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

内容：肖霖畅 杨承润 杨翼飞 伍睿智 介琛

大纲

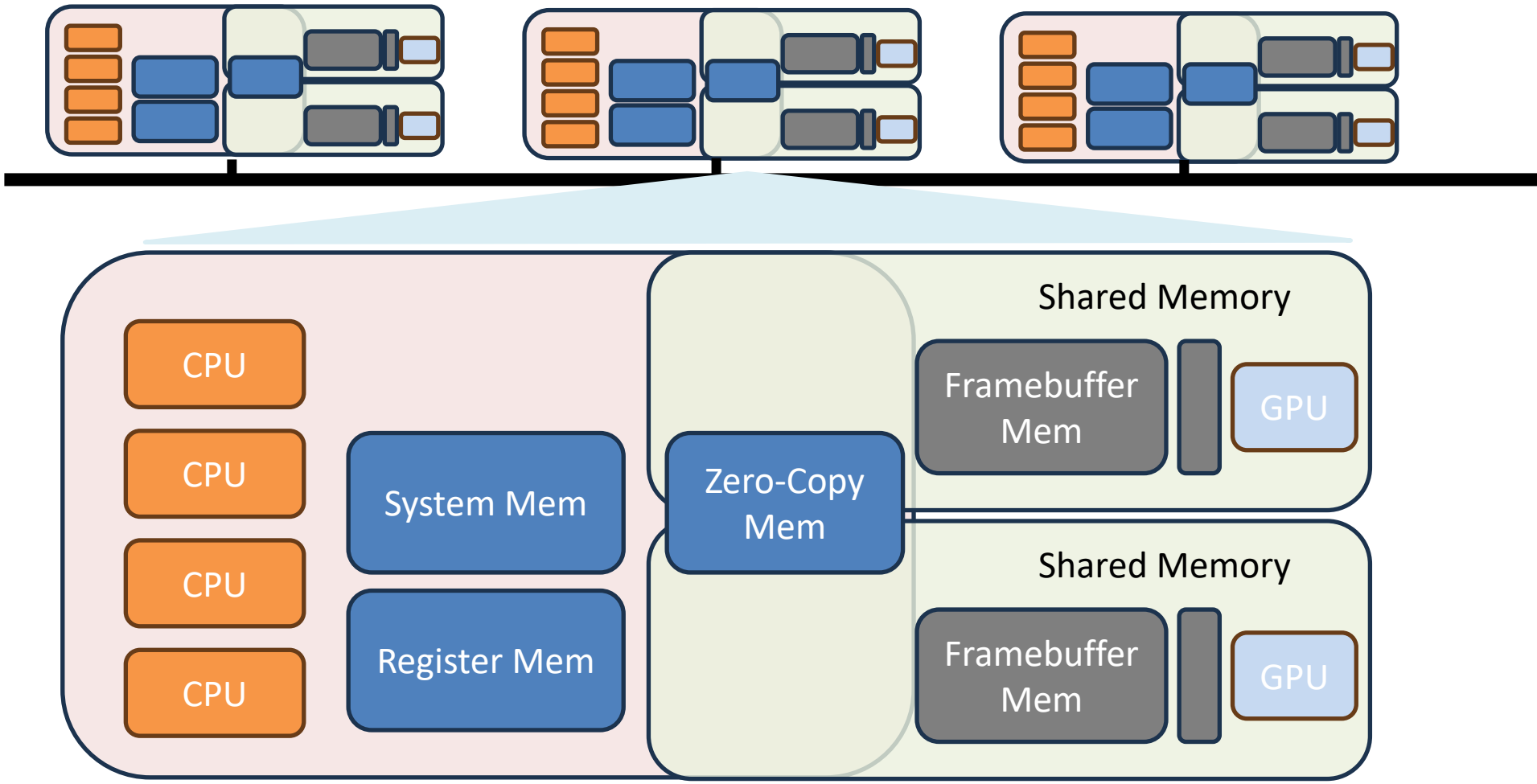
1. **背景**
2. Legion软件栈层级
3. Legion软件栈详细介绍(自底向上)
4. 总结、未来研究方向和现有问题

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

- 复杂的分布式内存层级



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深度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。

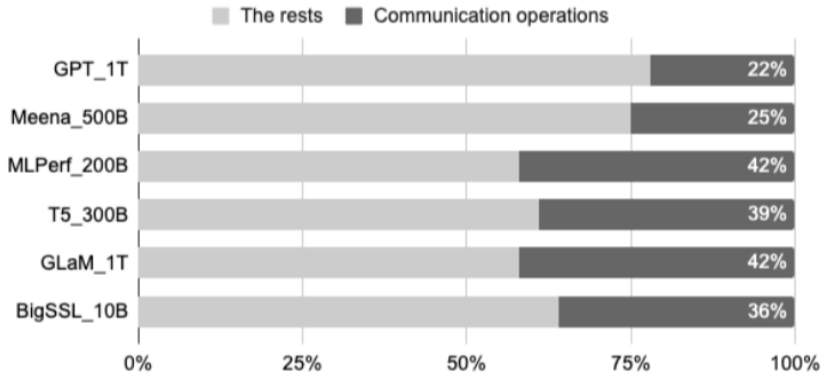
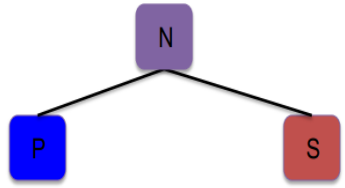
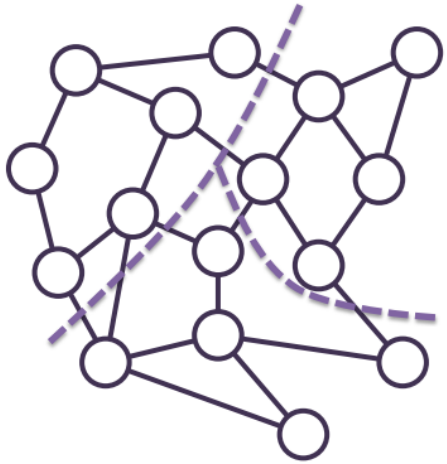


Figure 1: Training step time breakdown of large models. The models run on 128 - 2048 TPU chips depending on the model size. Details on the large models can be found in Section 6.





[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!
  - 设计原则：Data, not compute, matters most.
  - 主要参与单位：
    - Stanford University
    - Nvidia
    - Los Alamos National Laboratory
    - SLAC National Accelerator Laboratory

[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
  - 设计原则：Data, not compute, matters most.
  - 提出了一套较为通用的、适合于并行计算编程的描述数据逻辑结构的抽象
  - 基于任务的编程系统(Task-based programming system)：隐式地推导、提取计算任务之间存在的依赖关系，并增加必要的数据拷贝与移动操作来保证并行的正确性与效率
  - 提供了可拓展的mapping接口，允许用户介入legion runtime的工作流中，来自定义地控制计算任务实际执行时所使用的计算资源与存储资源，以及其他一些具体的执行细节
  - 针对异构的设备提供了统一的抽象与拓展接口

[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

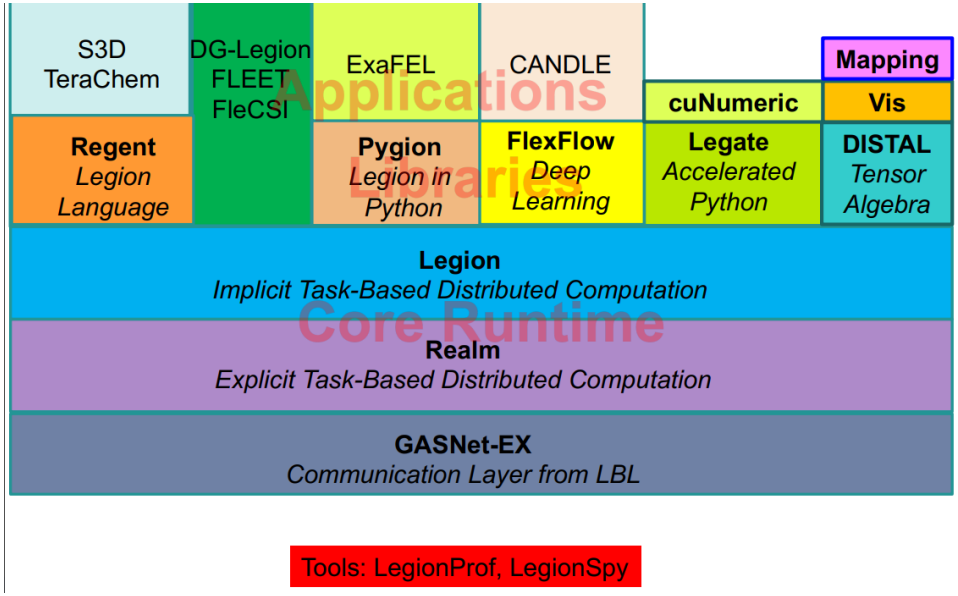


Table of Contents

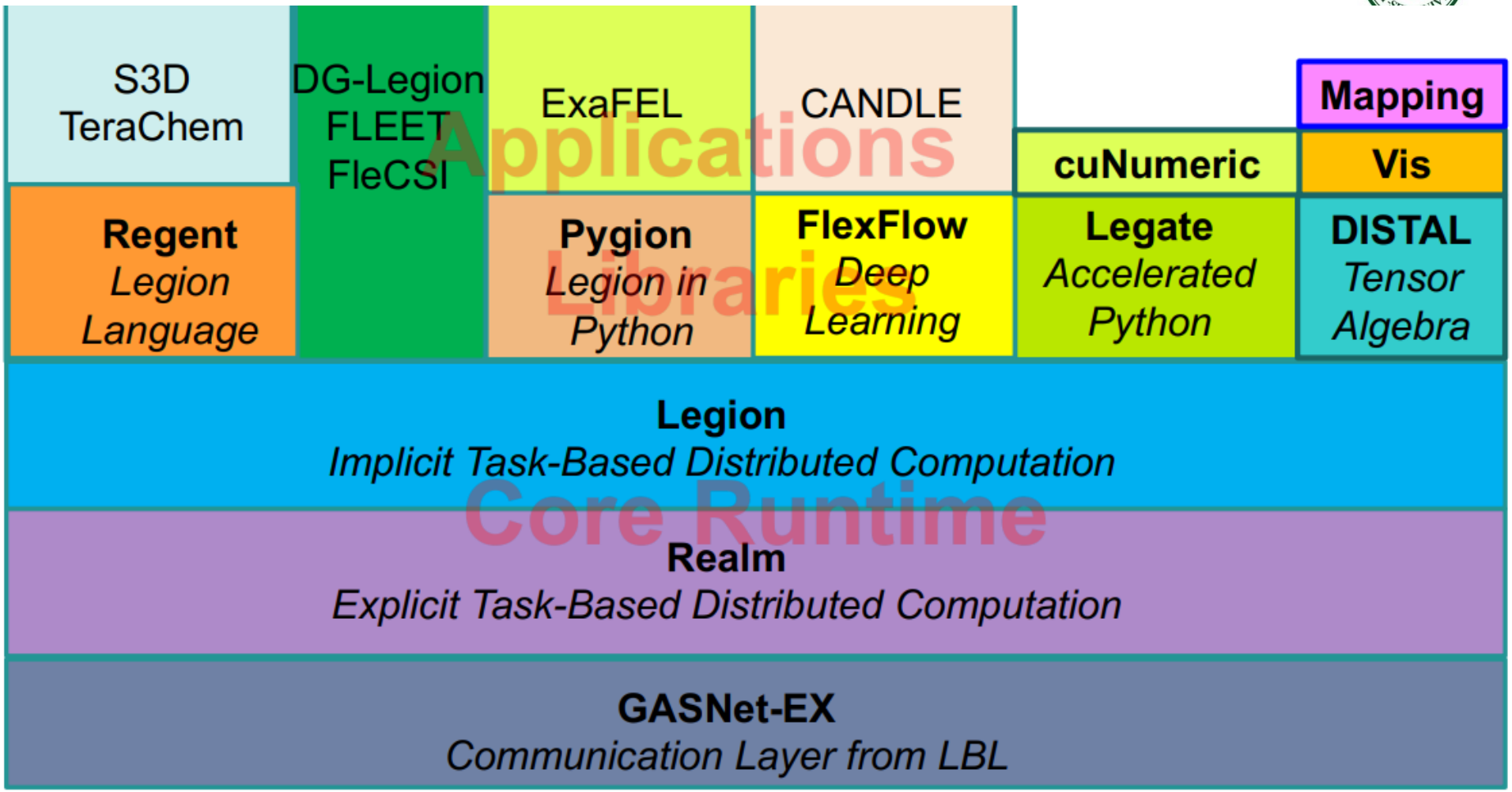
- Legion Runtime:
  - Overview (SC 2012) [PDF]
  - Structure Slicing (SC 2014) [PDF]
  - Tracing (SC 2018) [PDF]
  - Correctness of Dependence Analysis (Correctness 2018) [PDF]
  - Dynamic Control Replication (PPoPP 2021) [PDF]
  - Index Launches (SC 2021) [PDF]
  - Visibility Algorithms (PPoPP 2023) [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(自底向上)
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

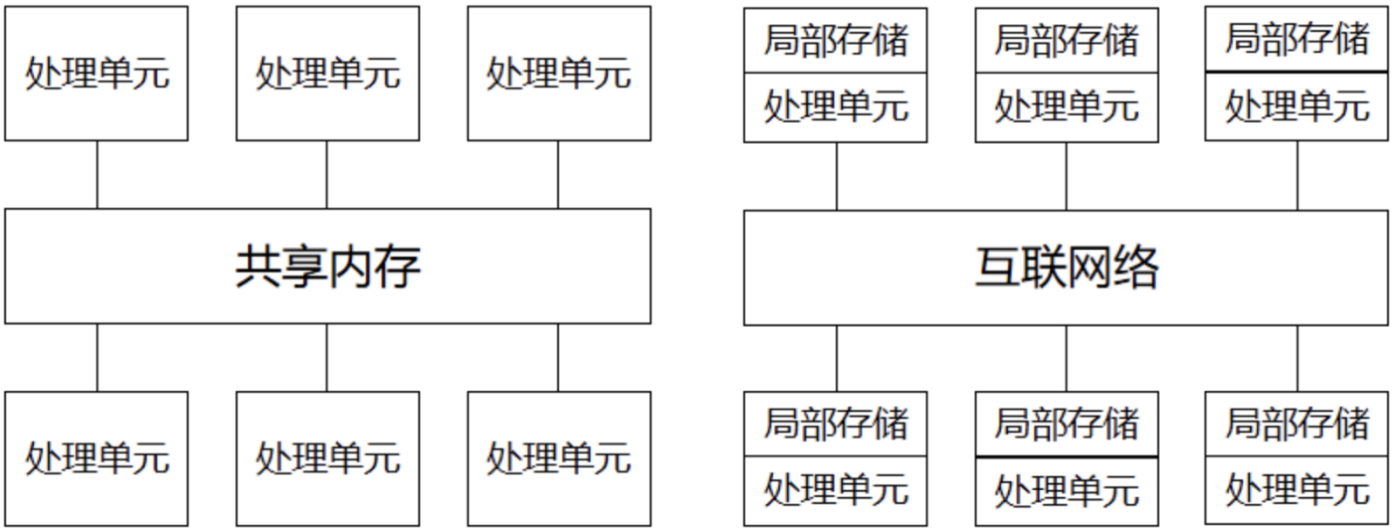
- Core Runtime级
  - **GASNet-EX**: Communication Layer
  - **Realm**: Explicit Task-Based Distributed Computation 显式任务分布式计算
  - **Legion**: Implicit Task-Based Distributed Computation 隐式任务分布式计算
- Libraries级
  - **Regent**: 易用的静态类型领域语言
  - **Pygion**: Python动态类型语言支持
  - **Flexflow**: 自动深度学习任务并行优化底层支持库
  - **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(自底向上)**
4. 总结、未来研究方向和现有问题

Legion软件栈详细介绍 - GASNet-EX

- 并行计算相关背景
  - **共享内存**：共享内存的编程在数据交换和访问上有较大的优势，程序编写起来更加简单。但在扩展性上有较大的瓶颈。
  - **分布式内存**：每个计算单元有单独的内存，计算单元之间的数据访问通过互联网络去传输。这一架构在可移植性和扩展上会强很多，但消息的传递会成为程序设计中的难点。 [通信库和消息传递编程模型和标准成为关键]
  - 两者的结合：**分布式共享内存并行计算机** [现代最常用的体系结构]



Legion软件栈详细介绍 - GASNet-EX

- 并行计算相关背景
  - MPI：分布式内存编程模型的并行程序中，一个标准和可移植的通信接口。
    - 1994年开始维护，主流的并行计算编程模型。
    - **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**：每个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。

Legion软件栈详细介绍 - GASNet-EX

- 并行计算相关背景
  - MPI同步通信

PROCESS A	PROCESS B
`double send_buff[N];`	`double recv_buff[N];`
`(...)`	`(...)`
`MPI_Ssend(send_buff, N, MPI_DOUBLE, 1, TAG, COMM);`	`(...)`
`// Wait until receive is done`	`MPI_Recv(recv_buff, N, MPI_DOUBLE, 0, TAG, COMM, STATUS);`
`// Synchronization barrier`	`// Synchronization barrier`
`(...) // Can modify send_buff`	`(...) // Received data from 0 in recv_buff`

  - MPI异步通信

PROCESS 0	PROCESS 1
`double send_buff[N];` `MPI_Request req;`	`double recv_buff[N];`
`(...)`	`(...)`
`MPI_Isend(send_buff, N, MPI_DOUBLE, 1, TAG, COMM, &req);`	`(...)`
`(...) // Do things, but don't modify send_buff`	`MPI_Recv(recv_buff, N, MPI_DOUBLE, 0, TAG, COMM, STATUS);`
`MPI_Wait(&req, MPI_STATUS_IGNORE);`	`(...) // received data from 0 in recv_buff`
`(...) // Now can modify send_buff`	

Legion软件栈详细介绍 - GASNet-EX

- GASNet
  - 全局地址空间网络，2002年开始维护
  - **编程模型**：采用的是全局地址空间编程模型。GASNet 允许程序员将一个大的、全局的地址空间分配给所有的处理器，从而使得所有处理器都能够直接访问全局内存。
  - **通信模型**：通过远程存储（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。

Legion软件栈详细介绍 - GASNet-EX

- Active Messages in GASNet
  - 所有process都可以操作local_buff
  - 远程调用机制，利用gasnet_AMRequestMedium0(...)向目标process发送少量参数，让目标process启动注册的函数

PROCESS 0

```
double local_buff[N];

void handler(token_t t, void *buff, int size)
{
      memcpy(local_buff, buff, size);
}

int main()
{
      (...) // register function ‚handler‘ with ID
      ‚s_ID‘

      gasnet_AMRequestMedium0(s_ID, local_buff, N);

      // can directly modify local_buff!
}
```

PROCESS 1

```
double local_buff[N];

void handler(token_t t, void *buff, int size)
{
      memcpy(local_buff, buff, size);
}

int main()
{
      (...) // register function ‚handler‘ with ID ‚s_ID‘

      gasnet_AMPoll(); // executes all pending requests

      // got local_buff from process 0!
}
```

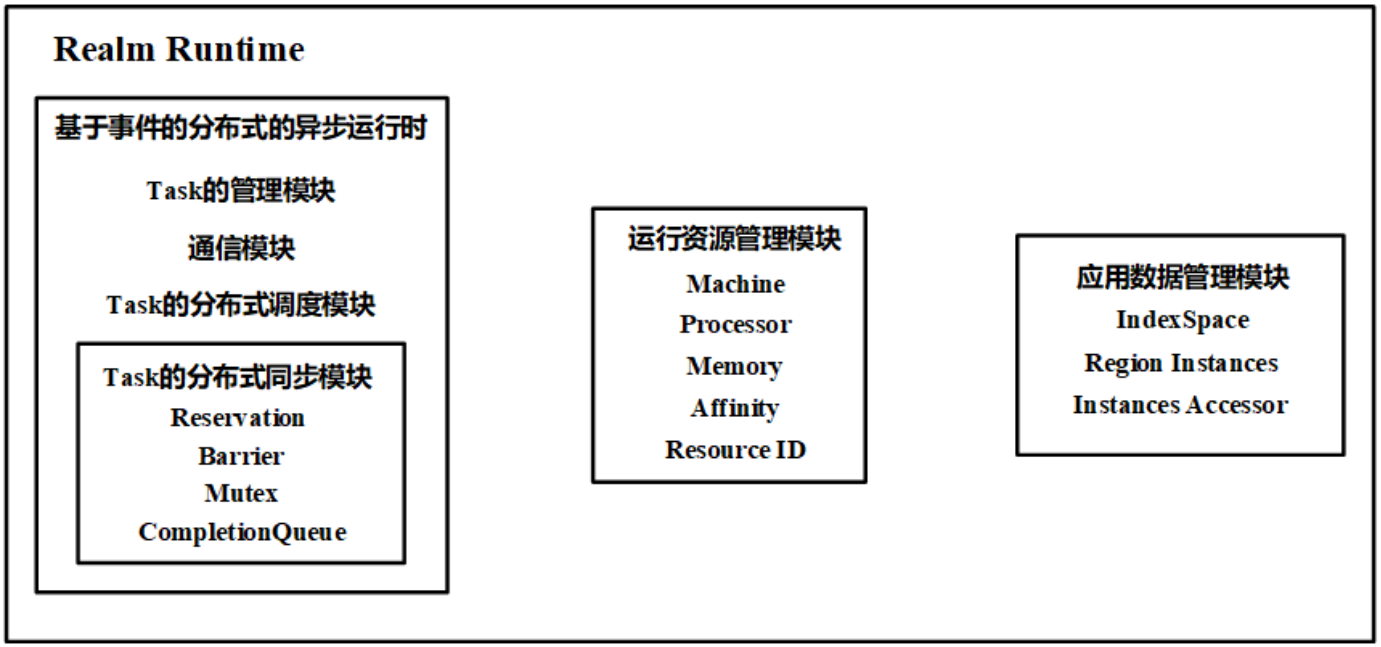
Legion软件栈详细介绍 - GASNet-EX

- MPI和GASNet选型
  - **MPI**: 适用于各种不同类型的计算，包括密集型计算和大规模并行计算。 广泛用于传统的高性能计算（HPC）领域，如科学和工程领域的模拟、计算流体力学等。
  - **GASNet**: 通常用于一些具有分布式内存模型的系统，例如 PGAS（Partitioned Global Address Space）编程模型。主要在一些 PGAS 模型的实现中使用，如 UPC（Unified Parallel C）和 Chapel 等。

- Legion选型原因
  - GASNet使得程序员可以方便地在分布式内存系统中进行数据共享和通信。
  - Legion 的设计目标之一是在大规模并行计算环境中实现**可扩展性**。GASNet 被设计为适应大规模系统的通信需求，具有良好的可扩展性。

Legion软件栈详细介绍 - Realm

- Realm: Explicit Task-Based Distributed Computation
  - 基于事件的的分布式异步运行时：类似Ray
  - 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)



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。具体地，Machine model中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

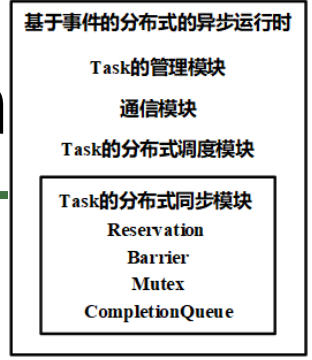
运行资源管理模块
Machine
Processor
Memory
Affinity
Resource ID

- 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，Event是realm编程模型的一个核心，用于描述不同操作之间的依赖关系
    - 所有的realm的操作都由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(类似future的编程模型，但不包含数据的传递)
    - 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
  - 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

Legion软件栈详细介绍 - Realm

- Realm 应用数据管理模块
  - 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: Implicit Task-Based Distributed Computation
  - 基于Realm的提供更多功能的Runtime库

Realm的模块:

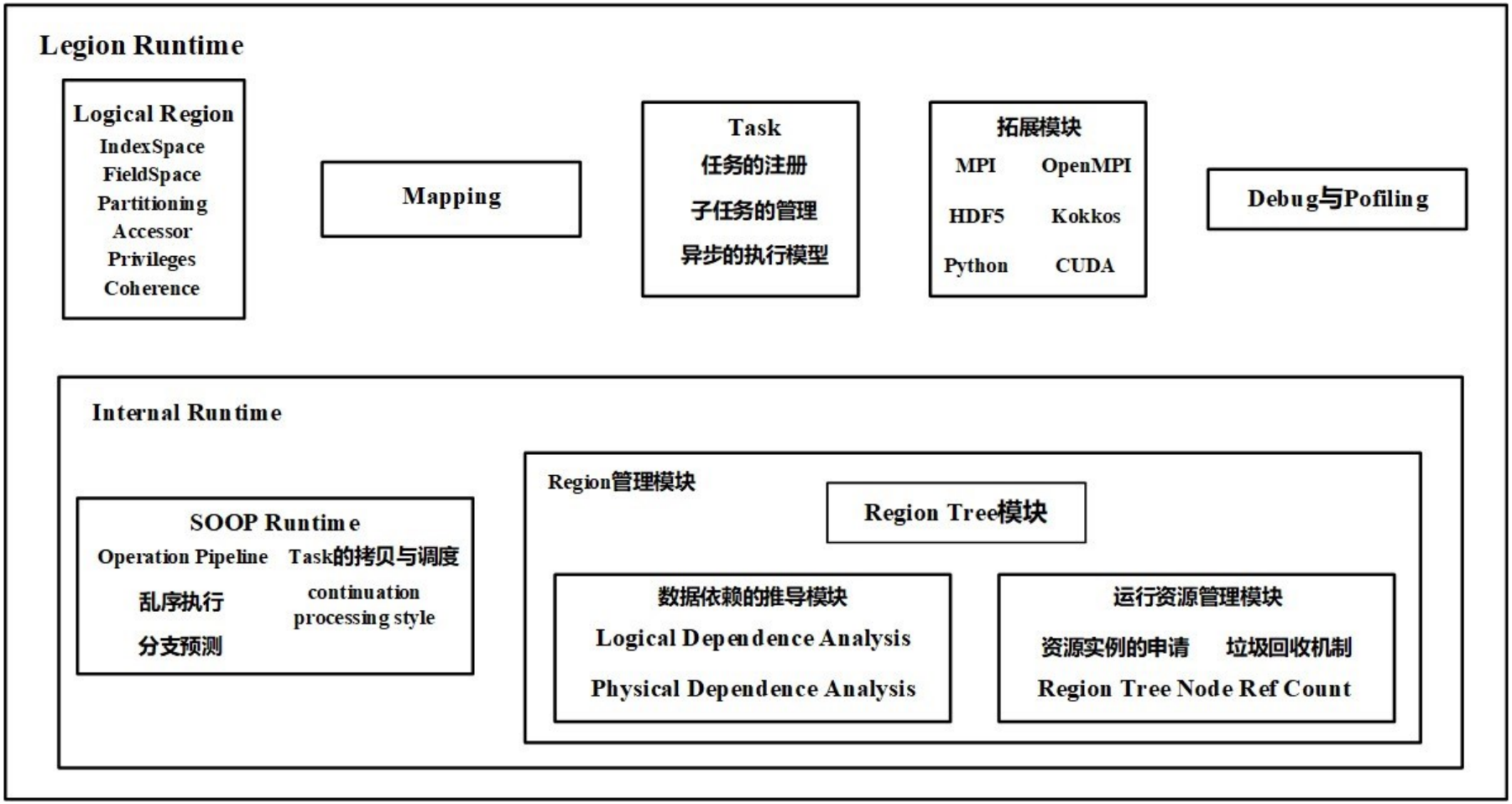
- events
- tasks/processors/scheduling
- index spaces/sparsity maps
- instance/accessors
- memories/allocation
- copies
- reservations
- dependent partitioning
- cuda
- networking
- logging/profiling
- start-up/tear-donw/modules

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

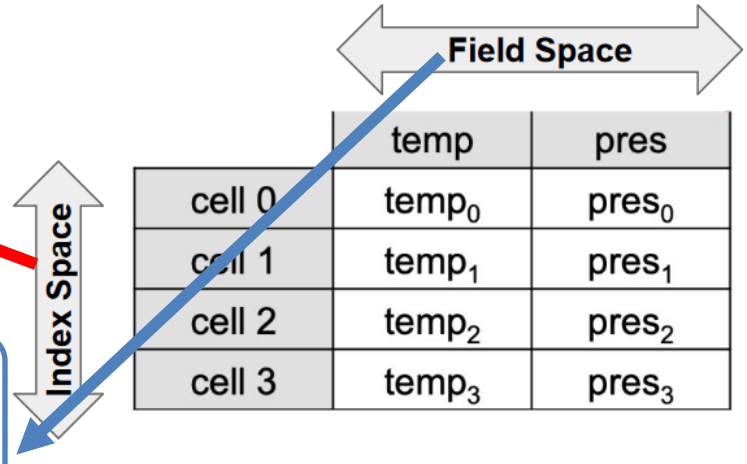


Legion软件栈详细介绍 - Legion

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

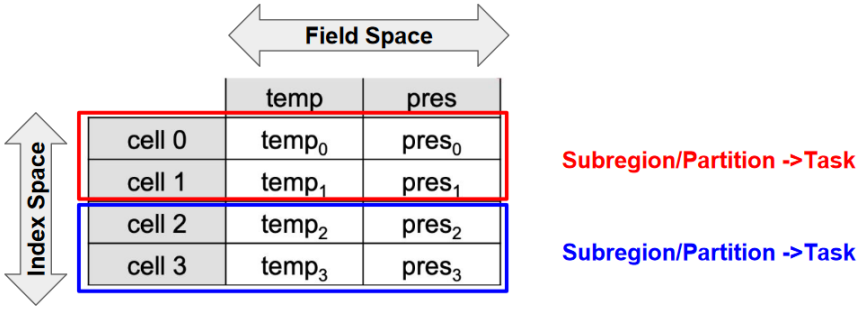
```
enum FieldIDs {
  FID_FIELD_A,
  FID_FIELD_B,
};

void top_task(const Task *task, const std::vector<PhysicalRegion> &regions,
              Context ctx, Runtime *runtime) {

  // 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);

  // 利用索引空间与字段空间来创建逻辑区域
  LogicalRegion untyped_lr =
      runtime->create_logical_region(ctx, index: untyped_is, fields: fs);
  LogicalRegionT<1> typed_lr =
      runtime->create_logical_region(ctx, index: typed_is, fields: fs);

  // 最后进行资源的销毁
  runtime->destroy_logical_region(ctx, handle: untyped_lr);
  runtime->destroy_logical_region(ctx, handle: typed_lr);
  runtime->destroy_field_space(ctx, handle: fs);
  runtime->destroy_index_space(ctx, handle: untyped_is);
  runtime->destroy_index_space(ctx, handle: typed_is);
}
```



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 vs. Physical Region
    - Logical Region：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，可以设置Privilege和Coherence
    - Physical Region：物理上的数据存储，可以细粒度赋值和修改
    - 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、reduce
  - 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

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(privileges)
  - 实际编码时，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
  - 批量launch：支持批量地创建task

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的注册

```
// NOTE 描述顶层任务id
enum TaskID {
    HELLO_WORLD_ID,
};

// NOTE Legion描述任务的统一函数签名,形参需要保持一致,但允许返回值不同
void hello_world_task(const Task *task,
                      const std::vector<PhysicalRegion> &regions,
                      Context ctx, Runtime *runtime) {
    // A task runs just like a normal C++ function.
    printf(format: "Hello World!\n");
}

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

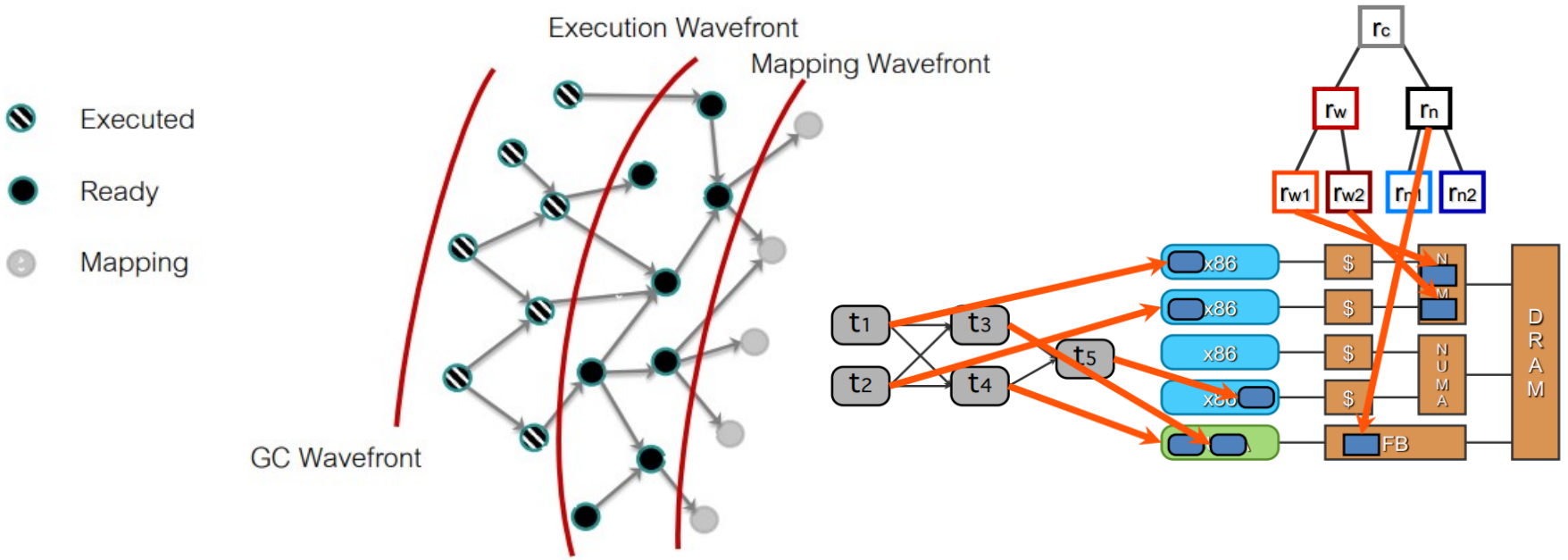
- Legion中表示计算工作流的Task
  - 例子：子任务的派生和异步执行

```
void top_level_task(const Task *task,
                    const std::vector<PhysicalRegion> &regions,
                    Context ctx, Runtime *runtime) {
  int num_fibonacci = 15;
  const InputArgs &command_args = Runtime::get_input_args();
  for (int i = 1; i < command_args.argc; i++) {
    // Skip any legion runtime configuration parameters
    if (command_args.argv[i][0] == '-') {
      i++;
      continue;
    }

    num_fibonacci = atoi(nptr: command_args.argv[i]);
    assert(num_fibonacci >= 0);
    break;
  }
  printf(format: "Computing the first %d Fibonacci numbers...\n", num_fibonacci);

  std::vector<Future> fib_results;

  Future fib_start_time = runtime->get_current_time(ctx);
  std::vector<Future> fib_finish_times;

  for (int i = 0; i < num_fibonacci; i++) {
    TaskLauncher launcher(tid: FIBONACCI_TASK_ID, arg: TaskArgument(arg: &i, argsize: sizeof(i)));
    fib_results.push_back(x: runtime->execute_task(ctx, launcher));
    fib_finish_times.push_back(x: runtime->get_current_time(ctx, precondition: fib_results.back()));
  }

  for (int i = 0; i < num_fibonacci; i++) {
    int result = fib_results[i].get_result<int>();
    double elapsed = (fib_finish_times[i].get_result<double>() -
                      fib_start_time.get_result<double>());
    printf(format: "Fibonacci(%d) = %d (elapsed = %.9f s)\n", i, result, elapsed);
  }

  fib_results.clear();
}
```

```
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>();
}

int sum_task(const Task *task,
             const std::vector<PhysicalRegion> &regions,
             Context ctx, Runtime *runtime) {
  assert(task->futures.size() == 2);
  Future f1 = task->futures[0];
  int r1 = f1.get_result<int>();
  Future f2 = task->futures[1];
  int r2 = f2.get_result<int>();

  return (r1 + r2);
}
```

Legion软件栈详细介绍 - Legion

- Legion中的Mapping
  - Legion需要决定：Task在哪里执行；Region在哪里放置
  - 由用户自己决定Mapping的具体策略
  - After tasks are mapped, they are distributed to their target nodes.



Legion软件栈详细介绍 - Libraries

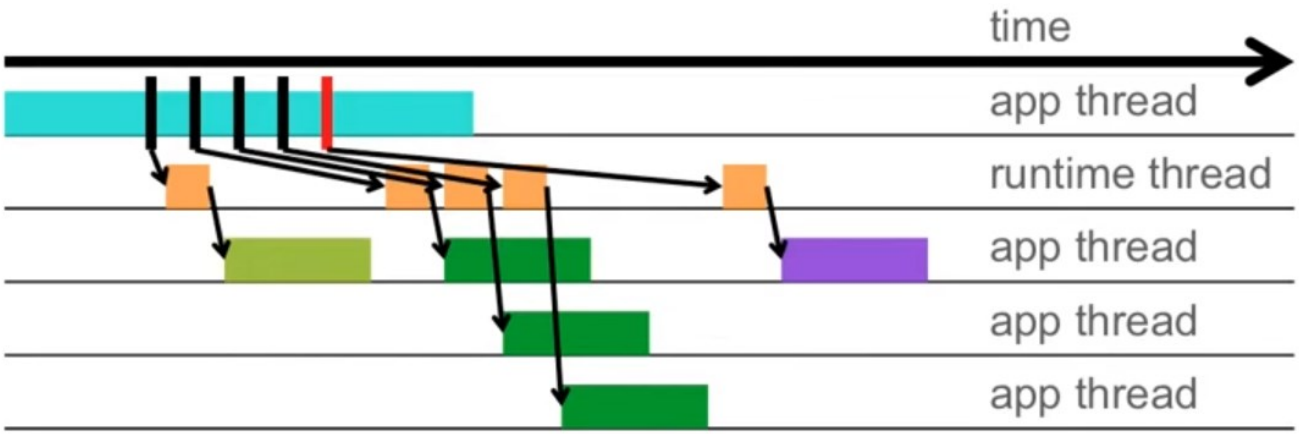
- Regent
  - Regent简化了Legion的编程模型
  - Regent程序的性能和精细调整的Legion程序具有相同的性能

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

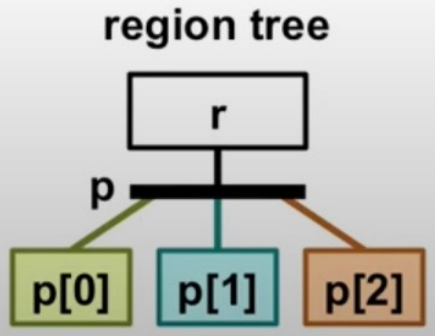
```
runtime->unmap_region(ctx, physical_r);
TaskLauncher launcher_A(TASK_A, TaskArgument());
launcher_A.add_region_requirement(
  RegionRequirement(r, READ_WRITE, EXCLUSIVE, r));
launcher_A.add_field(0, FIELD_X);
launcher_A.add_field(0, FIELD_Y);
runtime->execute_task(ctx, launcher_A);
Domain domain = 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);
runtime->execute_task(ctx, launcher_C);
runtime->map_region(ctx, physical_r);
```

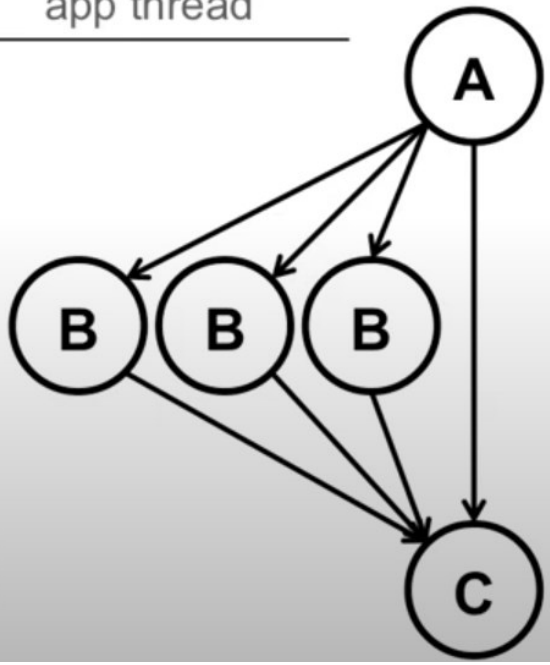
Legion软件栈详细介绍 - Libraries

- Regent例子




```
var r = region(…)
var p = partition(disjoint, r, …)
A(r)
for i = 0, 3 do
  B(p[i])
end
C(r)
```





Legion软件栈详细介绍 - Libraries

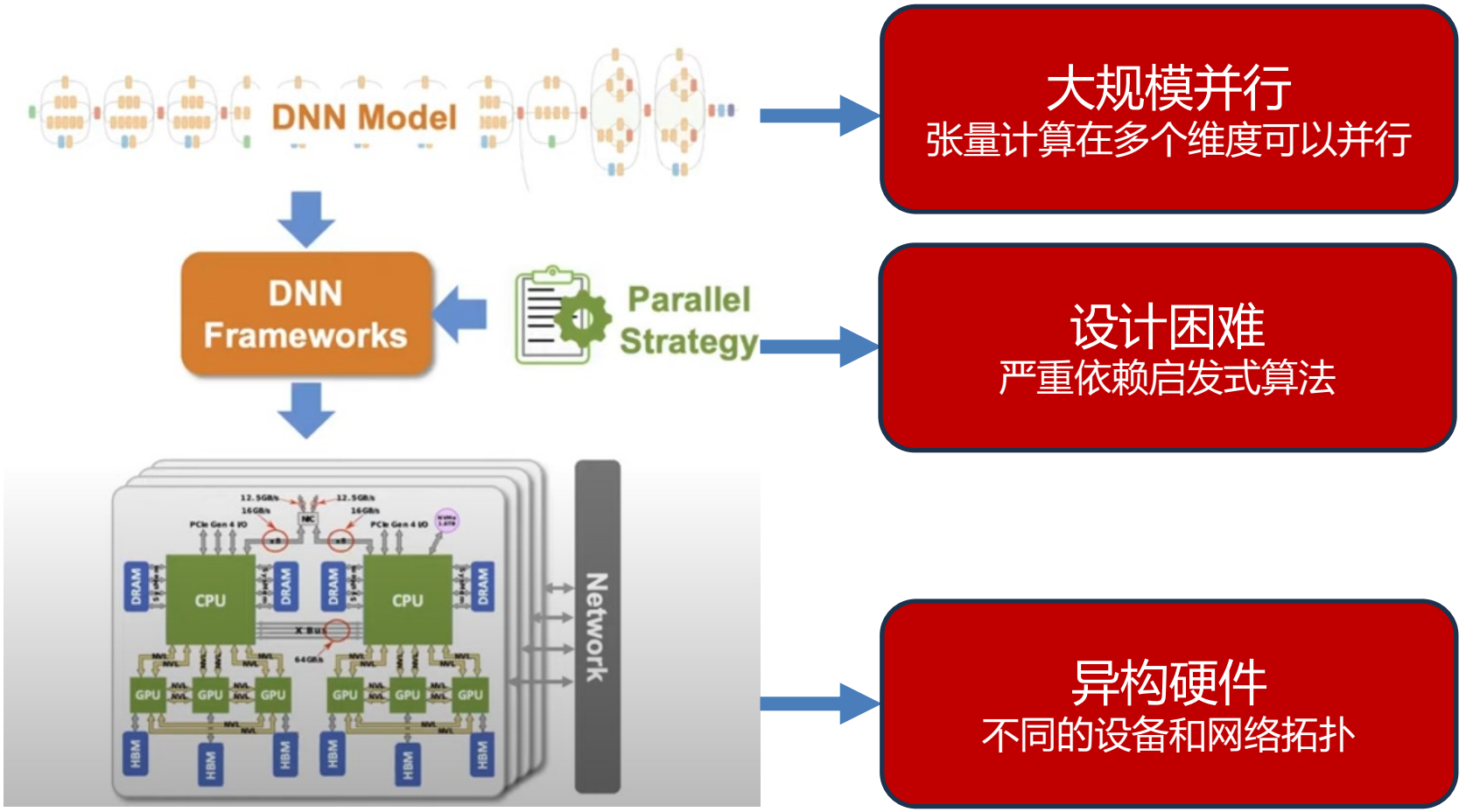
- Pygion产生背景
  - Legion C++：复杂的API，容易出错
  - Regent：简化API，但仍然是静态类型语言

```
@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)
```

```
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);
```

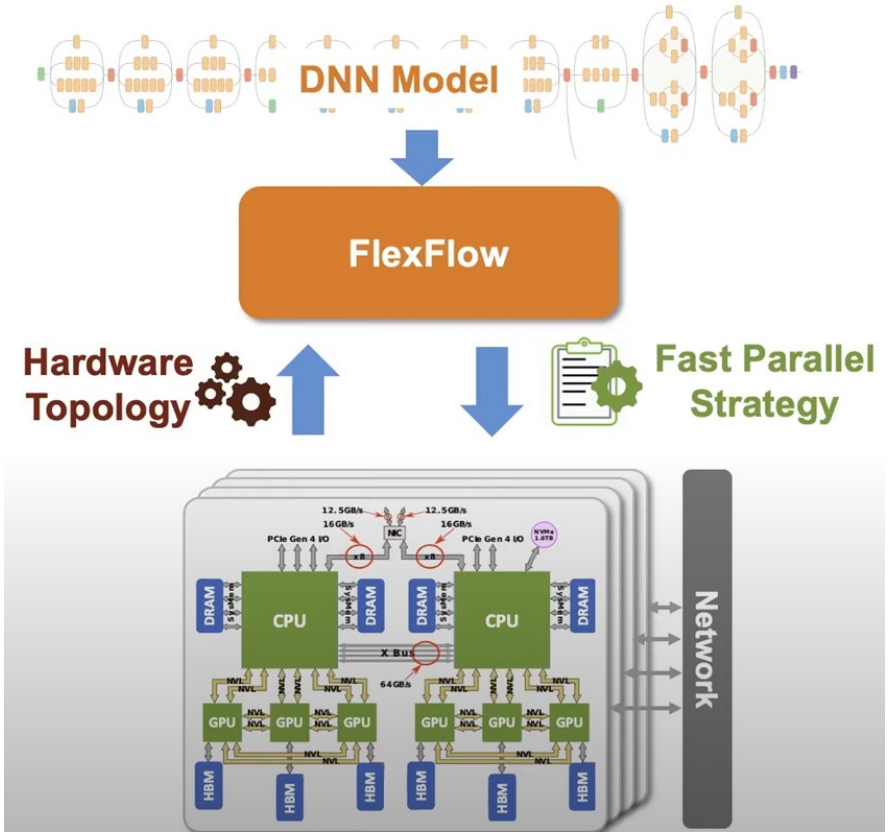
Legion软件栈详细介绍 - Libraries

- Flexflow: Automatically Optimizing DNN Parallelization
  - 深度学习并行优化的挑战



Legion软件栈详细介绍 - Libraries

- Flexflow: Automatically Optimizing DNN Parallelization
  - Flexflow的贡献



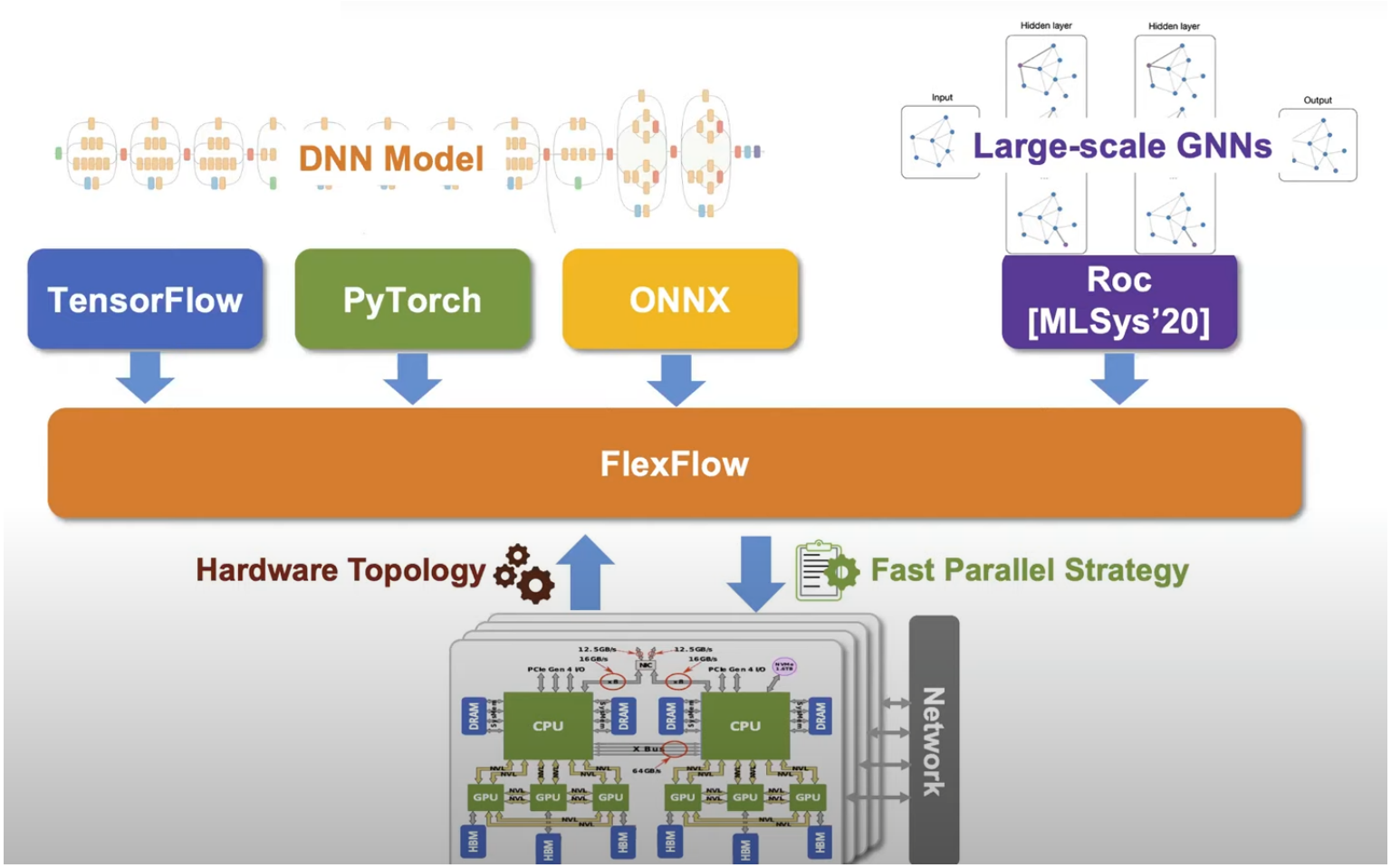
更好的性能
相比当时的手动调整策略获得了10x速度提升

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

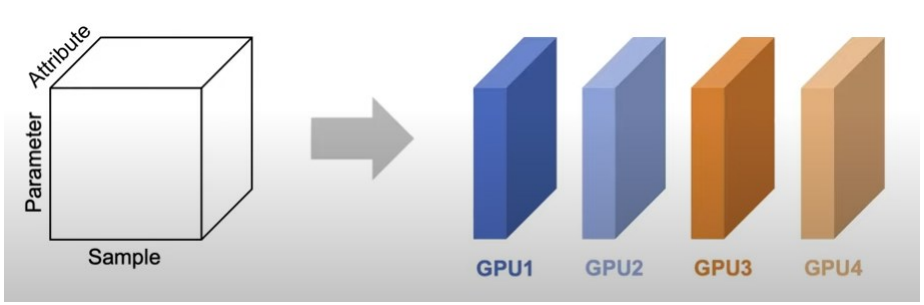
Legion软件栈详细介绍 - Libraries

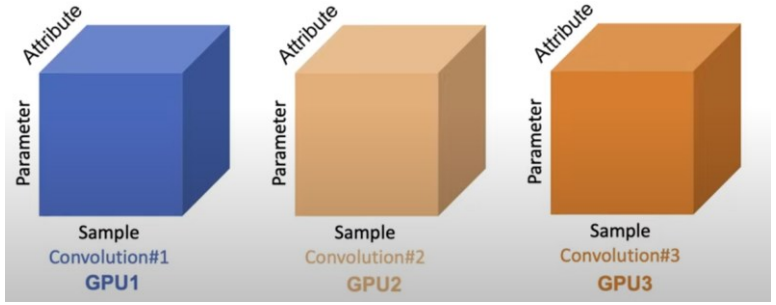
- Flexflow: Automatically Optimizing DNN Parallelization
  - Flexflow软件生态

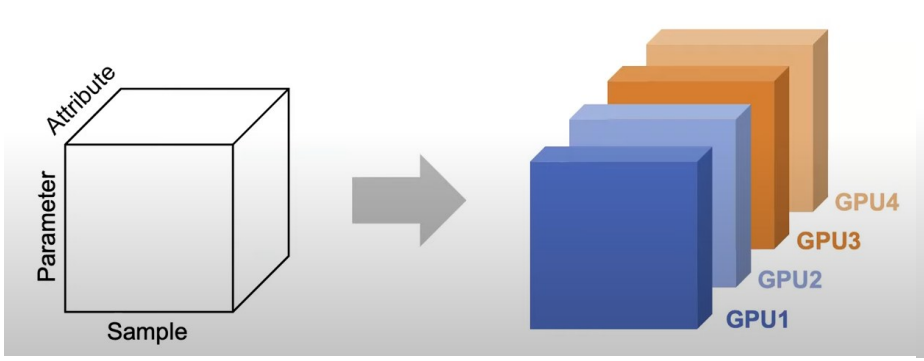


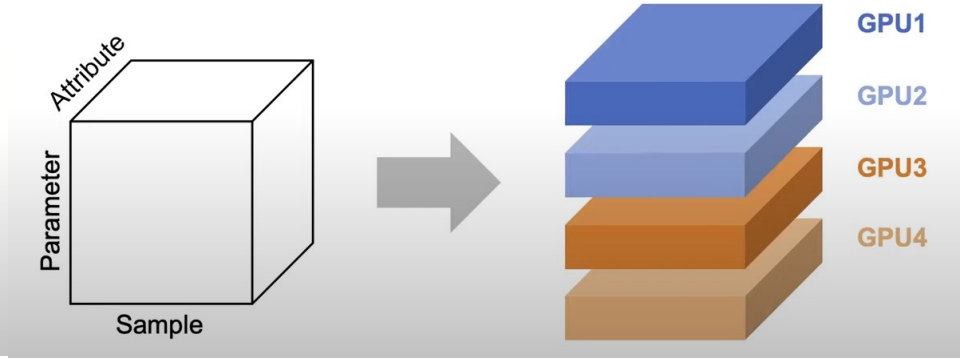
Legion软件栈详细介绍 - Libraries

- Flexflow 决策搜索空间： SOAP搜索空间
  - Sample： 划分训练样本（数据并行）
  - Operator：划分ML算子 （模型并行）
  - Attributes：在一个样本中划分属性（比如pixels）
  - Parameters：在一个ML算子中划分参数



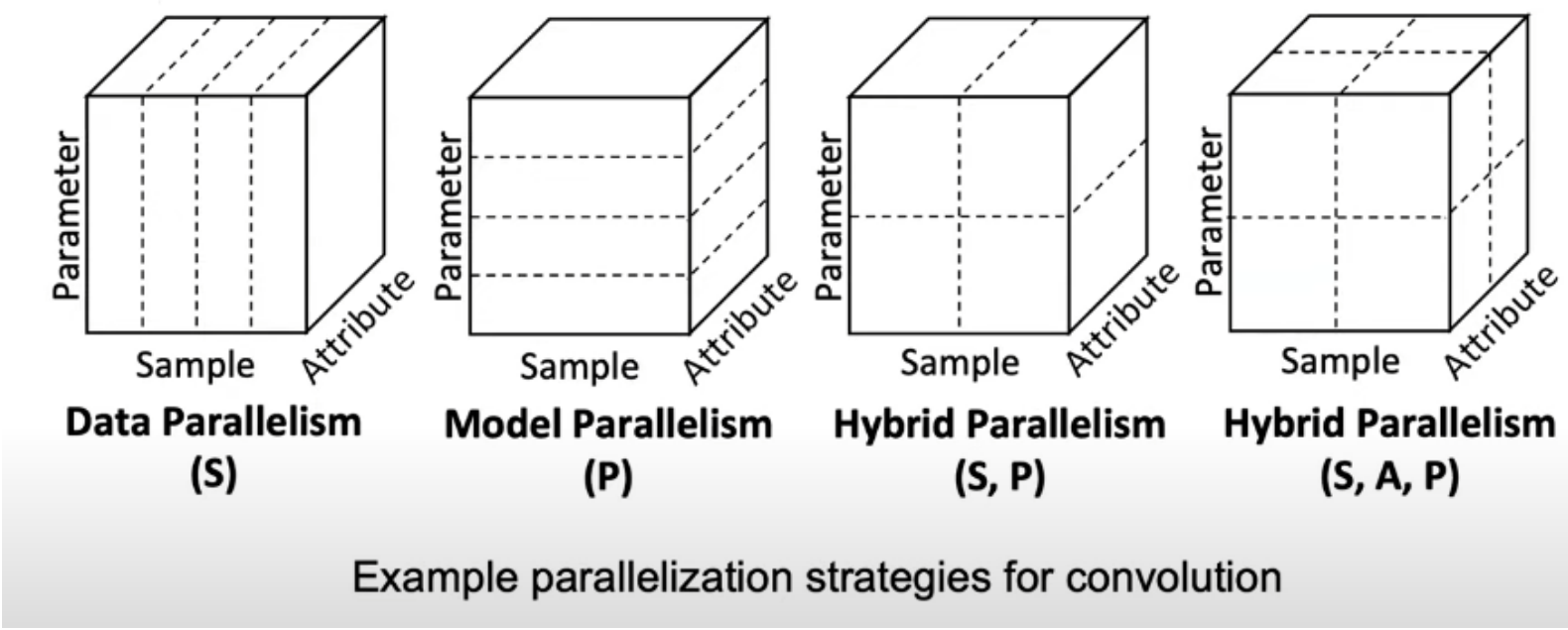






Legion软件栈详细介绍 - Libraries

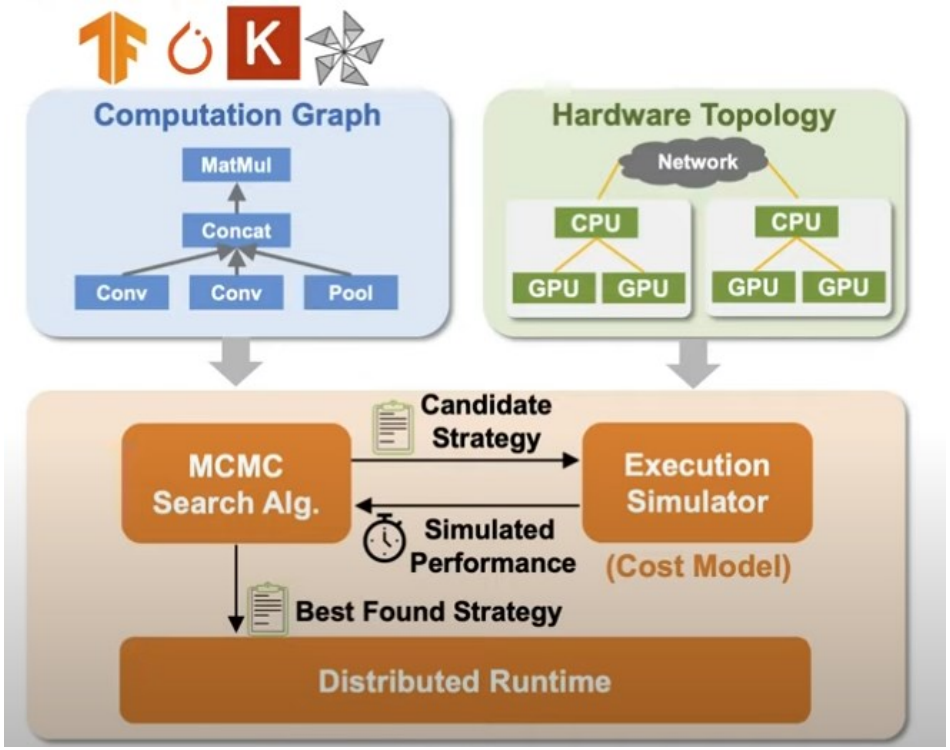
- Flexflow SOAP 混合并行



- SOAP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

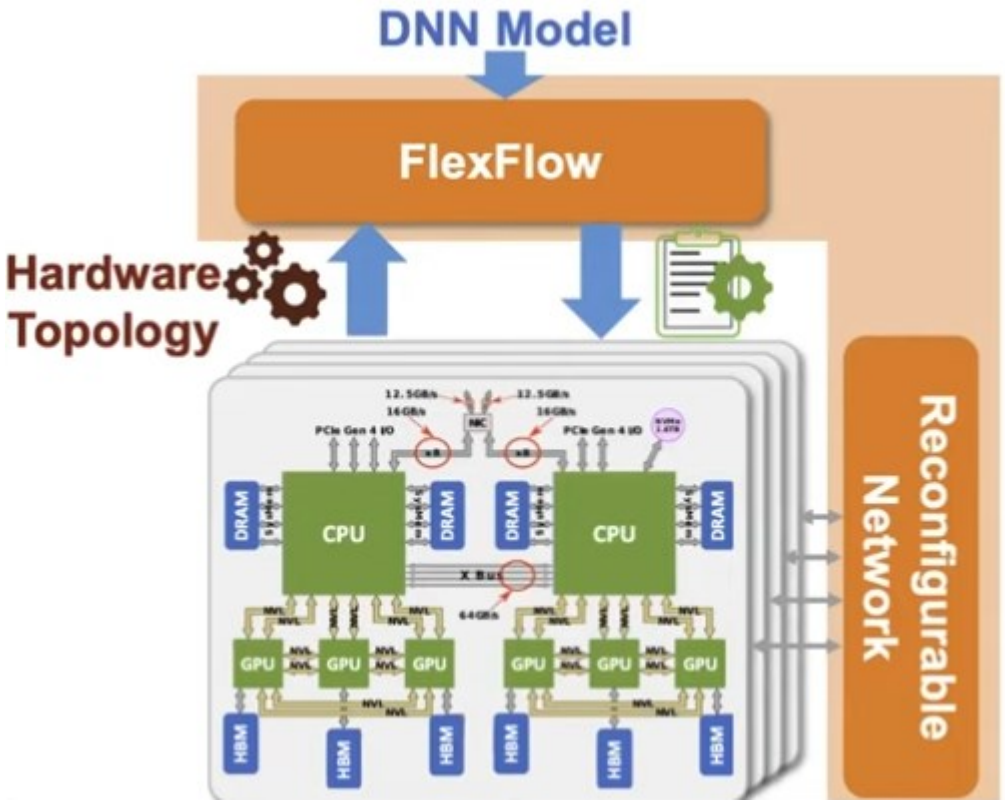
Legion软件栈详细介绍 - Libraries

- Flexflow 搜索优化方案
  - 依赖的观察1：ML算子的性能是高度可预测的
  - 依赖的观察2：ML模型只包含少量的不同类型的算子
  - 推论：性能 = 每个类型算子的性能 + 通信带宽



Legion软件栈详细介绍 - Libraries

- Flexflow 未来优化方向
  - 联合优化并行策略和网络拓扑



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

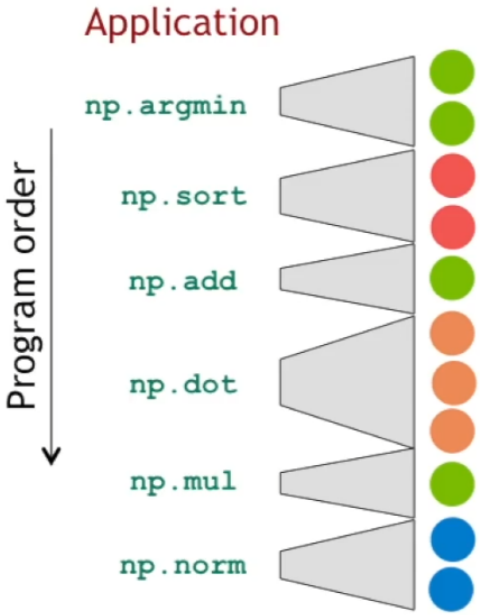
```python
import numpy as np

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
```

```python
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

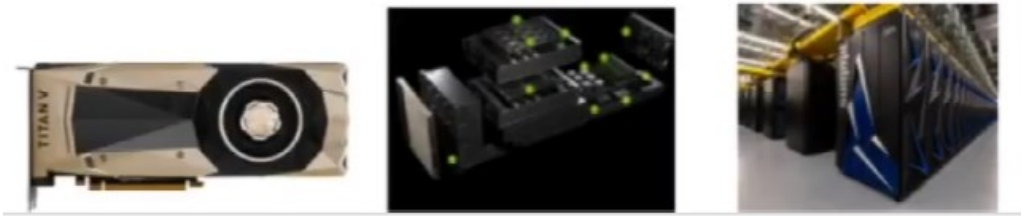
- Legate + cuNumeric

cuNumeric将API调用转化为Task



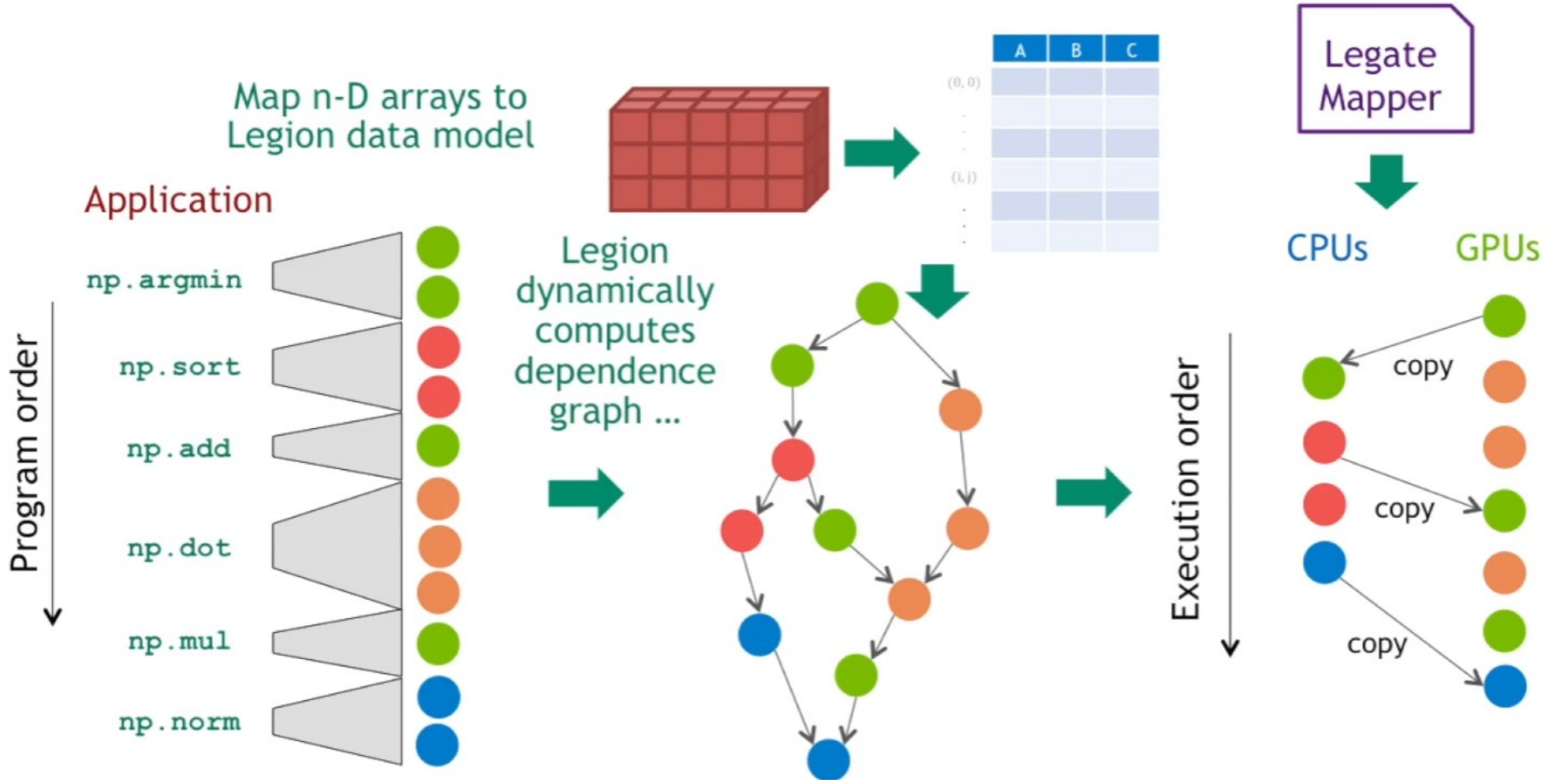
cuNumeric提供更简易的任务实现(Python)

cuNumeric提供Legion Mapping的自定义实现



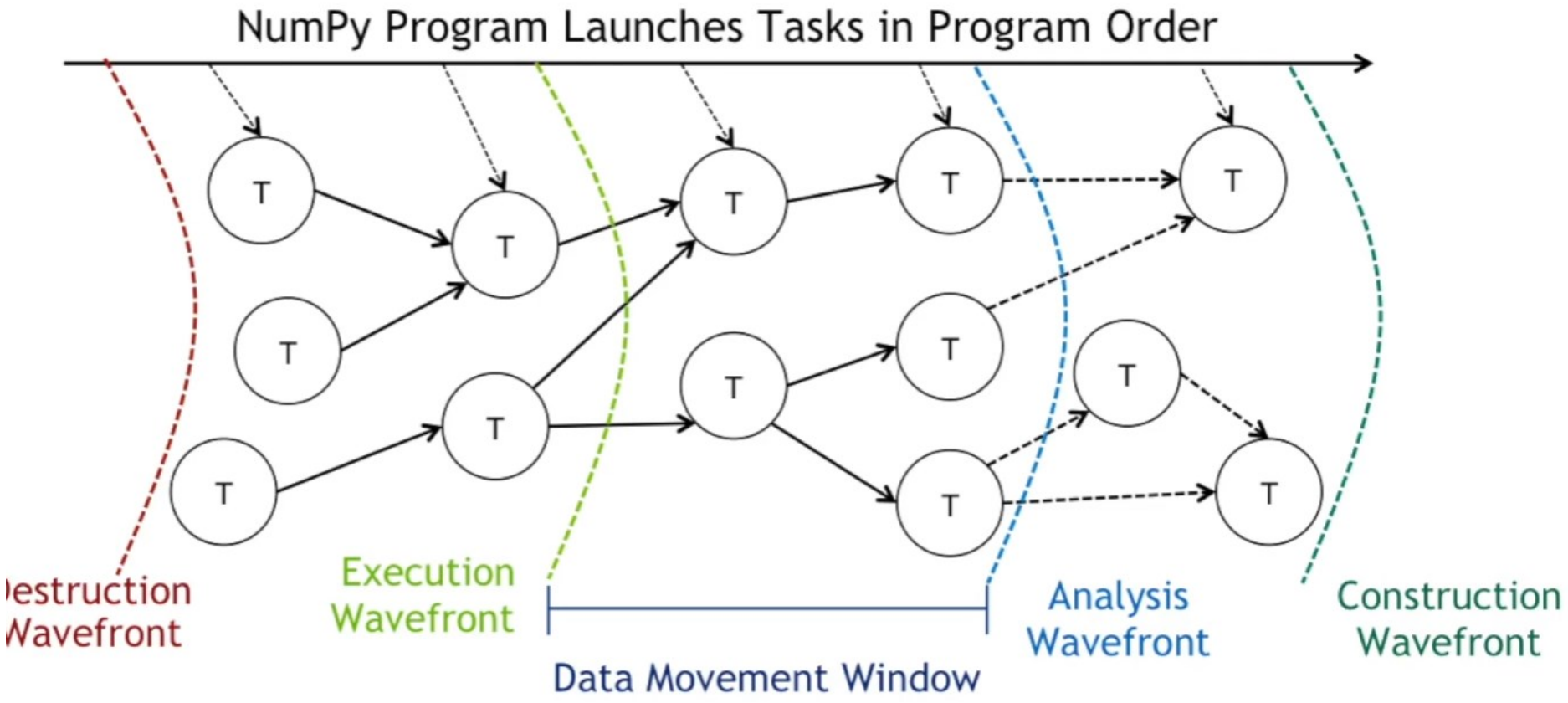
Legion软件栈详细介绍 – Libraries/Apps

- Legate + cuNumeric



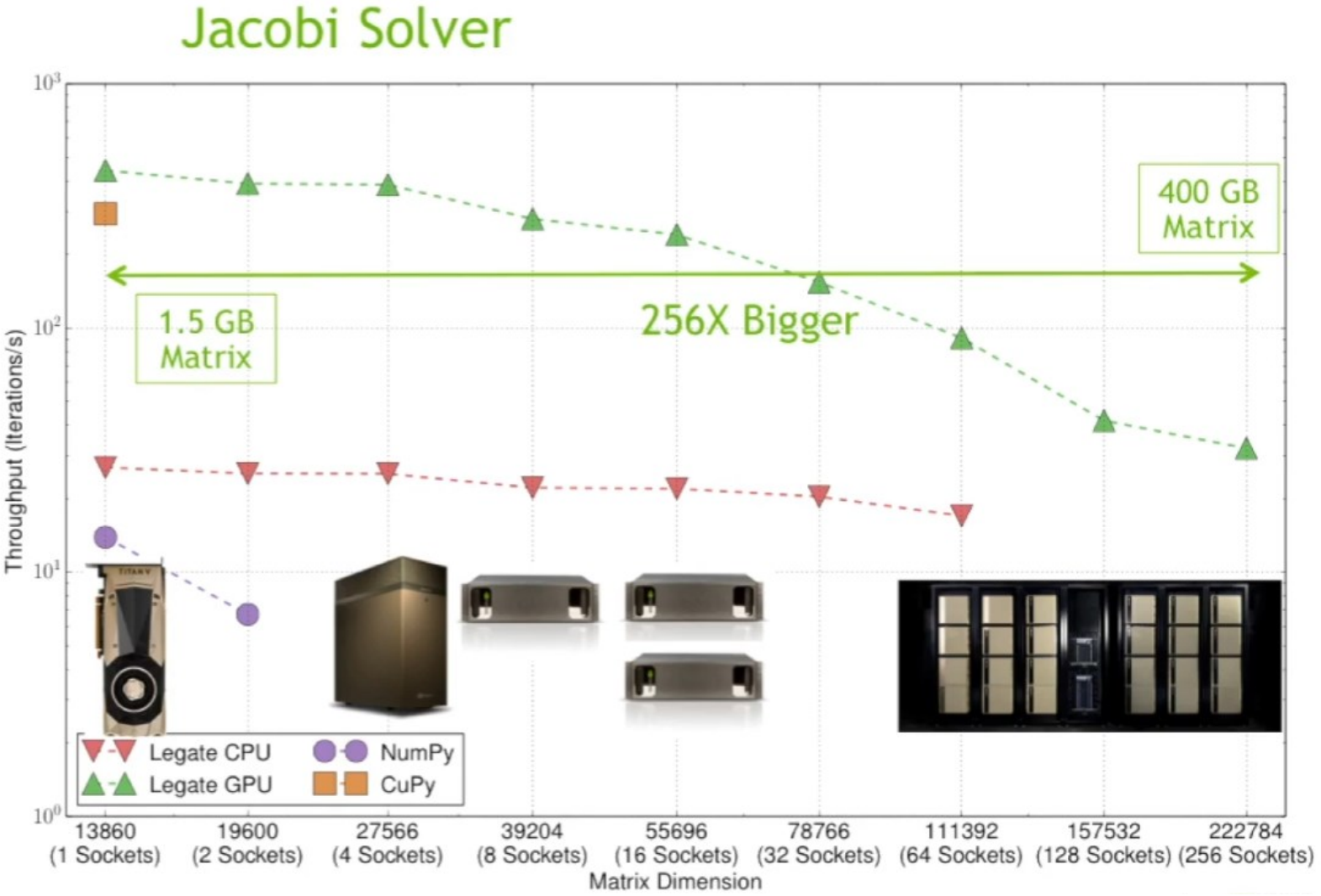
Legion软件栈详细介绍 – Libraries/Apps

- Legate + cuNumeric



Legion软件栈详细介绍 – Libraries/Apps

- Legate + cuNumeric 性能



大纲

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

总结

- 优点：
  - 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
- 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，大量的自己造轮子的组件，和现有的成熟实现相比可能存在性能劣势

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

- 利用Legion和Realm提供的抽象设计，将超算的底层软件栈实现适配
- 现有问题：超算的体系结构非常不同，软件栈十分复杂，难以适配

未来工作方向2：基于Legion和Realm针对新的多元任务设计接口

- 针对新的多元应用进行适配，提供更广泛的并行优化
- 现有问题：上层多元应用的需求不明确，并行业务需要进一步分析

未来工作方向3：针对特定的任务实现高效的Mapper算法

- 自定义更优秀的Mapper策略，尽量超越现有的SOTA调度方案
- 现有问题：上层多元应用的需求不明确，并行业务需要进一步分析

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

内容：肖霖畅 杨承润 杨翼飞 伍睿智 介琛