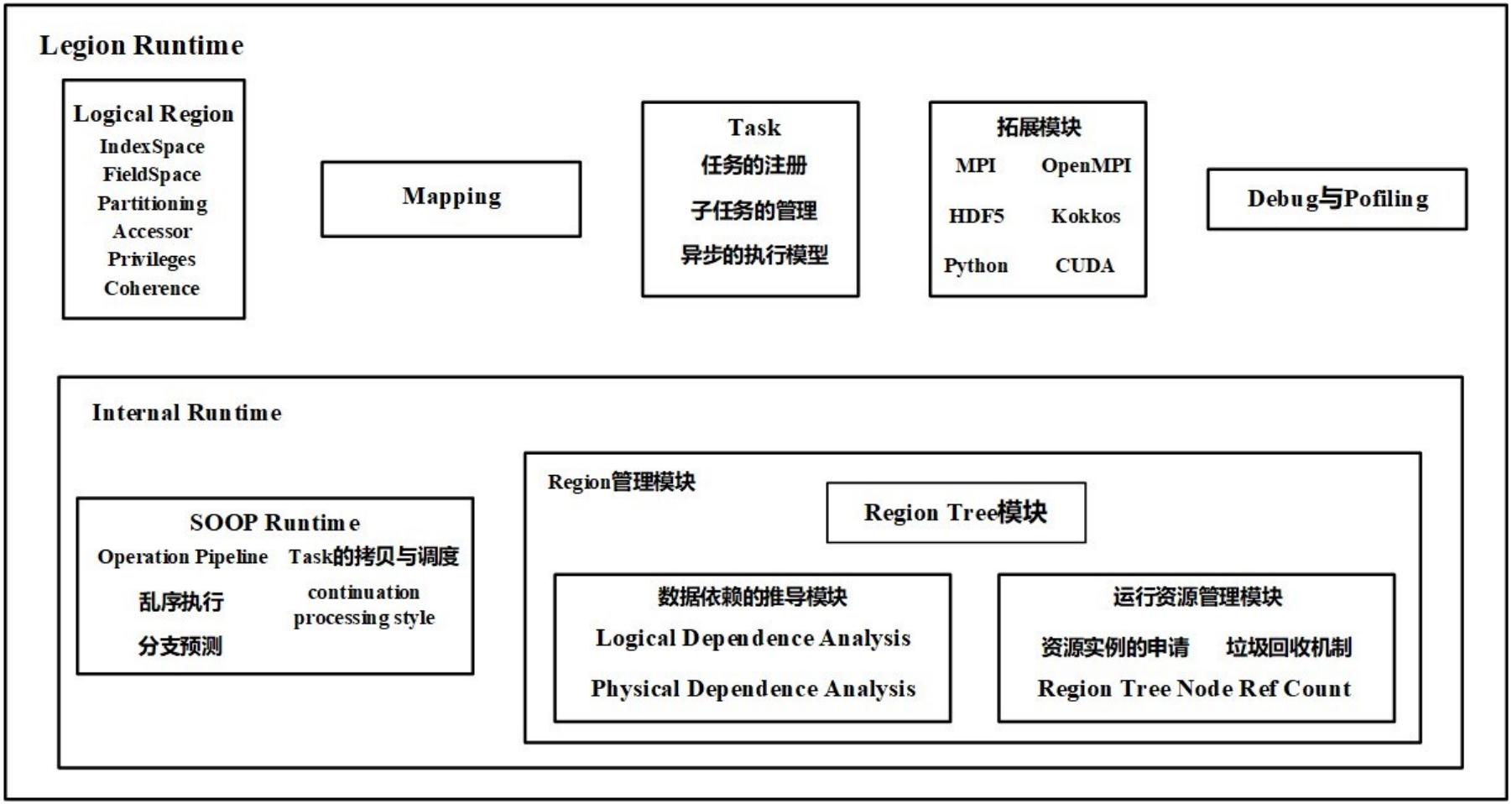
- events
- tasks/processors/scheduling
- index spaces/sparsity maps
- instance/accessors
- memories/allocation
- copies
- reservations
- dependent partitioning
- cuda
- networking
- logging/profiling
- start-up/tear-donw/modules

Legion的模块:

- operations and the pipeline
- tasks/contexts/scheduling
- mapping/mapper managers
- memories/instances/layouts/accessors
- index space and region trees
- index space expressions and common sub-expression elimination engine
- logical dependence analysis
  - tree traversal
  - refinement analysis
- physical dependence analysis
  - equivalence sets
  - physical analyses
  - copy fill aggregators
  - logical views
    - individual views: materialized and reduction
    - collective views: replicated and allreduce
    - deferred views: fll and phi
  - relaxed coherence
  - predication
- correctness debugging and legion spy
- futures and future maps
- profiling
- distributed collectable objects and memory management
- start-up and tear-down
- tracing
  - logical
  - physical
- control replication
  - shards and shard managers
  - replicated contexts
  - shard collectives
  - replicated operations
  - sharding functions and modifications to logical analysis

Legion软件栈详细介绍 - Legion

- Legion的核心功能模块

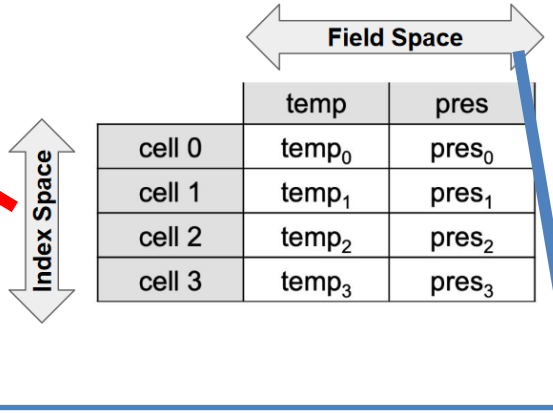


Legion软件栈详细介绍 - Legion

Logical Region
IndexSpace
FieldSpace
Partitioning
Accessor
Privileges
Coherence

- Legion中表示应用数据结构的Logical Region
  - **Logical** Region 用于抽象计算时所使用的数据，它类似一张关系表，由索引每条记录的索引空间(Index Space)以及表示每条记录所拥有的字段信息的字段空间(Field Space)所组成

```
// 索引空间的创建
// 非结构化的索引空间
const Domain domain(lo: DomainPoint(index: 0), hi: DomainPoint(index: 1023));
IndexSpace untyped_is = runtime->create_index_space(ctx, bounds: domain);

// 结构化的索引空间
const Rect<1> rect(lo: 0, hi: 1023);
IndexSpaceT<1> typed_is = runtime->create_index_space(ctx, bounds: rect);
```




```
// 字段空间的创建
// 动态地创建字段时,必须显式声明字段存储所需要的字节数,以及可选的字段ID
FieldSpace fs = runtime->create_field_space(ctx);
FieldAllocator allocator = runtime->create_field_allocator(ctx, handle: fs);
FieldID fida = allocator.allocate_field(field_size: sizeof(double), desired_fieldid: FID_FIELD_A);
FieldID fidb = allocator.allocate_field(field_size: sizeof(int), desired_fieldid: FID_FIELD_B);
```

Legion软件栈详细介绍 - Legion

Logical Region
IndexSpace
FieldSpace
Partitioning
Accessor
Privileges
Coherence

- Legion中表示应用数据结构的Logical Region
  - **Logical Region的属性 Privileges**：描述task在使用region时的权限，包括read-write、read-only等
  - **Logical Region的属性 Coherence**：描述region在被多个task并行使用时，Runtime要如何进行控制，包括atomic与exclusive等，从而不需要用户自己使用一些同步机制来保证数据的一致性
  - Privileges和Coherence为task的数据依赖分析与调度提供额外的信息

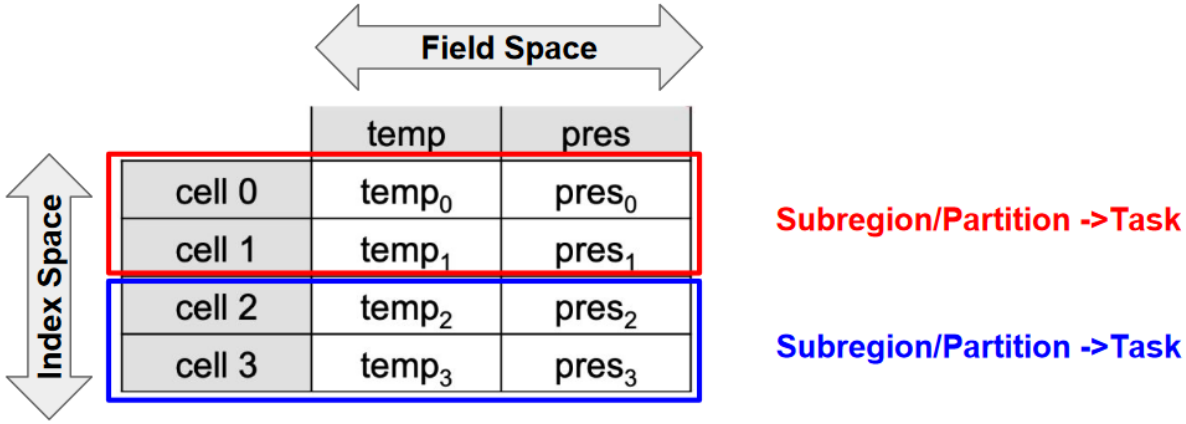
	Exclusive	Atomic	Simultaneous	Relaxed
Exclusive	Dep	Dep	Dep	Dep
Atomic	Dep	Same	Cont	Cont
Simultaneous	Dep	Cont	Same	None
Relaxed	Dep	Cont	None	None

Fig. 2. Dependence table.

Legion软件栈详细介绍 - Legion

Logical Region
IndexSpace
FieldSpace
Partitioning
Accessor
Privileges
Coherence

- Legion中表示应用数据结构的Logical Region
  - Legion为Logical Region提供了数据操作的接口，来方便描述并行计算中常见的数据操作，包括划分、拷贝以及launch为Physical Region等



  - Logical Region 无法被直接访问
    - 只是逻辑上的数据存储，可以设置简单的相同的初始值，没有映射到实际物理内存。
    - **需要从Logical Region到Physical Region**：通过Inline mapping进行launch，根据Logical Region的Privilege、Coherence以及Mapping规则映射到对应的物理内存上

Legion软件栈详细介绍 - Legion

Logical Region
IndexSpace
FieldSpace
Partitioning
Accessor
Privileges
Coherence

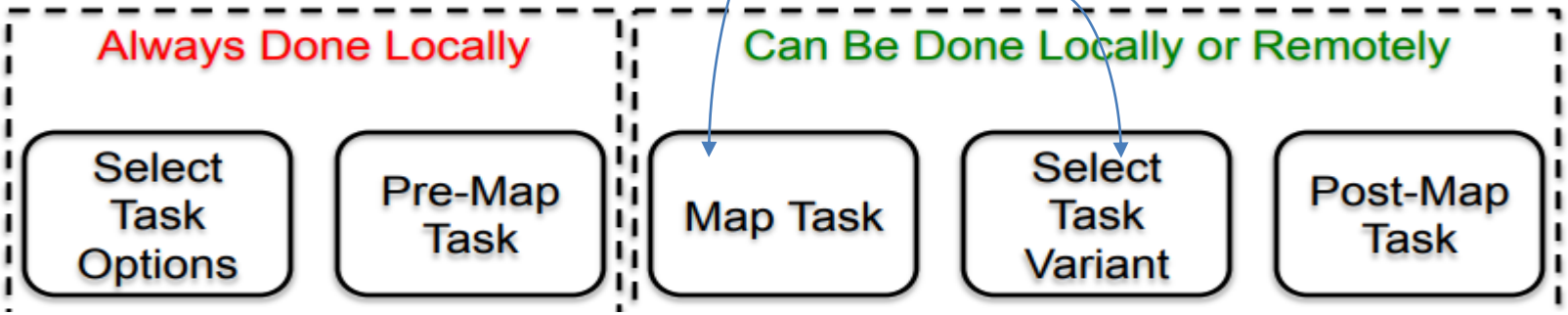
- Legion中从Logical Region到Physical Region
  - Step1：描述Region的需求（包括index，field，Privilege，Coherence）
  - Step2：初始化Logical Region的Inline Mapping
  - Step3：决定是否新建一个Physical Region实例还是重用老实例
  - Step4：决定Physical Region实例中要存储哪些字段Field
  - Step5：决定需要多少内存空间用于存放Physical Region实例
  - Step6：决定Physical Region实例要如何布局（例如AOS或SOA）
  - Step7：决定哪些Task对数据进行操作
  - Step8：决定哪些数据需要被拷贝（例如传输到其他位置的内存中）
  - Step9：等待，直到任务或者拷贝完成

Application　Runtime　Mapper

Legion软件栈详细介绍 - Legion

Task
任务的注册
子任务的管理
异步的执行模型

- Legion中表示计算工作流的Task
  - Legion利用Task来描述计算的工作流，是Legion中调度的基本单元。
  - 每个Task运行时与特定的Physical Region实例相关联，且向运行时提供Task要如何使用Region的相关信息，还要由Mapping策略放置到处理器上执行
  - 实际编写代码时，Task是拥有特定签名与返回值的函数，且运行时以异步的方式来执行每个Task
  - 允许用户为每个Task定义面向不同计算设备的Task变体
  - 每个Task在运行时都允许动态地产生子任务，并可以异步地等待子任务完成



Legion软件栈详细介绍 - Legion

Task
任务的注册
子任务的管理
异步的执行模型

- Legion中表示计算工作流的Task
  - 例子：子任务的派生和异步执行 <Future编程模型>

```
int fibonacci_task(const Task *task,
                   const std::vector<PhysicalRegion> &regions,
                   Context ctx, Runtime *runtime) {
    assert(task->arglen == sizeof(int));
    int fib_num = *(const int *) task->args;
    if (fib_num == 0)
        return 0;
    if (fib_num == 1)
        return 1;

    // Launch fib-1
    const int fib1 = fib_num - 1;
    TaskLauncher t1(tid: FIBONACCI_TASK_ID, arg: TaskArgument(arg: &fib1, argsize: sizeof(fib1)));
    Future f1 = runtime->execute_task(ctx, launcher: t1);

    // Launch fib-2
    const int fib2 = fib_num - 2;
    TaskLauncher t2(tid: FIBONACCI_TASK_ID, arg: TaskArgument(arg: &fib2, argsize: sizeof(fib2)));
    Future f2 = runtime->execute_task(ctx, launcher: t2);

    TaskLauncher sum(tid: SUM_TASK_ID, arg: TaskArgument(arg: NULL, argsize: 0));
    sum.add_future(f: f1);
    sum.add_future(f: f2);
    Future result = runtime->execute_task(ctx, launcher: sum);

    return result.get_result<int>();
}
```

Legion软件栈详细介绍 - Legion

Task
任务的注册
子任务的管理
异步的执行模型

- Legion中表示计算工作流的Task
  - 例子：Task变体的注册

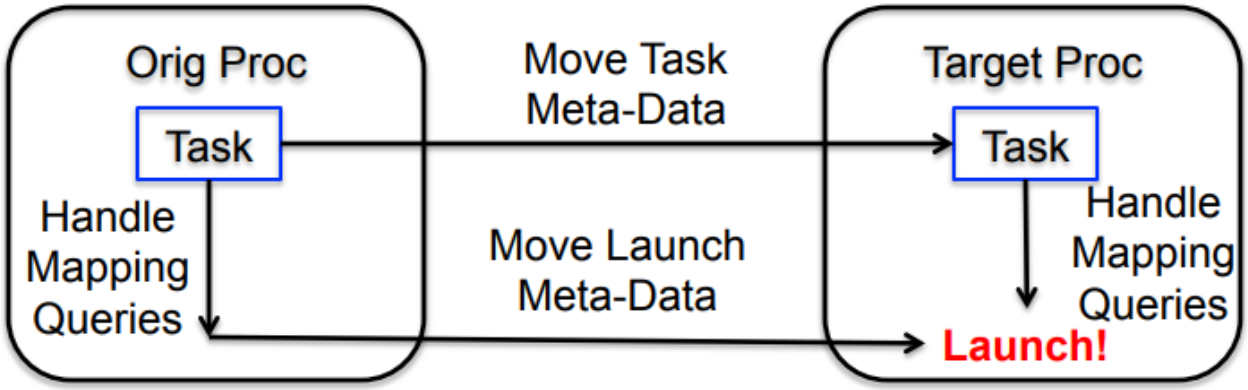
```
int main(int argc, char **argv) {
    // Before starting the Legion runtime, you first have to tell it
    // NOTE 运行时从一个顶层的任务开始启动,需要设置顶层任务的id
    Runtime::set_top_level_task_id(top_id: HELLO_WORLD_ID);

    {
        // NOTE 进行任务变体的注册,
        //  任务允许有不同的变体，同类任务的不同变体之间运行的任务函数是是一致的,
        //  但运行的条件与环境允许有所不同，比如使用的计算核、数据的布局等
        TaskVariantRegistrar registrar(task_id: HELLO_WORLD_ID, variant_name: "hello_world");
        registrar.add_constraint(constraint: ProcessorConstraint(kind: Processor::LOC_PROC));
        Runtime::preregister_task_variant<hello_world_task>(registrar, task_name: "hello_world");
    }

    // Now we're ready to start the runtime, so tell it to begin the
    // execution.  We'll only return from this call once the Legion
    // program is done executing.
    return Runtime::start(argc, argv);
}
```

Legion软件栈详细介绍 - Legion

Task
任务的注册
子任务的管理
异步的执行模型

- Legion中表示计算工作流的Task
  - 例子：Task的Mapping [固定Task的Owner Processor]

- **Locally:** `current_proc == orig_proc`
- **Remotely:** `current_proc == target_proc ( != orig_proc )`

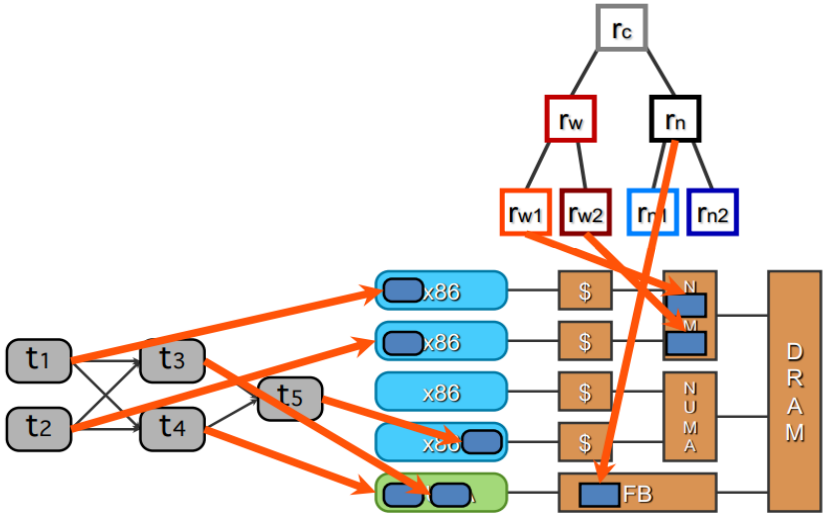


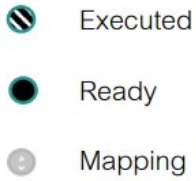
Task Meta-Data >> Launch Meta-Data

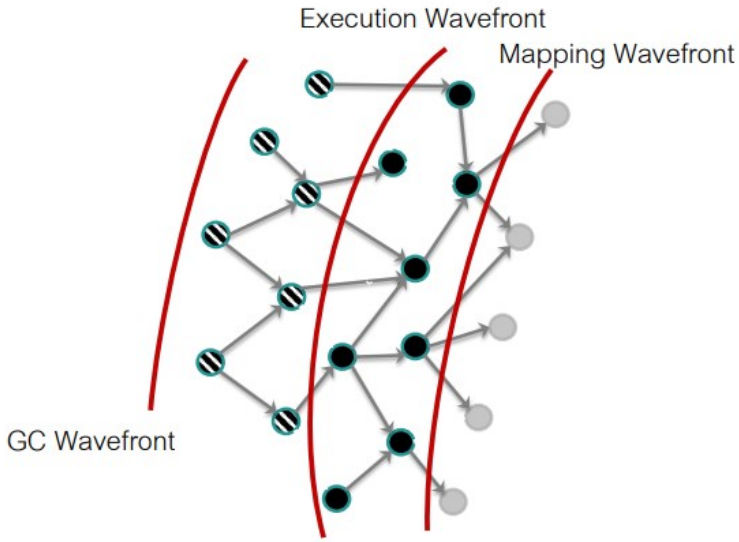
Remote Mapping -> More Parallelism

Legion软件栈详细介绍 - Legion

- Legion中的Mapping
  - Legion需要决定：Region在哪里放置； Task在哪个Processor执行
  - 异步Mapping: After tasks are mapped, they are distributed to their target nodes.

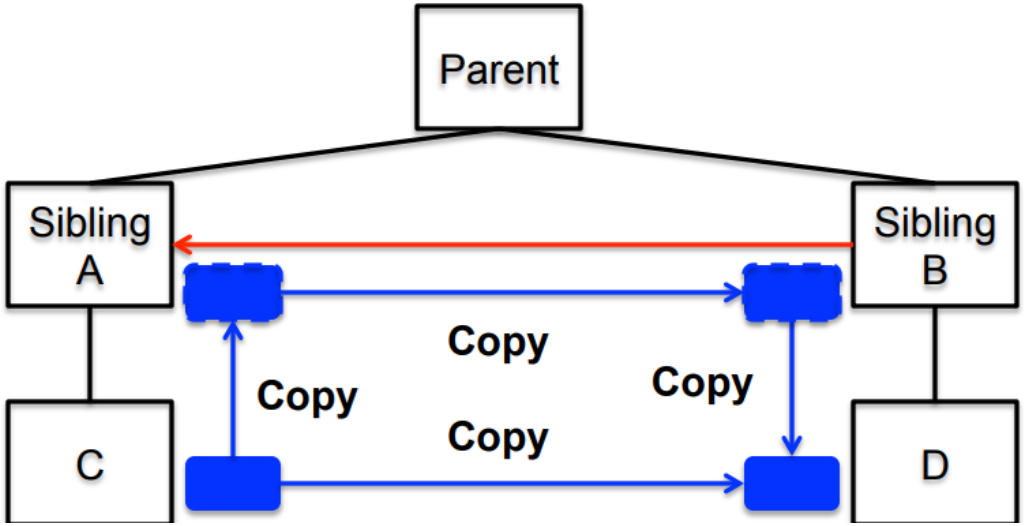


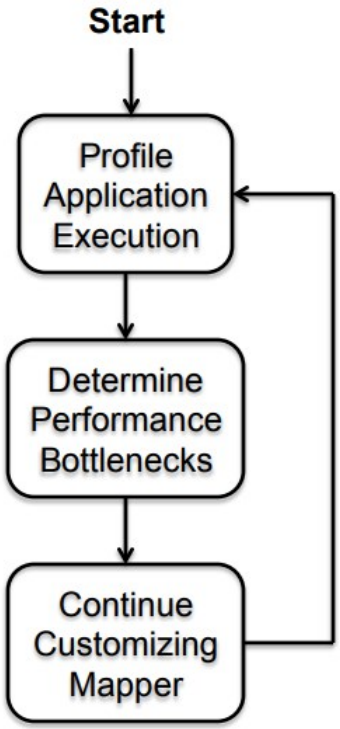




Legion软件栈详细介绍 - Legion

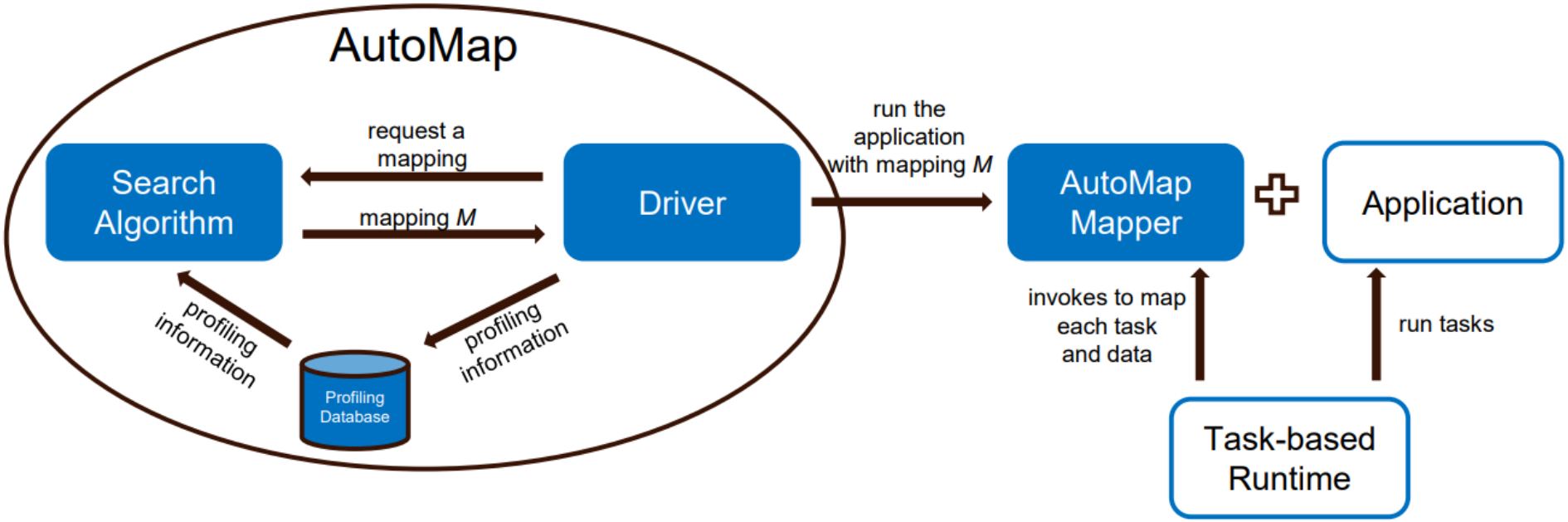
- Legion中的Mapping
  - 无为而治：Legion只提供默认策略，用户自己设计的具体Mapping策略
  - 支持节点间负载均衡策略：push Tasks / pull Tasks
  - 支持节点内负载均衡策略：支持Mapping任务到多个处理器上
  - 支持资源死锁检测和处理





Legion软件栈详细介绍 - Legion

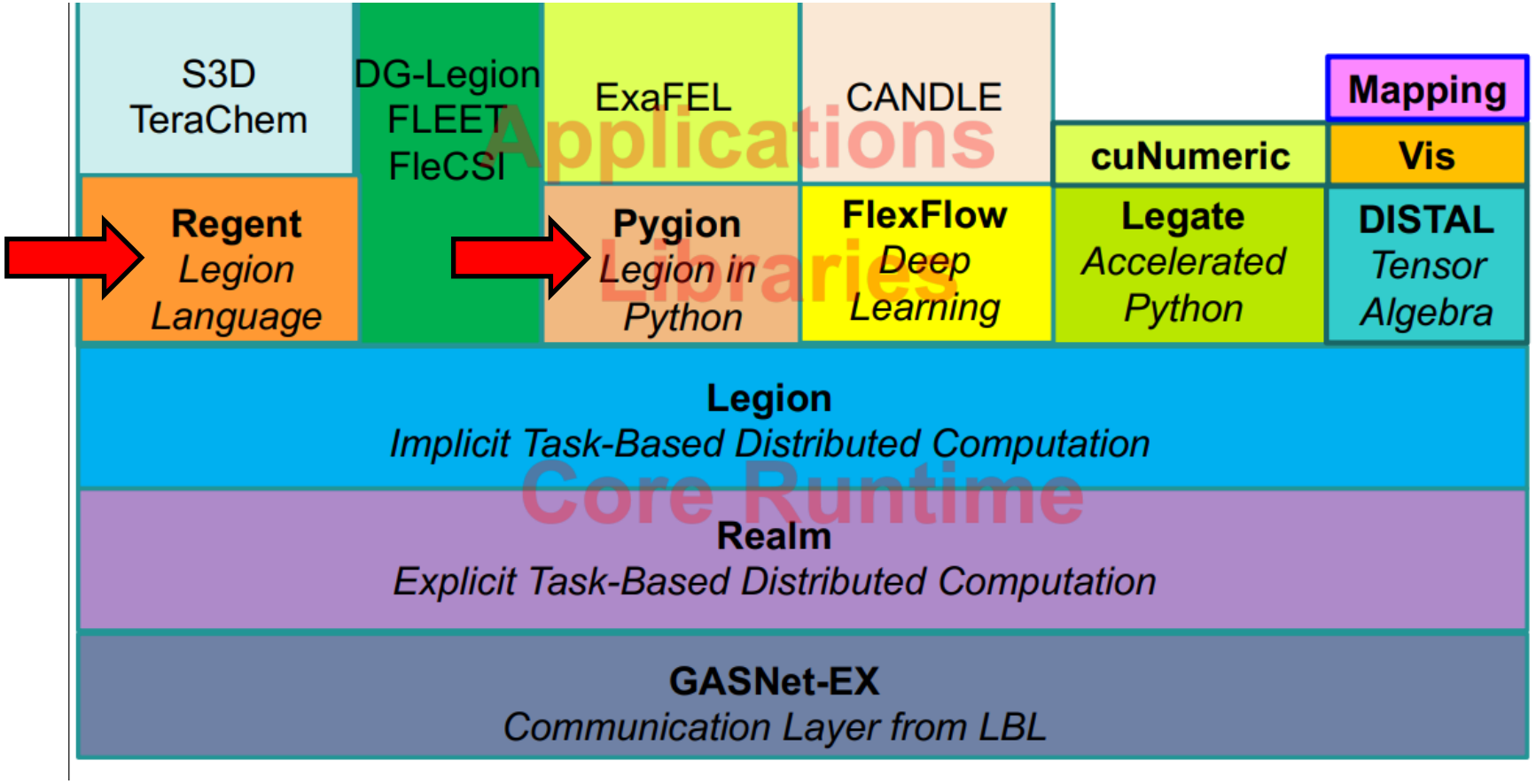
- Legion中的Mapping Tool：Automap[1]
  - 以迭代式应用的形式工作
  - 从多种Mapping策略中进行离线Profile，智能地搜索最优的Mapping策略



[1] SFX Teixeira, Thiago, et al. "Automated Mapping of Task-Based Programs onto Distributed and Heterogeneous Machines." Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. 2023.

Legion软件栈详细介绍 – Libraries/Apps

- Legion的上层库和应用
  - 第一类：易用的领域语言Regent和动态类型语言适配Pygion



Legion软件栈详细介绍 – Libraries/Apps

- Regent[1]：易用的静态类型领域语言
  - Regent简化了Legion的编程模型，构建了一套领域语言
  - Regent程序的性能和精细调整的Legion程序具有相同的性能

Legion
业务代码

```
runtime->unmap_region(ctx, physical_r);
TaskLauncher launcher_A(TASK_A, TaskArgument());
launcher_A.add_region_requirement(
  RegionRequirement(r, READ_WRITE, EXCLUSIVE, r));
launcher_A.add_field(0, FIELD_X);
launcher_A.add_field(0, FIELD_Y);
runtime->execute_task(ctx, launcher_A);
Domain domain = Domain::from_rect<1>(
  Rect<1>(Point<1>(0), Point<1>(2)));
IndexLauncher launcher_B(TASK_B, domain,
  TaskArgument(), ArgumentMap());
launcher_B.add_region_requirement(
  RegionRequirement(p, 0 /* projection */,
    READ_WRITE, EXCLUSIVE, r));
launcher_B.add_field(0, FIELD_X);
runtime->execute_index_space(ctx, launcher_B);
TaskLauncher launcher_C(TASK_A, TaskArgument());
launcher_C.add_region_requirement(
  RegionRequirement(r, READ_ONLY, EXCLUSIVE, r));
launcher_C.add_field(0, FIELD_X);
launcher_C.add_field(0, FIELD_Y);
```

Regent
业务代码

```
A(r)
for i = 0, 3 do
  B(p[i])
end
C(r)
```

[1] Slaughter, Elliott, et al. "Regent: a high-productivity programming language for HPC with logical regions." Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. 2015.

Legion软件栈详细介绍 – Libraries/Apps

- Pygion[1]：Python动态类型语言支持

Legion
业务代码

```
enum FIELD_IDS { FID_X, FID_Y };
enum TASK_IDS { TID_SAXPY, TID_MAIN };
void saxpy(const Task *task, const std::vector<PhysicalRegion> &regions,
           Context ctx, Runtime *runtime) {
  FieldAccessor<READ_WRITE,float,1> acc_y(regions[0], FID_Y);
  FieldAccessor<READ_WRITE,float,1> acc_x(regions[1], FID_X);
  float a = *(const float*)(task->args);
  Rect<1> rect = runtime->get_index_space_domain(
      ctx, task->regions[0].region.get_index_space());
  for (PointInRectIterator<1> i(rect); i(); i++)
    acc_y[*i] += + a * acc_x[*i];
}
void main(const Task *task, const std::vector<PhysicalRegion> &regions,
          Context ctx, Runtime *runtime) {
  IndexSpace I = runtime->create_index_space(ctx, Rect<1>(0, 9));
  FieldSpace F = runtime->create_field_space(ctx);
  FieldAllocator allocator = runtime->create_field_allocator(ctx, F);
  allocator.allocate_field(sizeof(float), FID_X);
  allocator.allocate_field(sizeof(float), FID_Y);
  LogicalRegion S = runtime->create_logical_region(ctx, I, F);
  IndexSpace colors = runtime->create_index_space(ctx, Rect<1>(0, 1));
  IndexPartition IP = runtime->create_equal_partition(ctx, I, colors);
  LogicalPartition P = runtime->get_logical_partition(ctx, S, IP);
  float a = 1.23;
  IndexLauncher launch(TID_SAXPY, colors, TaskArgument((void *)&a, sizeof(a)),
    ArgumentMap());
  launch.add_region_requirement(RegionRequirement(
      P, 0, READ_WRITE, EXCLUSIVE, S));
  launch.add_region_requirement(RegionRequirement(
      P, 0, READ_ONLY, EXCLUSIVE, S));
  launch.add_field(0, FID_Y);
  launch.add_field(1, FID_X);
```

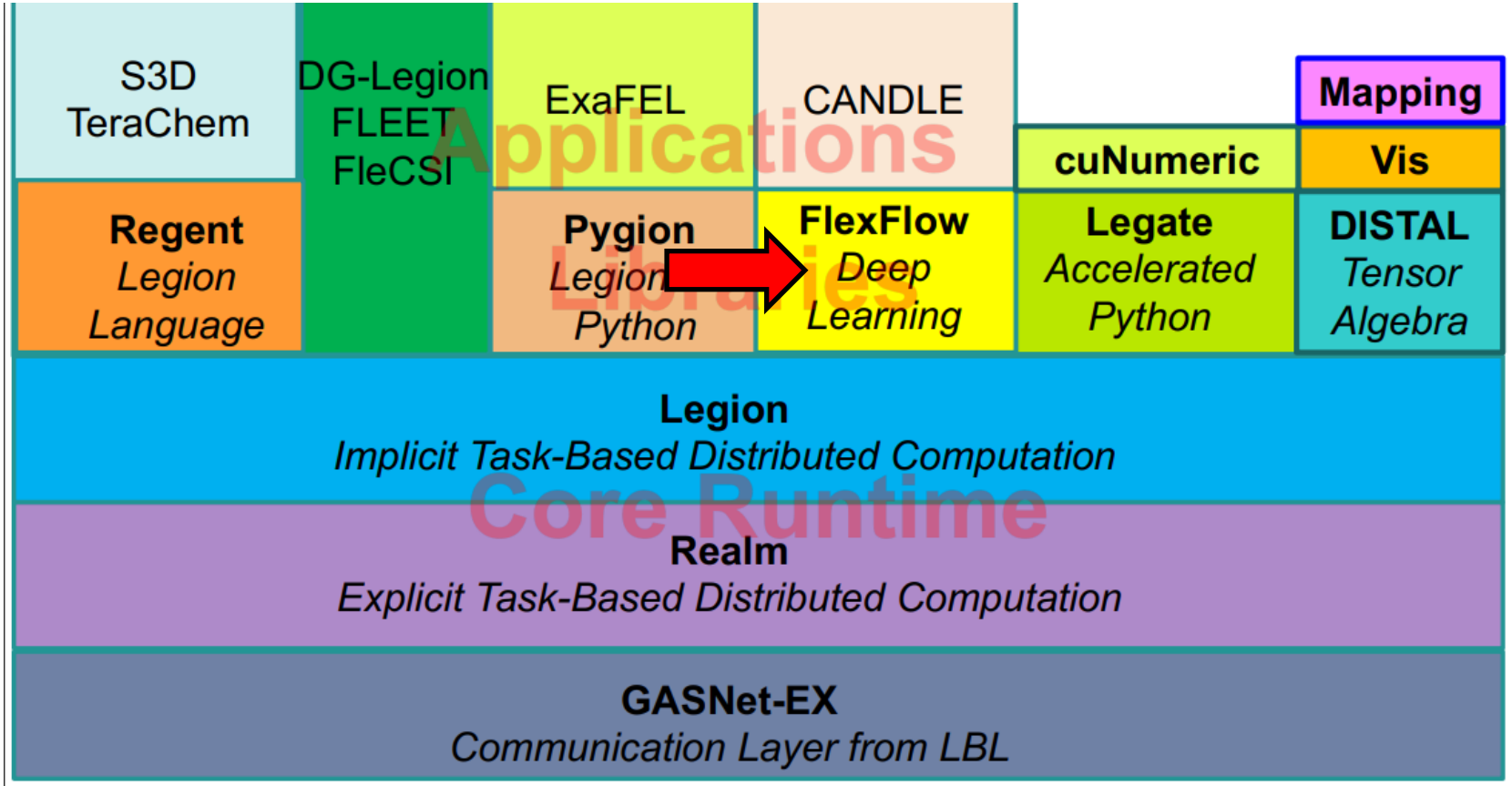
Pygion
业务代码

```
@task(privileges=[RW('y')+R('x')])
def saxpy(S, a):
  S.y += a * S.x

S = Region([10], {
        'x': float, 'y': float})
P = Partition.equal(S, [2])
for i in IndexLaunch([2]):
    saxpy(P[i], 1.23)
```

[1] Slaughter, Elliott, and Alex Aiken. "Pygion: Flexible, scalable task-based parallelism with python." 2019 IEEE/ACM Parallel Applications Workshop, Alternatives To MPI (PAW-ATM). IEEE, 2019.

Legion软件栈详细介绍 – Libraries/Apps

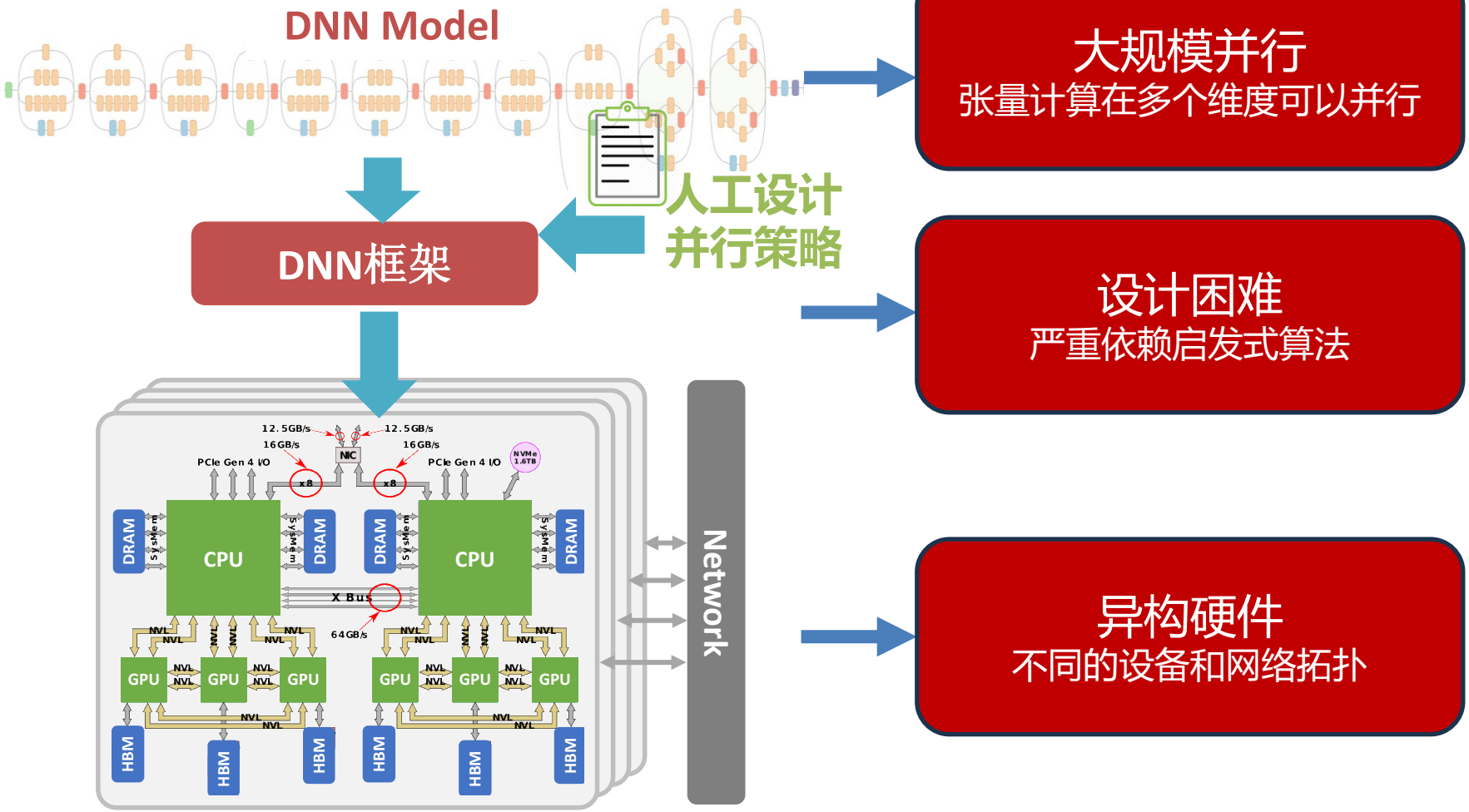
- Legion的上层库和应用
  - 第二类：针对深度学习任务的加速库 FlexFlow[1]



[1] Lu, Wenyan, et al. "Flexflow: A flexible dataflow accelerator architecture for convolutional neural networks." 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA). IEEE, 2017.

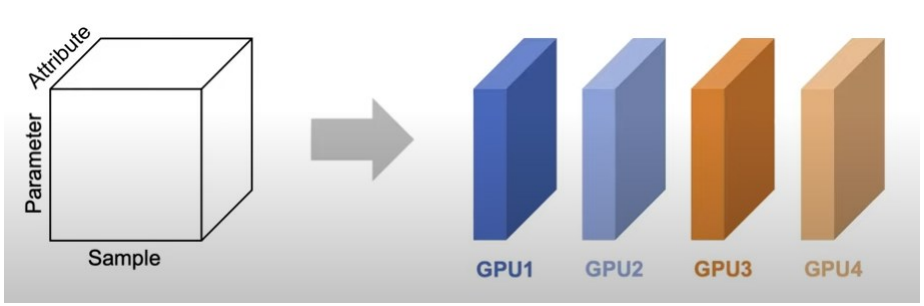
Legion软件栈详细介绍 – Libraries/Apps

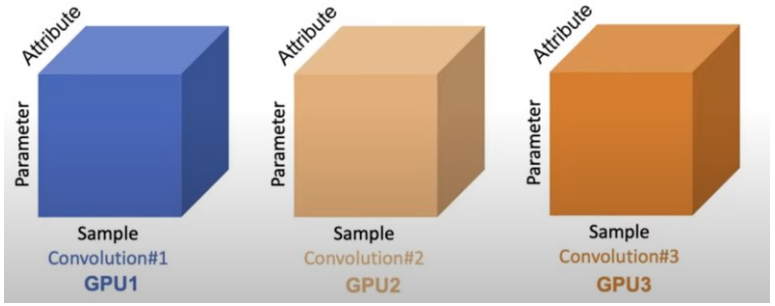
- Flexflow：自动深度学习任务并行优化底层支持库
  - 深度学习并行优化的挑战

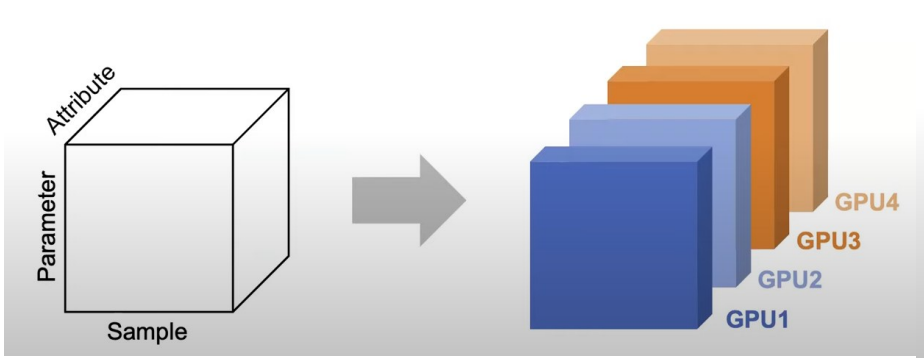


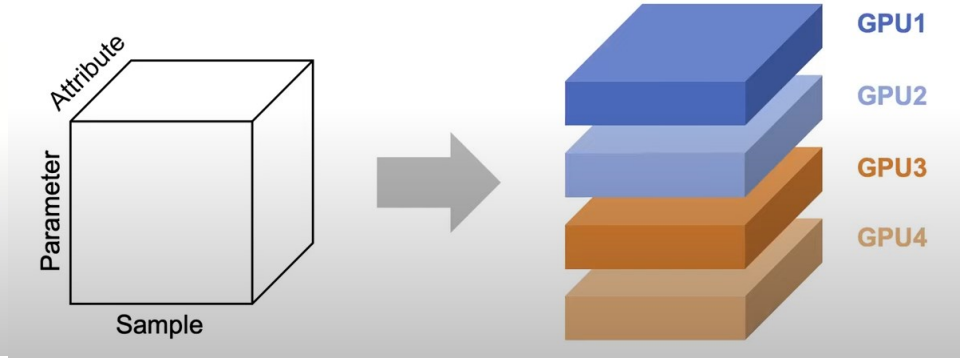
Legion软件栈详细介绍 – Libraries/Apps

- Flexflow 定义了并行策略的决策搜索空间： SOAP
  - **S**ample： 划分训练样本（数据并行）
  - **O**perator：划分ML算子 （模型并行）
  - **A**ttributes：在一个样本中划分属性（比如Pixels）
  - **P**arameters：在一个ML算子中划分参数



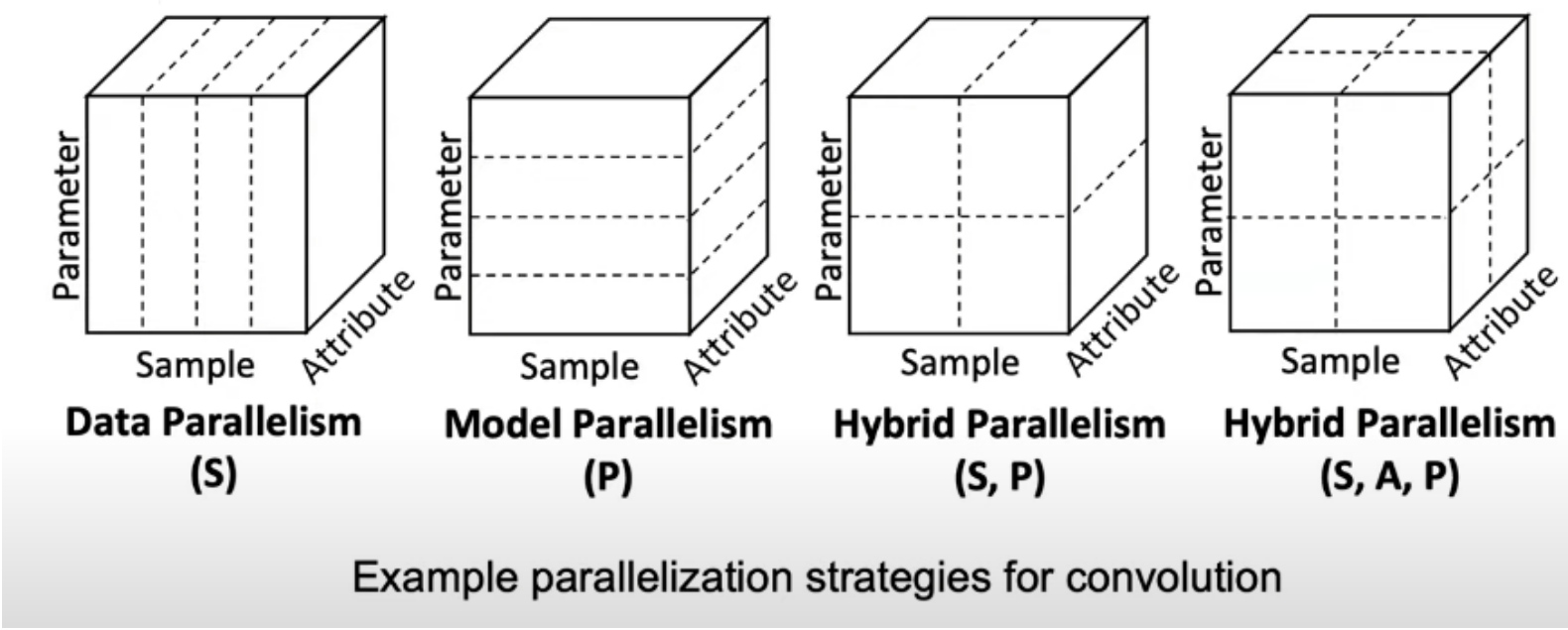






Legion软件栈详细介绍 – Libraries/Apps

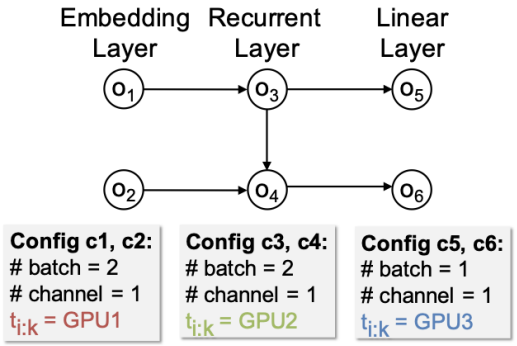
- SOAP混合并行的示例



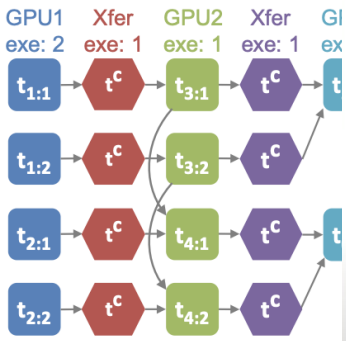
- SOAP混合并行的挑战
  - 搜索空间巨大 => 解决方案：MCMC搜索算法
  - 在硬件上评估一种策略的性能非常慢 => 解决方案：执行模拟器

Legion软件栈详细介绍 – Libraries/Apps

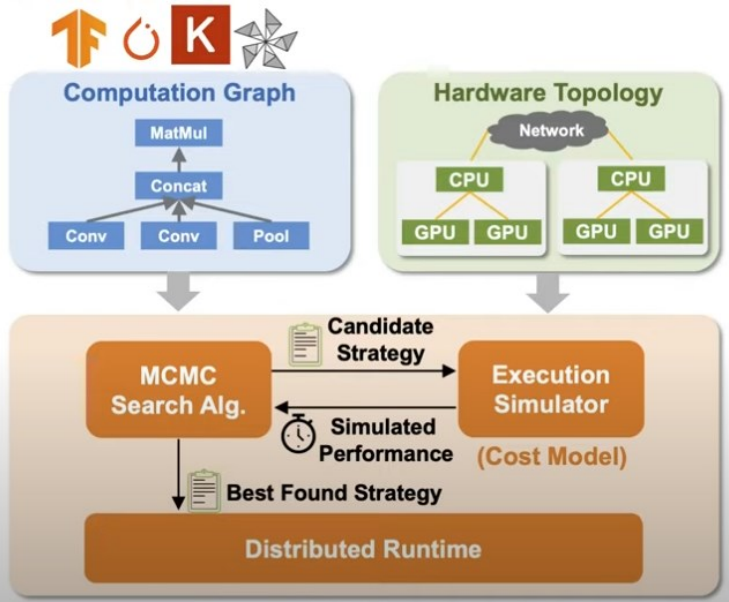
- Flexflow 搜索优化方案
  - Step.1 构建初始的并行策略，构建计算和传输的时空依赖图 【图a和b】
  - Step.2 执行全模拟算法(Full Simulation Algorithm)，利用Dijkstra算法获取执行时间 【图c，执行模拟器完成】
  - Step.3 利用MCMC搜索算法获取新的并行策略【执行优化器完成】
  - Step.4 利用delta模拟算法(Delta Simulation Algorithm)，获取新策略的执行时间 【图d，执行模拟器完成】

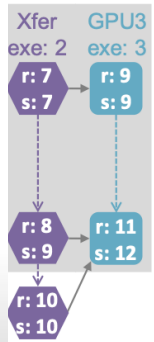


(a) An example parallelization strategy.



(b) The corresponding task graph.

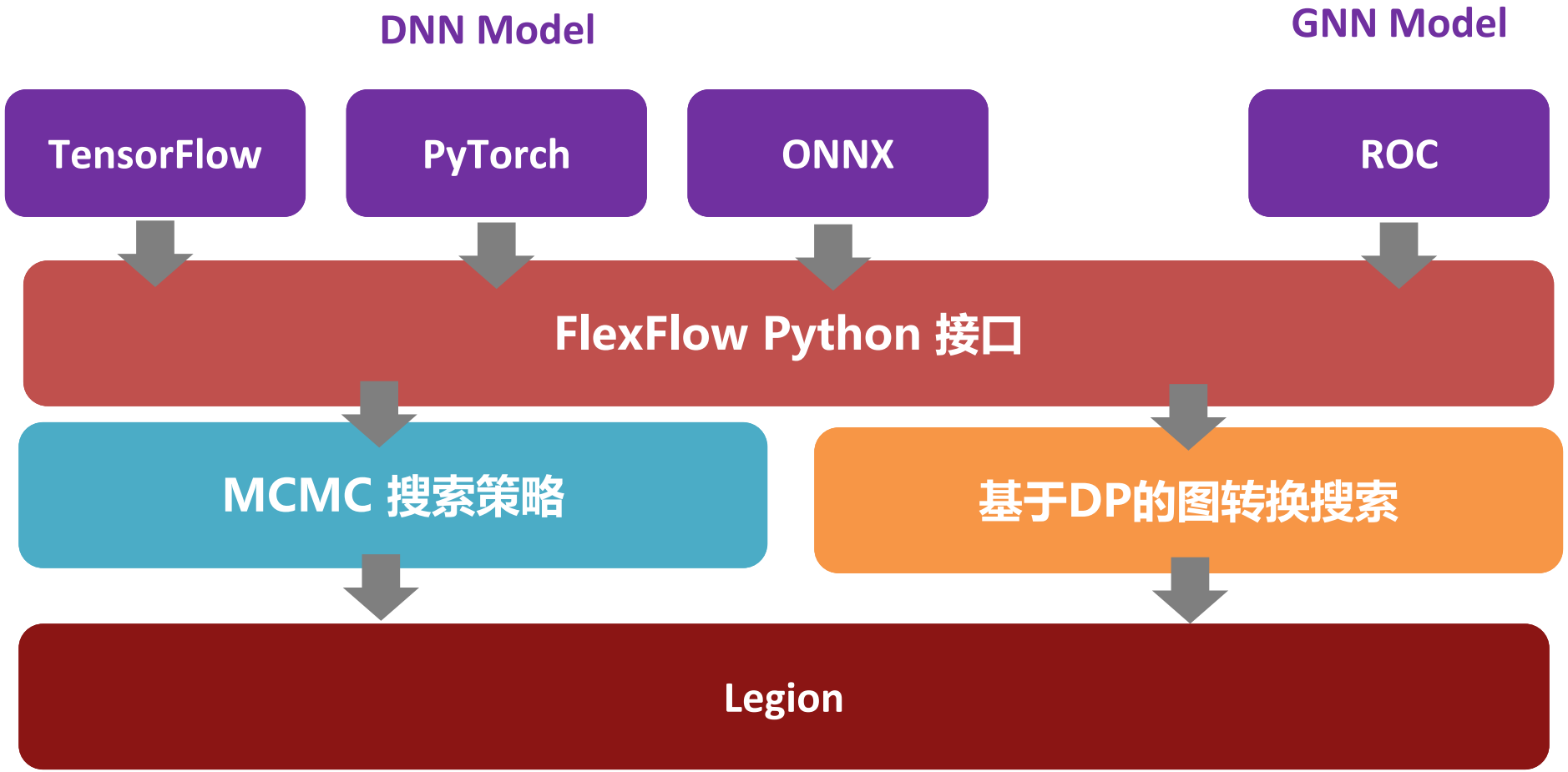




ıfter the
orithm.

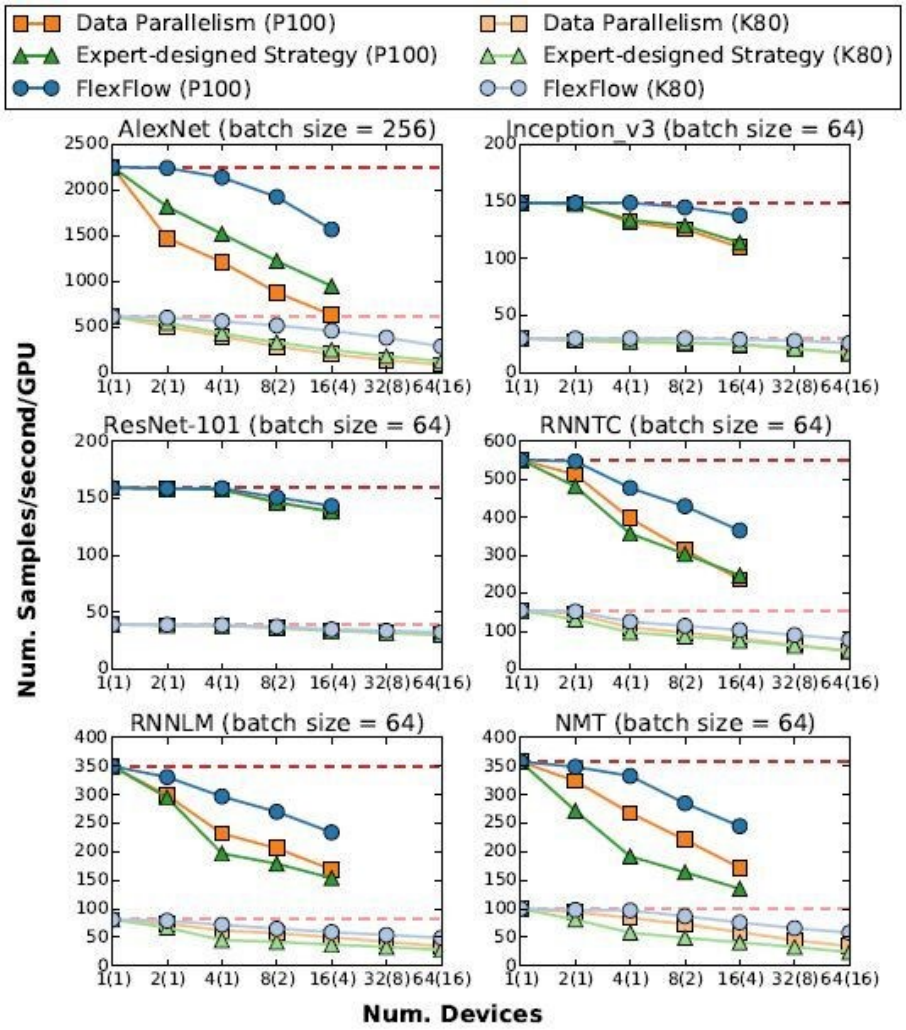
Legion软件栈详细介绍 – Libraries/Apps

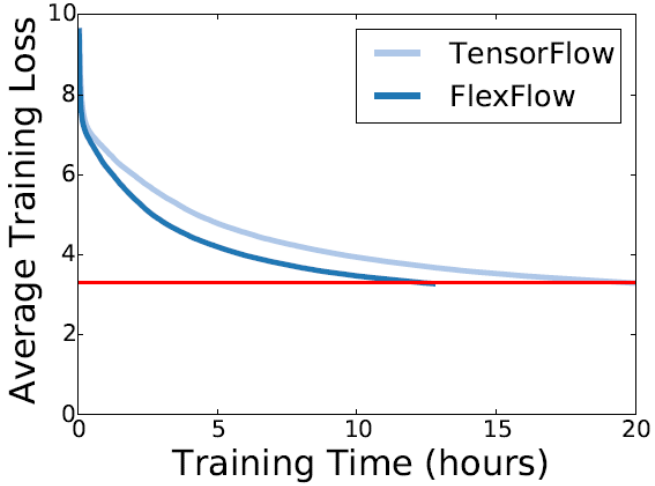
- Flexflow：自动深度学习任务并行优化底层支持库
  - Flexflow软件生态

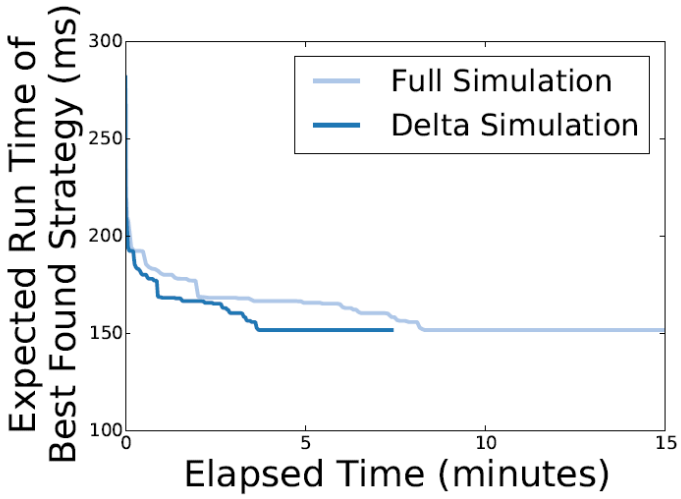


Legion软件栈详细介绍 – Libraries/Apps

- Flexflow 性能

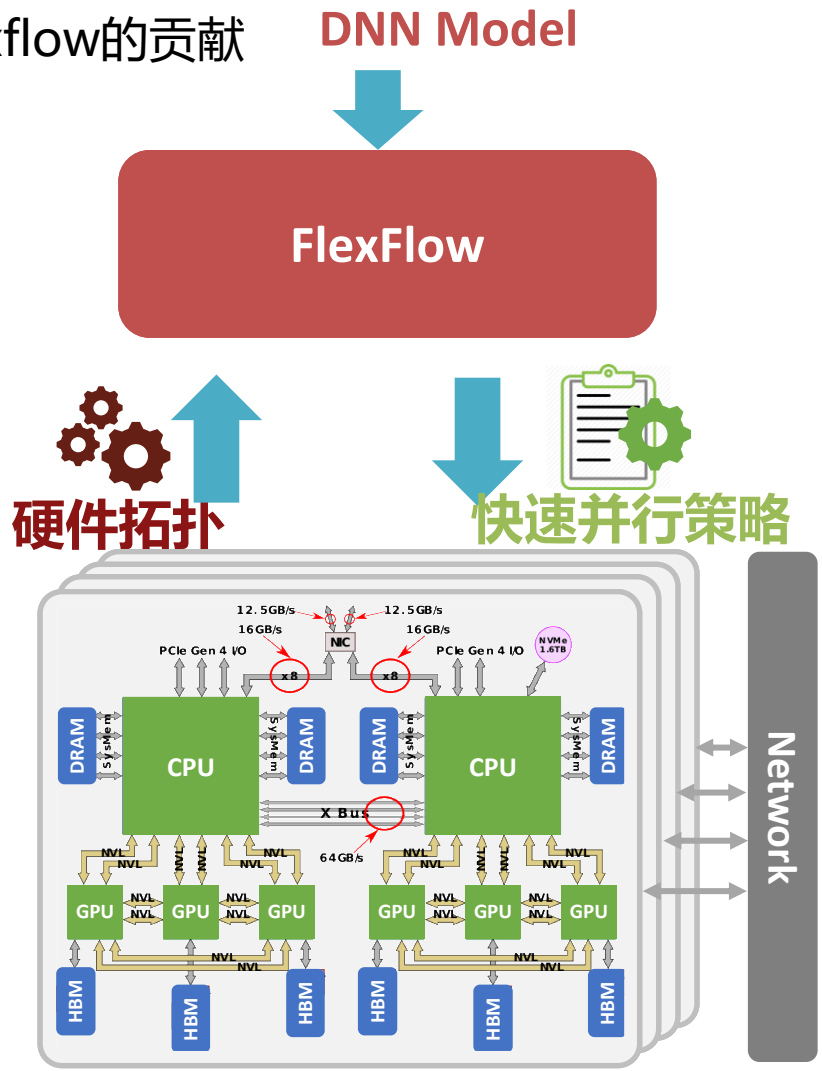






Legion软件栈详细介绍 – Libraries/Apps

- Flexflow：自动深度学习任务并行优化底层支持库
  - Flexflow的贡献



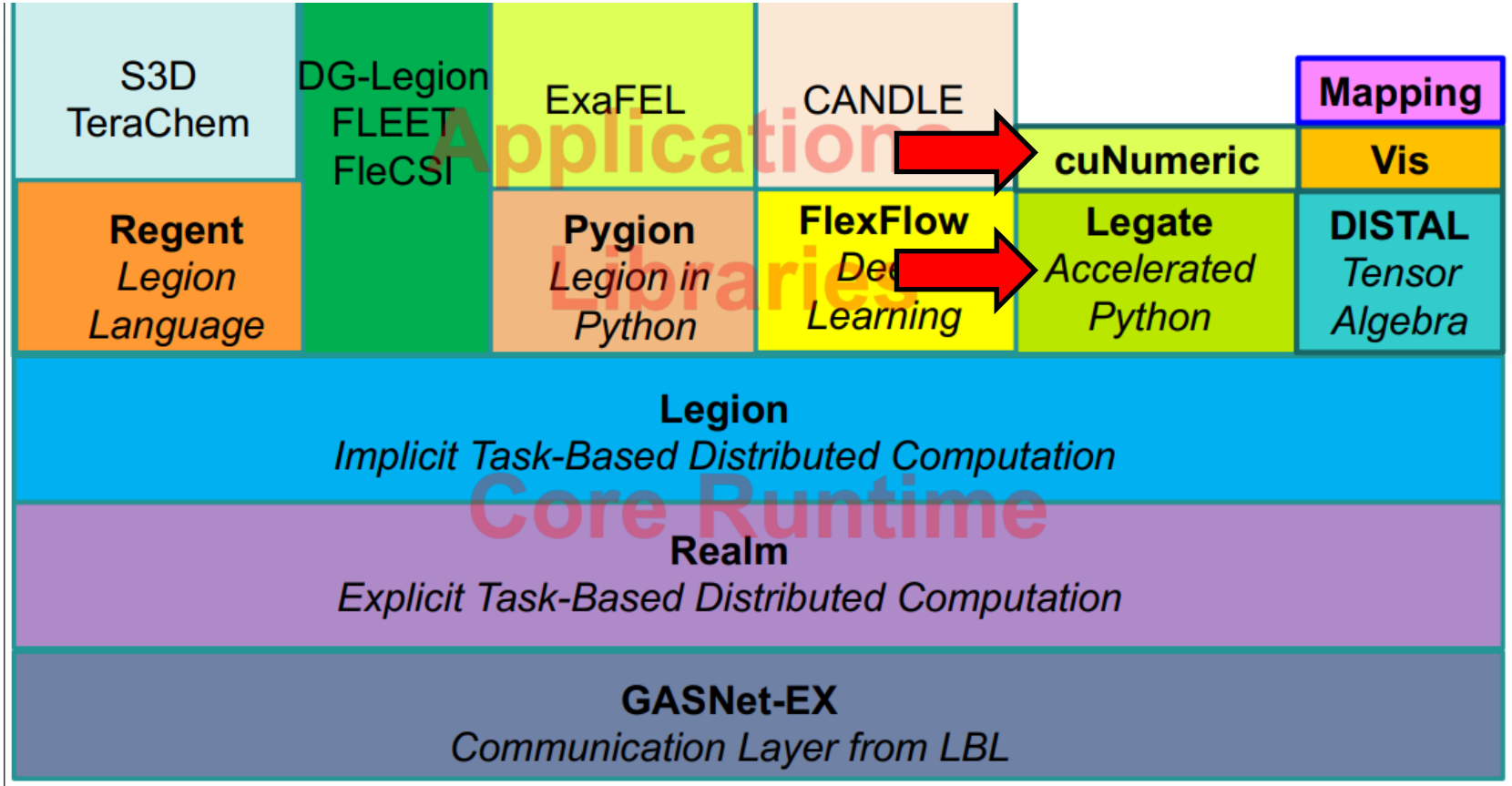
更好的性能
相比当时的手动调整策略获得了10x速度提升

快速部署
自动搜索策略用于发现性能更高的策略

快速迁移
针对全新的硬件和模型都可以快速迁移

Legion软件栈详细介绍 – Libraries/Apps

- Legion的上层库和应用
  - 第三类：Python加速库Legate[1] 和 针对Numpy的加速应用cuNumeric[1]



[1] Bauer, Michael, and Michael Garland. "Legate NumPy: Accelerated and distributed array computing." Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. 2019.

Legion软件栈详细介绍 – Libraries/Apps

- Legate + cuNumeric
  - 背景：Numpy是稠密矩阵计算的最受欢迎的Python包
  - Legate + cuNumeric：只需要一行代码的修改即可完成并行和分布式加速

```
import numpy as np

def cg_solve(A, b, tol=1e-10):
    x = np.zeros(A.shape[1])
    r = b - A.dot(x)
    p = r
    rsold = r.dot(r)
    for i in xrange(b.shape[0]):
        Ap = A.dot(p)
        alpha = rsold / (p.dot(Ap))
        x = x + alpha * p
        r = r - alpha * Ap
        rsnew = r.dot(r)
        if np.sqrt(rsnew) < tol:
            break
        beta = rsnew / rsold
        p = r + beta * p
        rsold = rsnew
    return x
```

```
import legate.numpy as np

def cg_solve(A, b, tol=1e-10):
    x = np.zeros(A.shape[1])
    r = b - A.dot(x)
    p = r
    rsold = r.dot(r)
    for i in xrange(b.shape[0]):
        Ap = A.dot(p)
        alpha = rsold / (p.dot(Ap))
        x = x + alpha * p
        r = r - alpha * Ap
        rsnew = r.dot(r)
        if np.sqrt(rsnew) < tol:
            break
        beta = rsnew / rsold
        p = r + beta * p
        rsold = rsnew
    return x
```

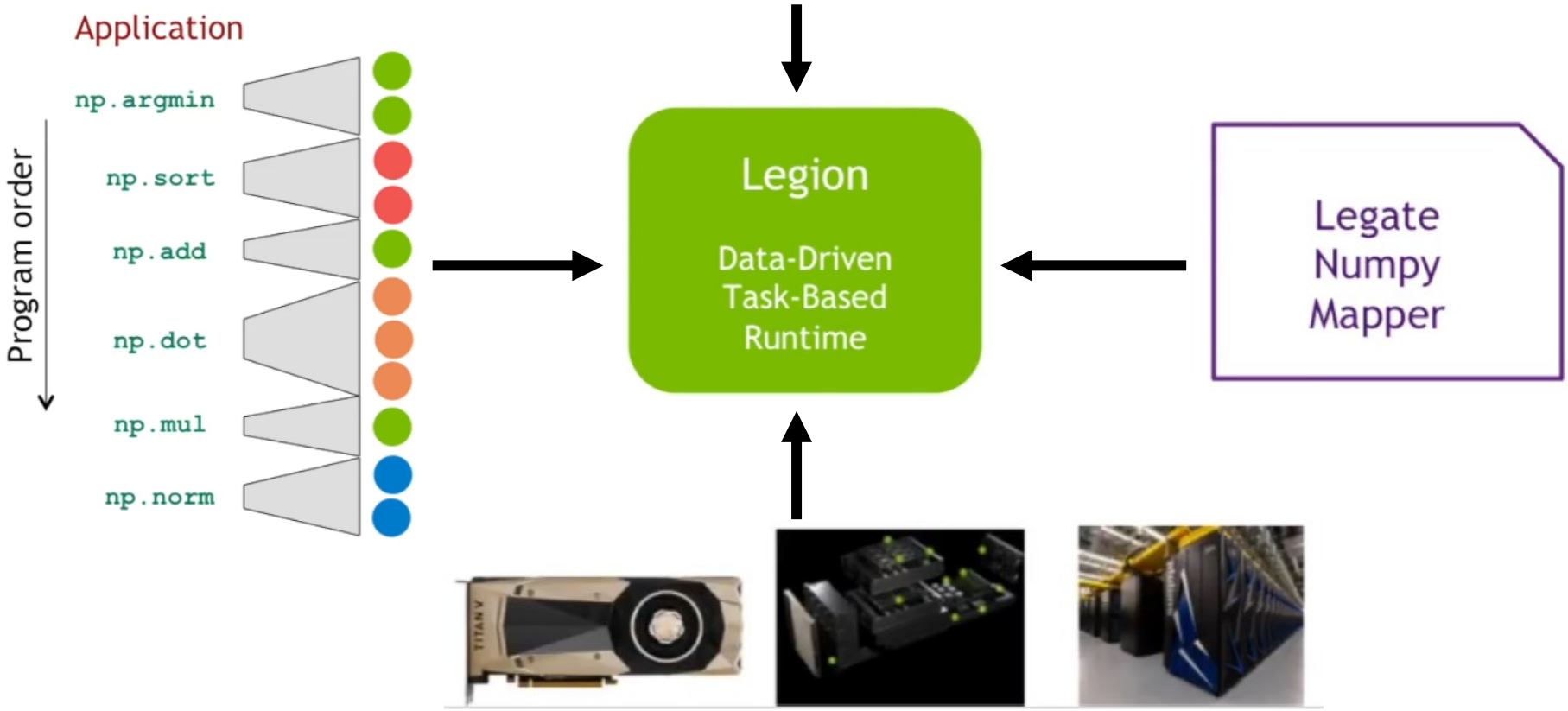
Legion软件栈详细介绍 – Libraries/Apps

- Legate + cuNumeric

cuNumeric将API调用转化为Legion Task

cuNumeric提供更简易的任务实现(Python)

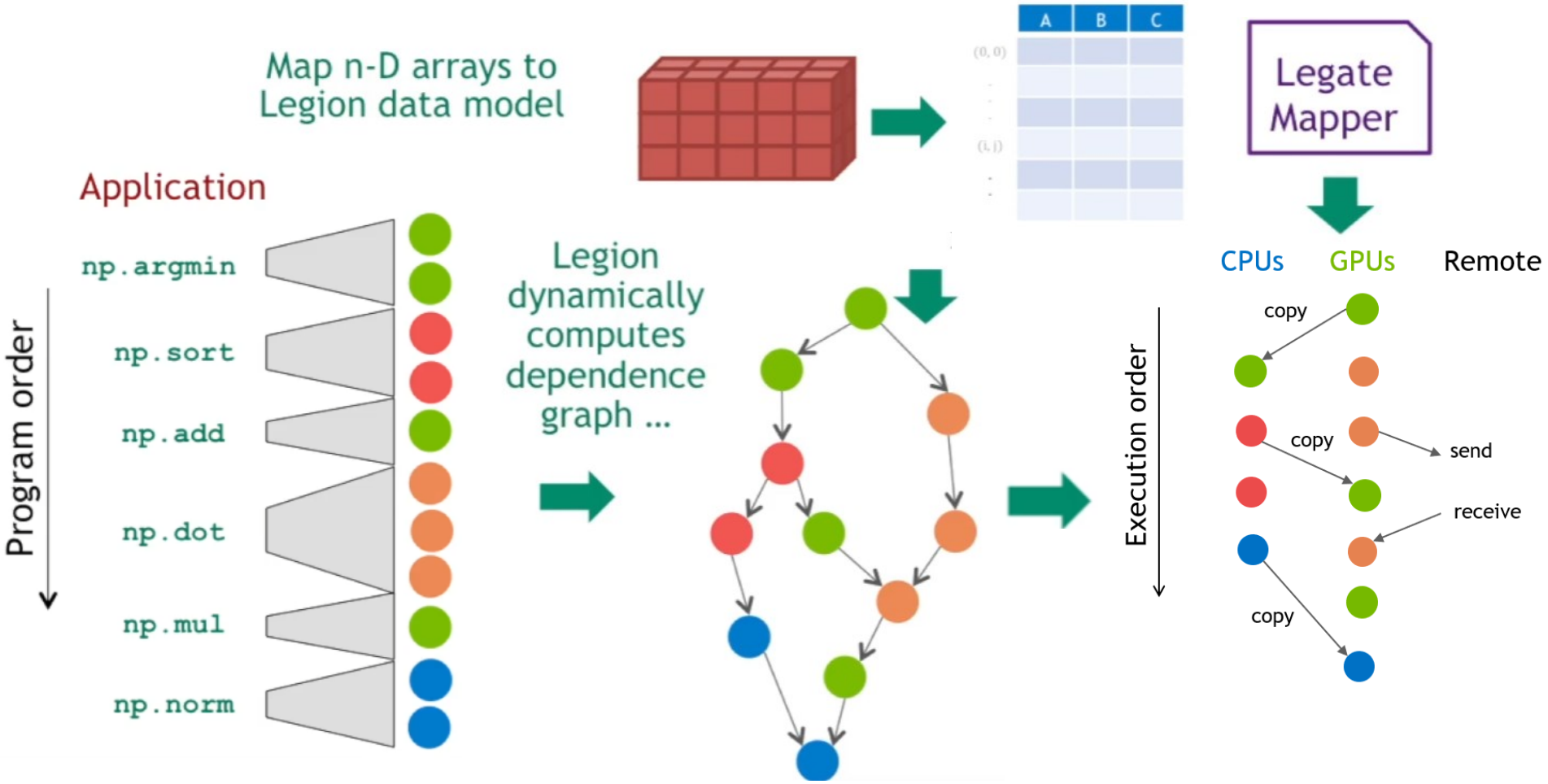
cuNumeric提供Legion Mapping的自定义实现



异构计算设备

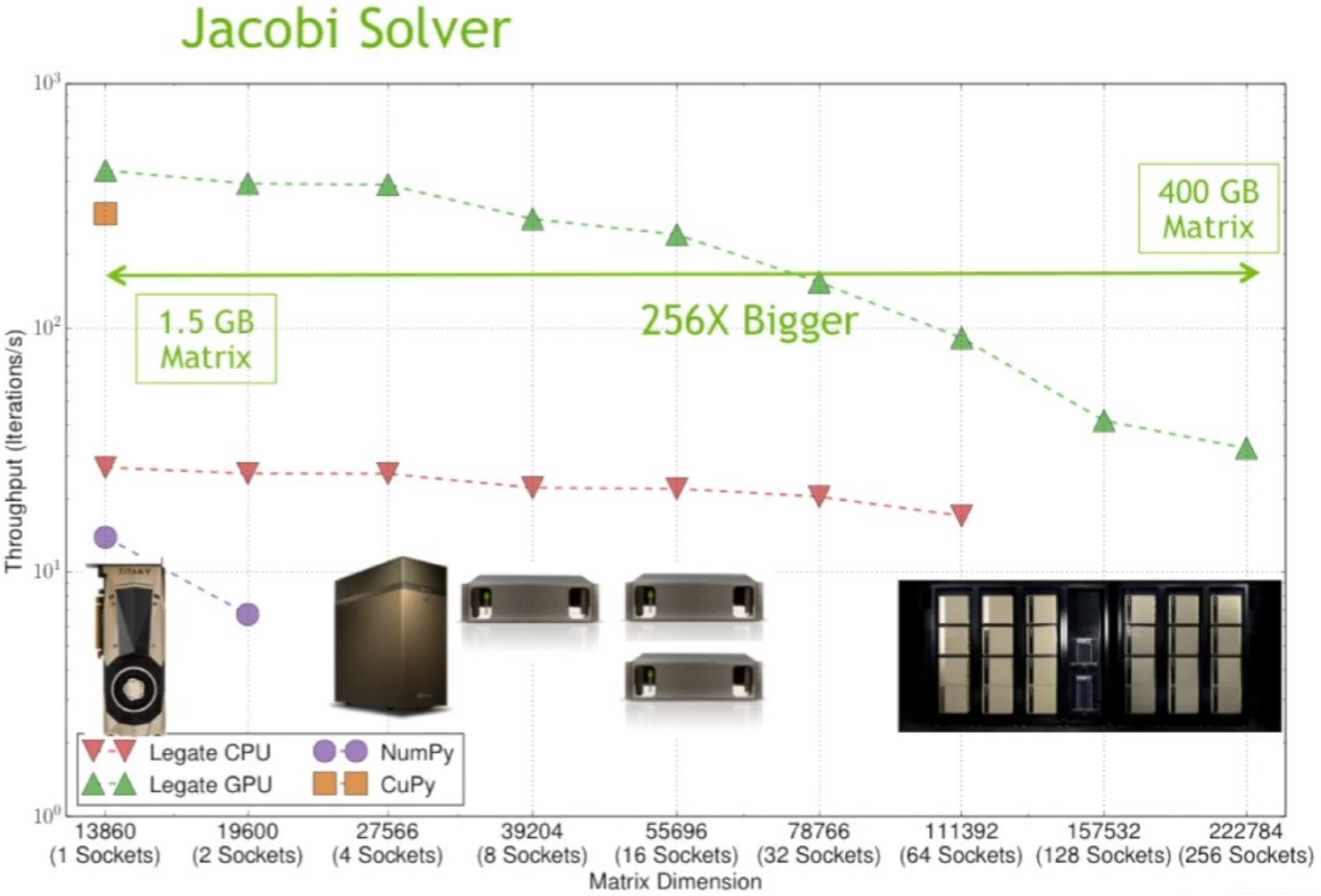
Legion软件栈详细介绍 – Libraries/Apps

- Legate + cuNumeric



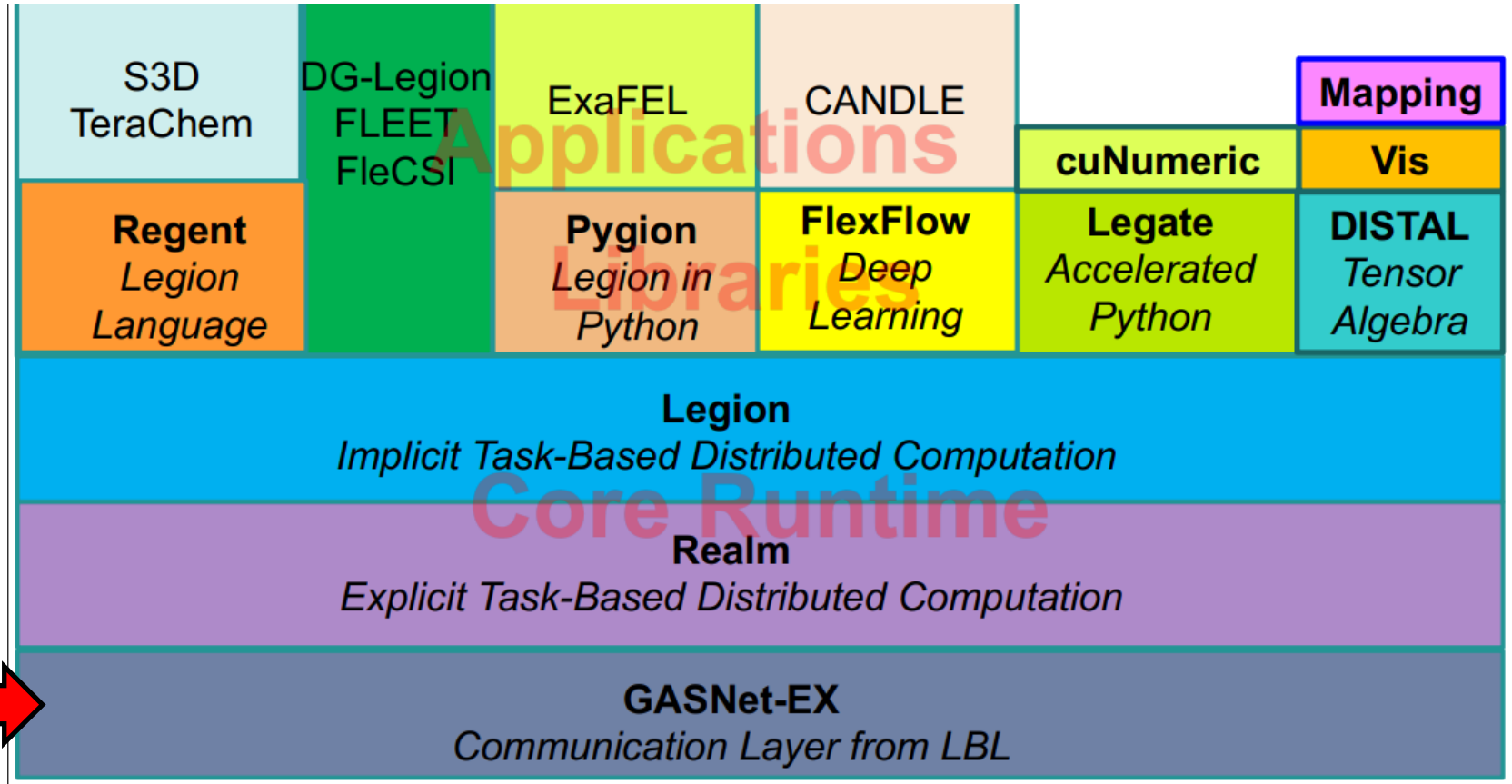
Legion软件栈详细介绍 – Libraries/Apps

- Legate + cuNumeric 性能



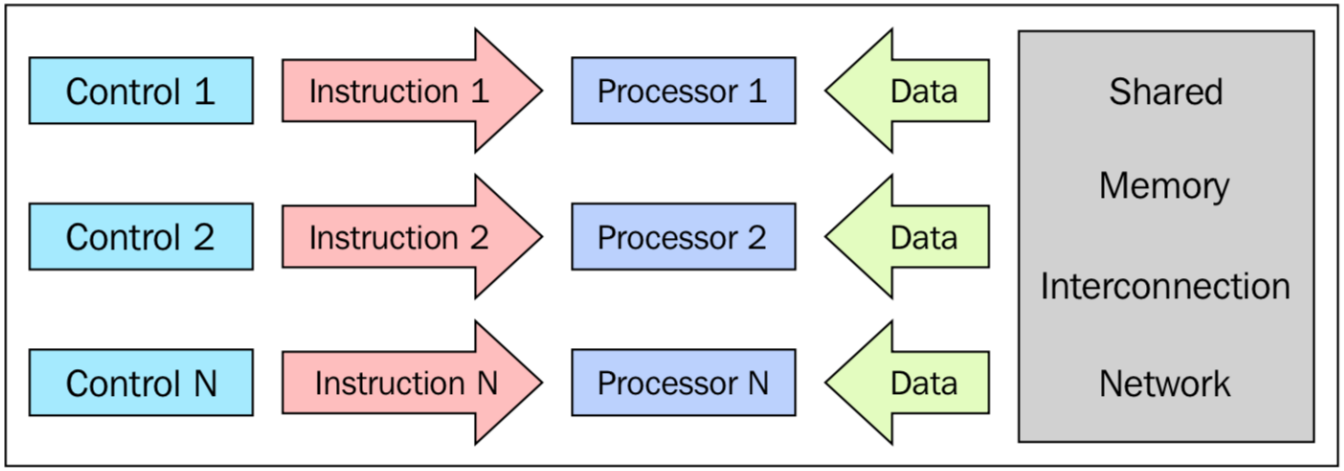
Legion软件栈详细介绍 – 分布式通信库

- Legion的底层分布式通信库
  - 支持GASNet、MPI、CUDA等
  - **支撑Legion从单节点推广到大规模多节点分布式系统！**



Legion软件栈详细介绍 – 分布式通信库

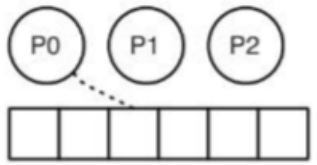
- 异构并行计算系统的内存架构：MIMD架构
  - 每一个处理器都有自己的控制单元和局部内存
  - 通过线程或进程层面的并行来实现处理器的异步工作



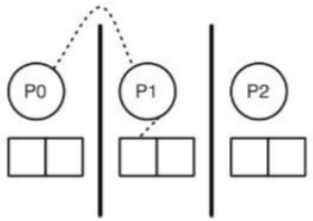
- 异构并行计算系统中的内存管理的挑战
  - **系统中存在多种类型的内存。**
  - **高性能的并行程序的编写复杂。**程序员需要思考如何分配一份数据到多个分区内存中。

Legion软件栈详细介绍 – 分布式通信库

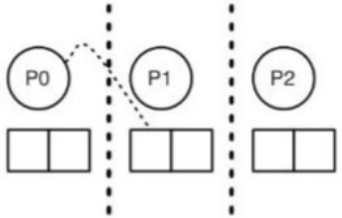
- 分布式并行计算的内存管理模型



  - **共享内存模型**
    - 各个处理器对内存中的数据和指令拥有平等的访问成本和访问权限。
    - 在内存中构建数据结构并在子进程间通过引用直接访问该数据结构。
  - **分布式内存模型**



    - 每个处理器都有自己专属的内存，其他处理器都不能访问。
    - 必须在每个局部内存保存共享数据的副本。一个处理器会向其他处理器发送含有共享数据的消息从而创建数据副本。
  - **分区全局地址空间模型 (Partitioned Global Address Space)**



    - 从消息传递模型继承了对于不同分区的访问成本和访问权限。
    - 从共享内存模型继承了数据在概念上被多个处理单元共有。

Legion软件栈详细介绍 – 分布式通信库

- MPI (Message Passing Interface)
  - 1994年开始维护，主流的并行计算编程模型，一个标准和可移植的通信接口。
  - **编程模型**：提供的是消息传递编程模型。在 MPI 中，进程之间通过消息进行通信，每个进程维护自己的地址空间。
  - **通信模型**：通过显式发送和接收消息实现的，消息的发送和接收是点对点的。
  - **内存模型**：进程拥有自己的私有地址空间，不能直接访问其他进程的地址空间。
- GASNet (Global Address Space Networking)
  - 全局地址空间网络，2002年开始维护，**Legion的分布式通信默认库**
  - **编程模型**：采用的是全局地址空间(Global Address Space)编程模型。
  - **通信模型**：通过远程存储访问(Remote Memory Access，RMA)模型，允许一个处理器直接读写另一个处理器的内存。
  - **内存模型**：提供了全局地址空间，允许程序员使用分布式内存共享数据。

大纲

1. 背景
2. Legion软件栈层级
3. Legion软件栈详细介绍
4. **总结、未来研究方向和现有问题**

总结

- 优点 (针对整个生态)：
  - Legion提供了较为通用的抽象，能够支持大部分多元应用的计算工作流，促进了整个软件体系的发展
  - Legion的“高内聚，低耦合”软件设计原则，搭建了坚实的底层基础，推动了论文的前进
- 缺点：
  - 接口的设计上不够简洁，编写Legion程序时会感觉比较的啰嗦，使得要接入多元应用中现有的主流软件生态时中可能需要大量的改写与包装
  - 整体框架是较为重的，大量的Legion团队自己造轮子的组件，对现有的优质软件生态结合程度有待提高。

未来工作方向和现有问题

未来工作方向1：Legion软件栈在新一代超算的迁移

- **现有挑战**：超算的体系结构差异巨大，编译运行软件栈十分复杂，难以适配
- **方向**：利用Legion和Realm提供的抽象设计，利用超算的底层软件栈实现适配

未来工作方向2：基于Legion和Realm针对新的多元任务设计接口

- **现有挑战**：上层多元应用的需求宽泛，并行业务的层次不一致
- **方向**：针对新的多元应用进行适配 (一个良好的适配≈一篇论文?)

未来工作方向3：针对特定的任务实现高效的Mapping算法

- **现有挑战**：上层应用多元，最优Mapping策略不同。启发式方案存在瓶颈。搜索方案搜索空间过大，速度较慢。
- **方向**：从算法角度设计更优秀的Mapping策略 (结合机器学习, 理论保证算法?)

未来工作方向和现有问题

未来工作方向4：针对Legion动态生成的任务/数据依赖预测分析

- **现有挑战**：动态运行时系统需要在线实时处理任务/数据的依赖，Legion依赖分析方案应该存在并行优化空间。
- **方向**：处理动态生成的任务/数据时，如果可以建立模型进行预测分析，可能可以带来巨大的优化。

未来工作方向5：Legion的分布式通信优化

- **现有挑战**：GASNet较为陈旧，难以迁移。Legion缺少这部分的优化。
- **方向**：测量新颖的分布式通信框架，迁移、架构适配并进行优化。

中山大學
SUN YAT-SEN UNIVERSITY

谢 谢 各 位 老 师 指 正

中山大学计算机学院

汇报人：肖霖畅