Towards Heterogeneous and Distributed Computing in C++

Gordon Brown – Senior Software Engineer, SYCL & C++
Ruyman Reyes - Principal Software Eng., Programming Models
Michael Wong – VP of R&D, SYCL Chair, Chair of ISO C++ TM/Low Latency

DHPCC++ – May 2018

Leadership Products Enabling Advanced Applications on Complex Processor Systems

Company

High-performance software solutions for custom heterogeneous systems

Enabling the toughest processor systems with open-standards-based tools and middleware

Established 2002 in Scotland, UK

Products

ComputeCpp™

C++ platform with SYCL, enabling vision

and machine learning applications e.g. TensorFlow™

ComputeAorta™

The heart of Codeplay's compute technology, enabling OpenCL™, SPIR™, HSA™ and Vulkan™

Markets

Vision Processing
Machine Learning
Data Compute

High Performance Computing (HPC)
Automotive (ISO 26262)
IoT, Smartphones & Tablets
Medical & Industrial

Partners

AMD
ARM
BROADCOM
Imagination
QUALCOMM
Movidius

Many Global Companies

About me...

- Background in C++ programming models for heterogeneous systems
- Developer with Codeplay Software for 6 years
- Worked on ComputeCpp (SYCL) since it’s inception
- Contributor to the Khronos SYCL standard for over 5 years
- Contributor to C++ executors and heterogeneity for 2 years

Disclaimer

The proposals describe here are work in progress

These may not reflect the final proposals

Acknowledgements

Jared Hoberock, Chris Kohlhoff, Chris Mysen, Michael Garland, Michael Wong, Carter Edwards, Hartmut Kaiser, Hans Boehm, Torvald Riegel, Lee Howes, David Hollman, Bryce Lelbach, Gor Nishanov, Thomas Heller, Geoffrey Romer, H. Carter Edwards, Thomas Rodgers, Mark Hoemmen, Patrice Roy, Carl Cook, Jeff Hammond, Christian Trott, Paul Blinzer, Alex Voicu, Nat Goodspeed, Christopher Di Bella, Toomas Remmelg and Morris Hafner

Agenda

- P0443r7 A Unified Executors Proposal for C++
- P1019r0 Integrating Executors with Parallel Algorithms
- P0796r2 Supporting Heterogeneous & Distributed Computing Through Affinity

P0443r7 A Unified Executors Proposal for C++

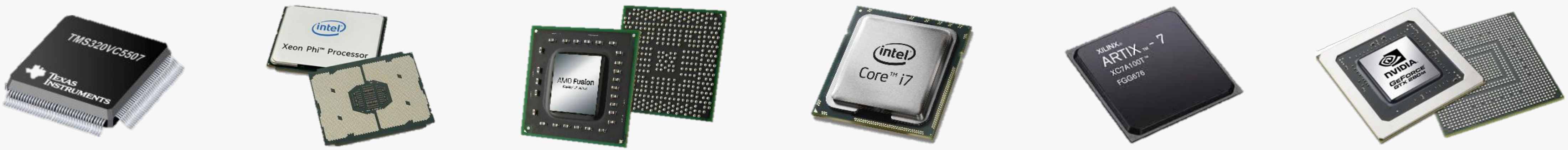
What are executors?

invoke async parallel algorithms future::then post

defer define_task_block dispatch asynchronous operations strand<>

Unified interface for execution

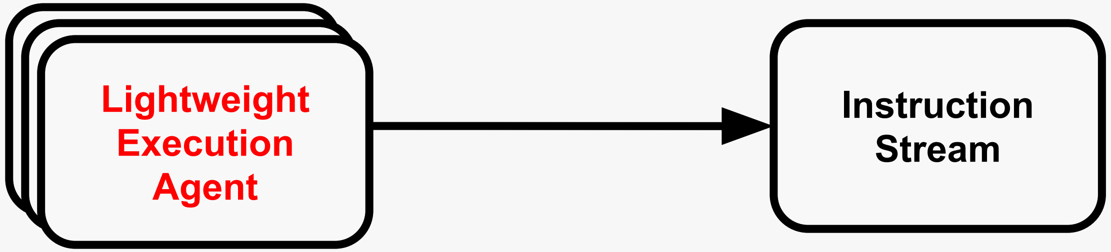
SYCL / OpenCL / CUDA / HCC	OpenMP / MPI	C++ Thread Pool	Boost.Asio / Networking TS



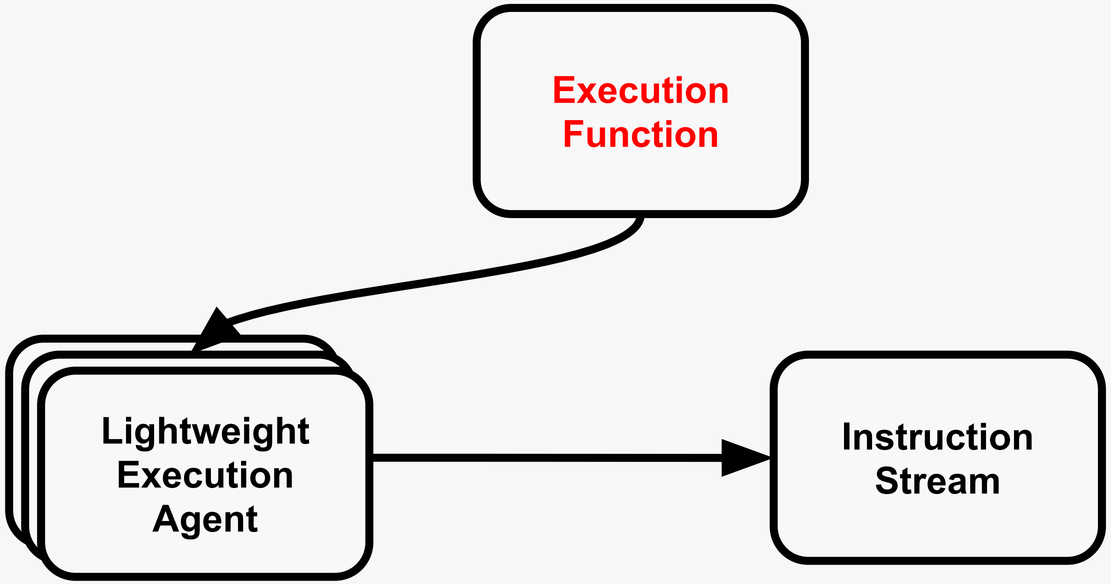
Topology of execution

- An instruction stream is a callable object that is to be executed

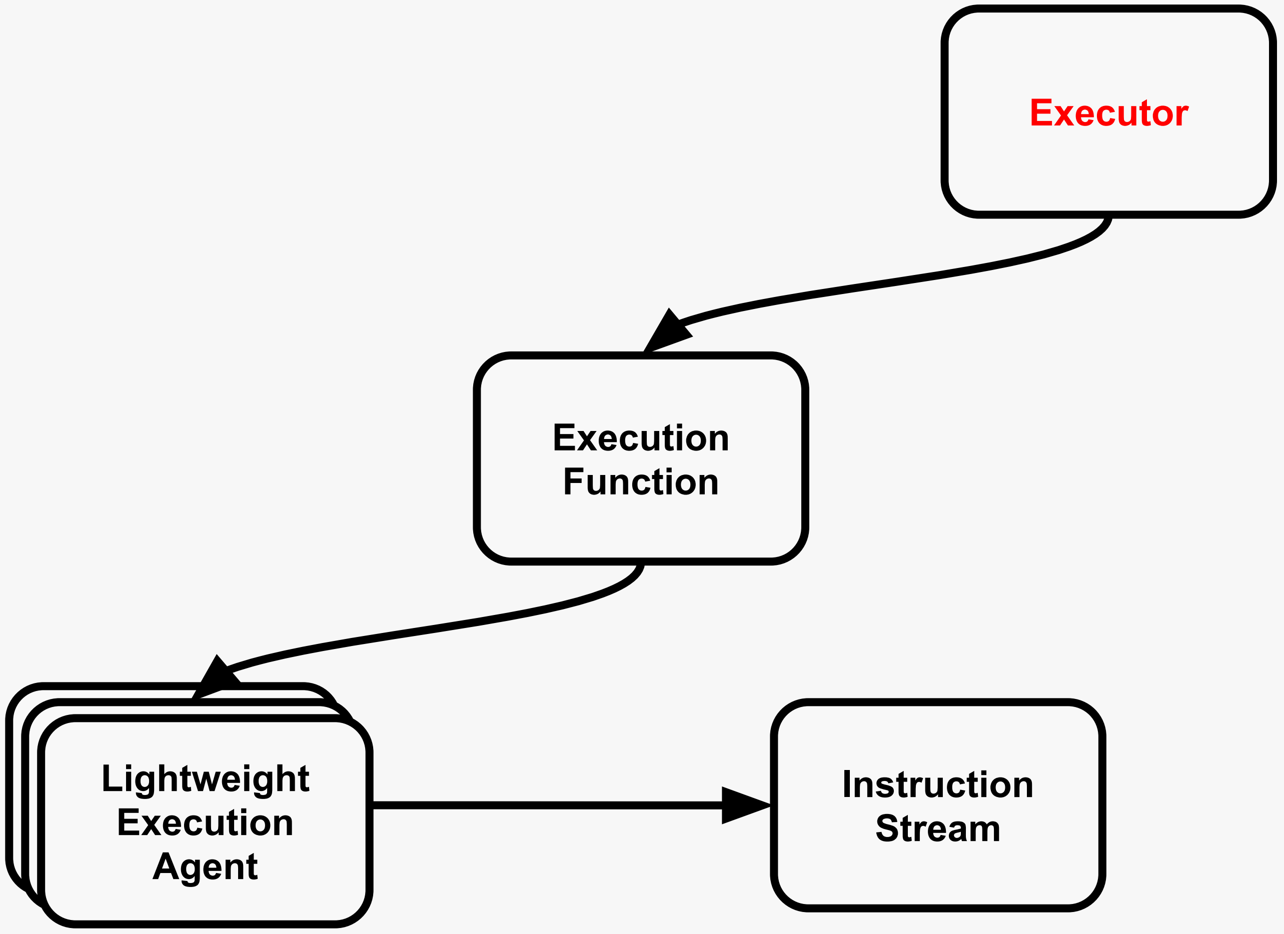
- A light-weight execution agent is a single thread of execution executing the instruction stream



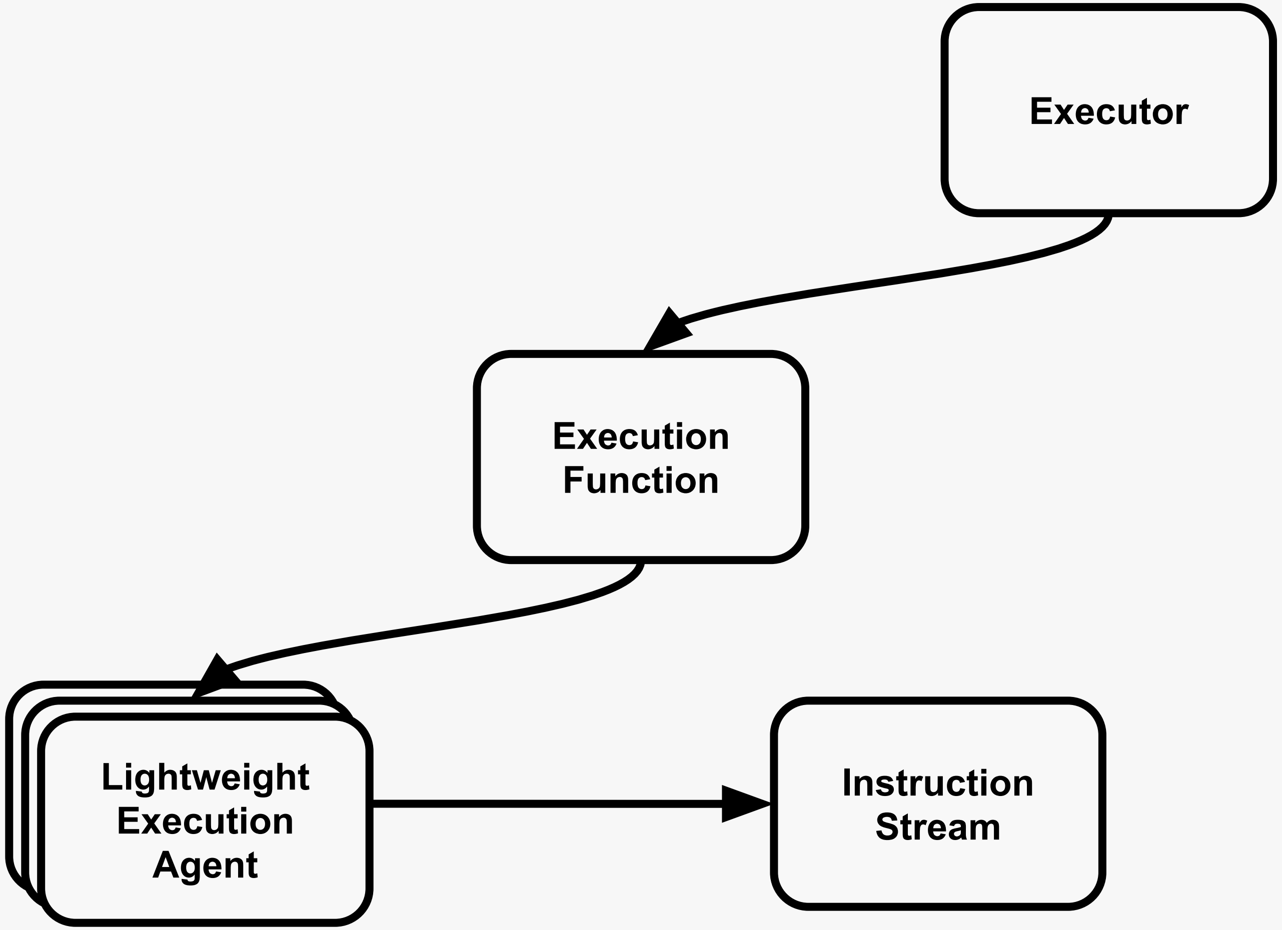
- An execution function is a function which executes an instruction stream on one or more light-weight execution agents with a particular set of properties

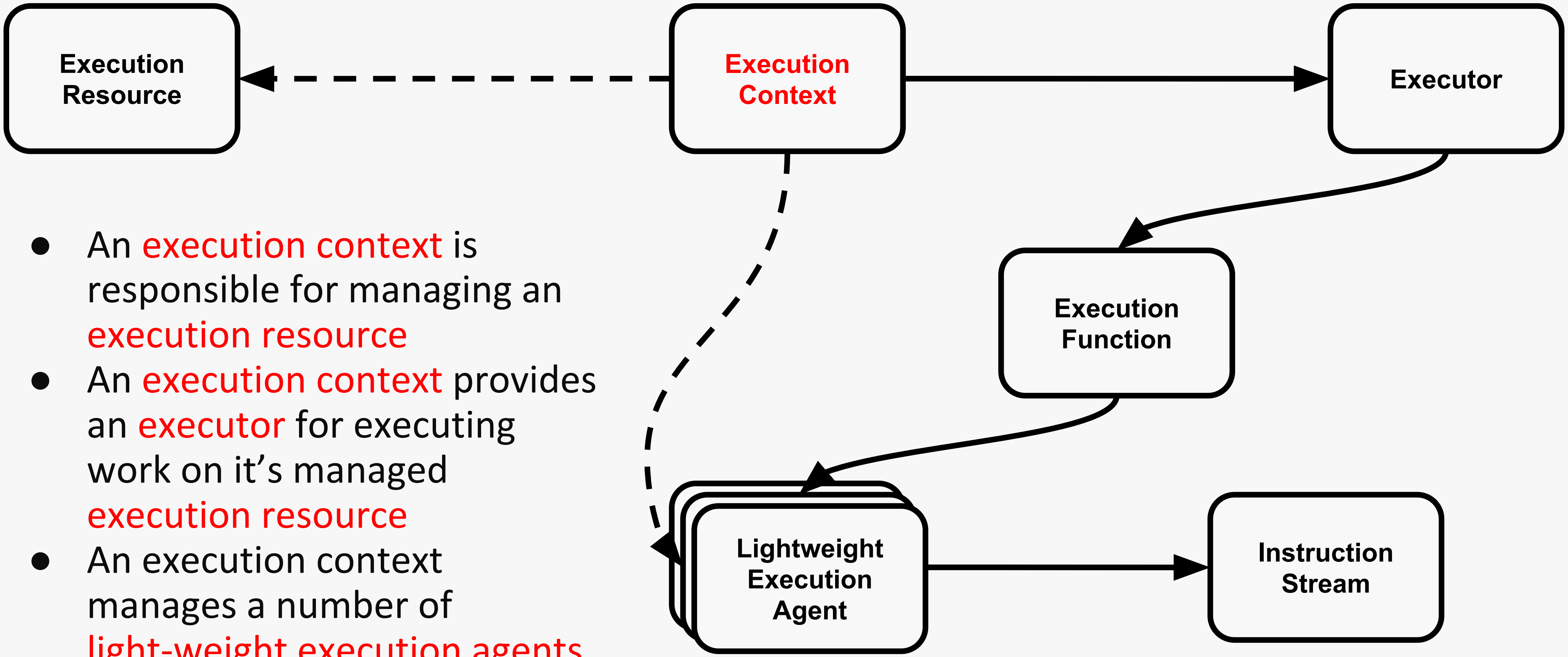


- An executor is an interface that describes where, when and how to execute work
- An executor can spawn one or more light-weight execution agents each executing the same instruction stream via execution functions



- An execution resource is the hardware abstraction which is executing the work
- Examples of an execution resource are a CPU thread pool, GPU context, network device





- An execution context is responsible for managing an execution resource
- An execution context provides an executor for executing work on it’s managed execution resource
- An execution context manages a number of light-weight execution agents

```
{
  static_thread_pool pool;
  auto exec = pool.executor();
  exec.execute([&](){ func(); });
}
```

Properties of execution

Properties	Description
Cardinality	Specifies whether the executor supports single and/or bulk execution
Directionality	Specifies whether the executor supports oneway and/or twoway execution
Blocking guarantees	Specifies whether the execution function will or may block the caller on completion

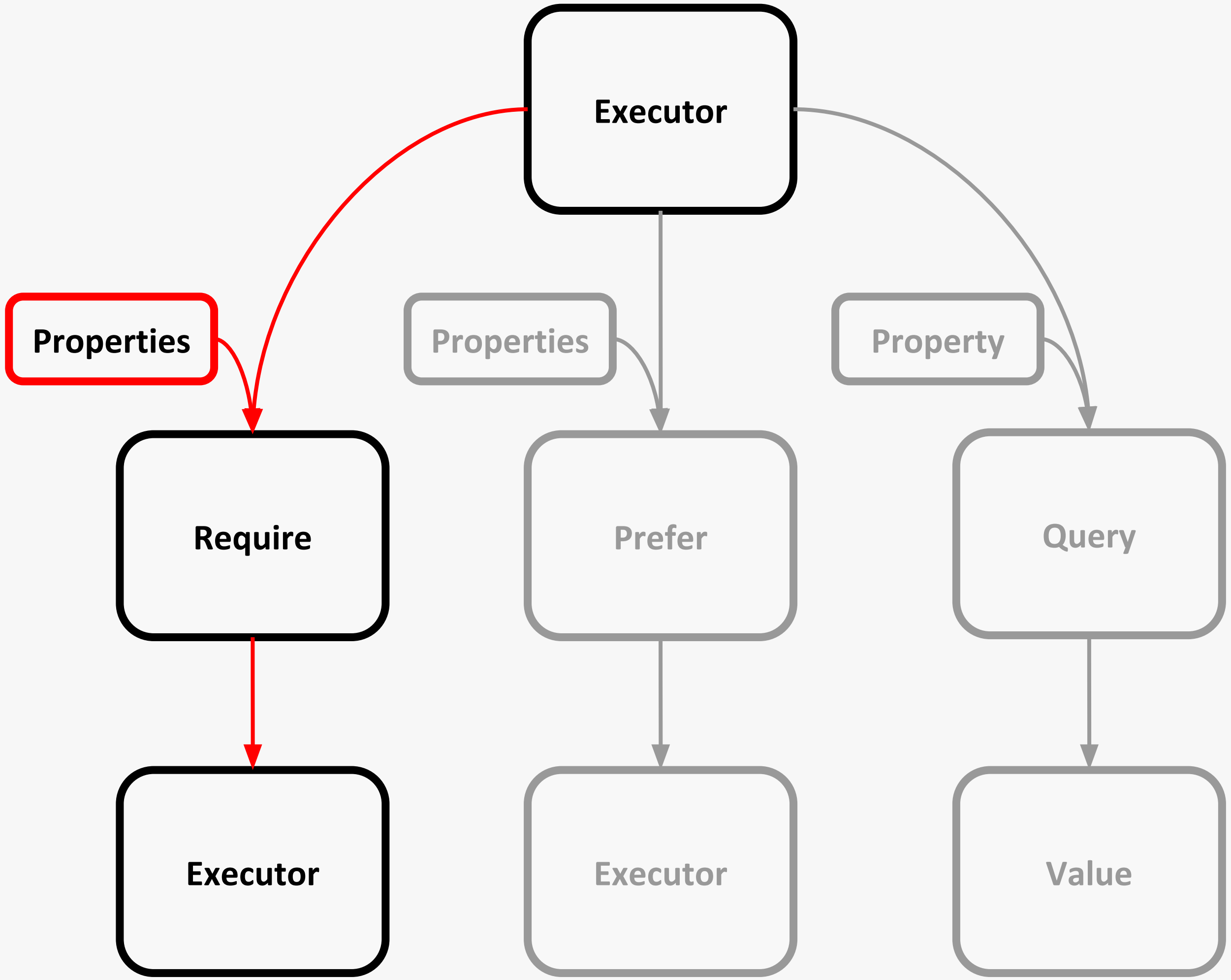
Properties	Description
Cardinality	Specifies whether the executor supports single and/or bulk execution
Directionality	Specifies whether the executor supports oneway and/or twoway execution
Blocking guarantees	Specifies whether the execution function will or may block the caller on completion

Properties which modify the interface

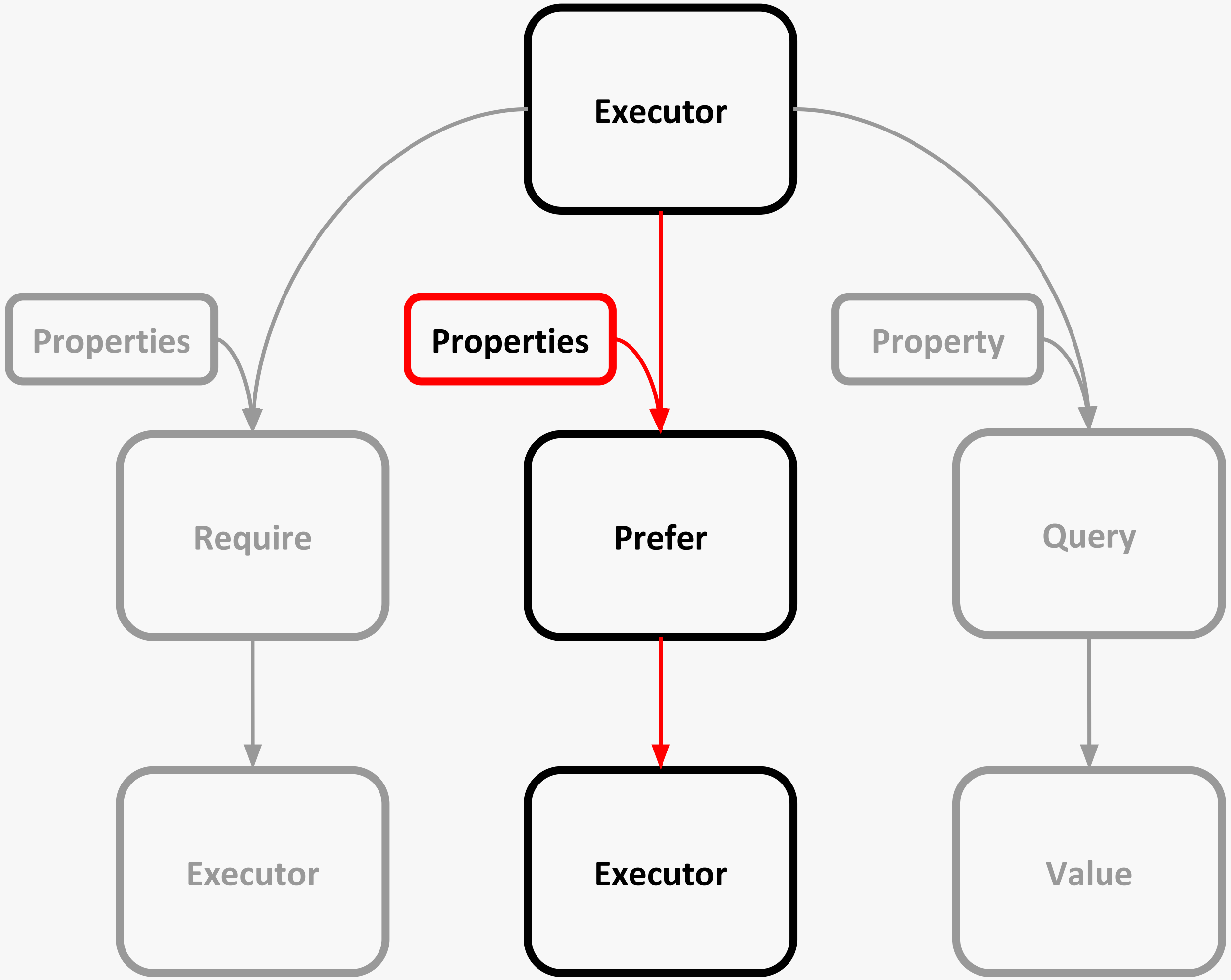
Properties	Description
Thread mapping semantics	Specifies the way in which the instruction stream is mapped to threads of execution
Bulk execution guarantees	Specifies the guarantees between threads of execution within a bulk execution
Continuation	Specifies whether the instruction stream should be executed as a continuation
Future work submission	Specifies whether or not the execution context should expect future work to be submitted
Allocator	Specifies the allocator to use when allocating memory for the instruction stream

Executor customisation

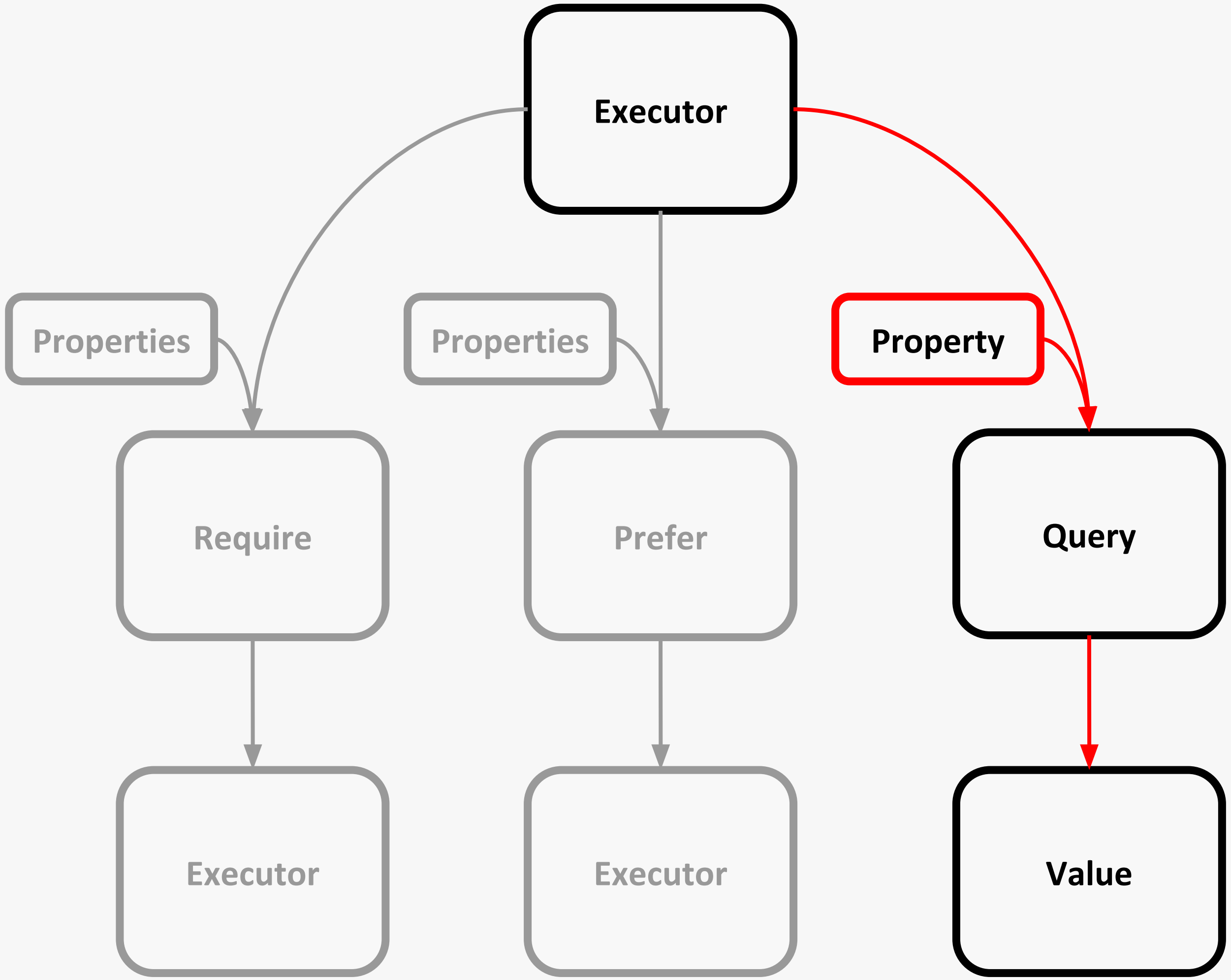
- Performing a require returns an executor that will have the requested properties
  - If the properties are already supported the original executor is returned
  - If the properties are not supported this will result in a compile-time error



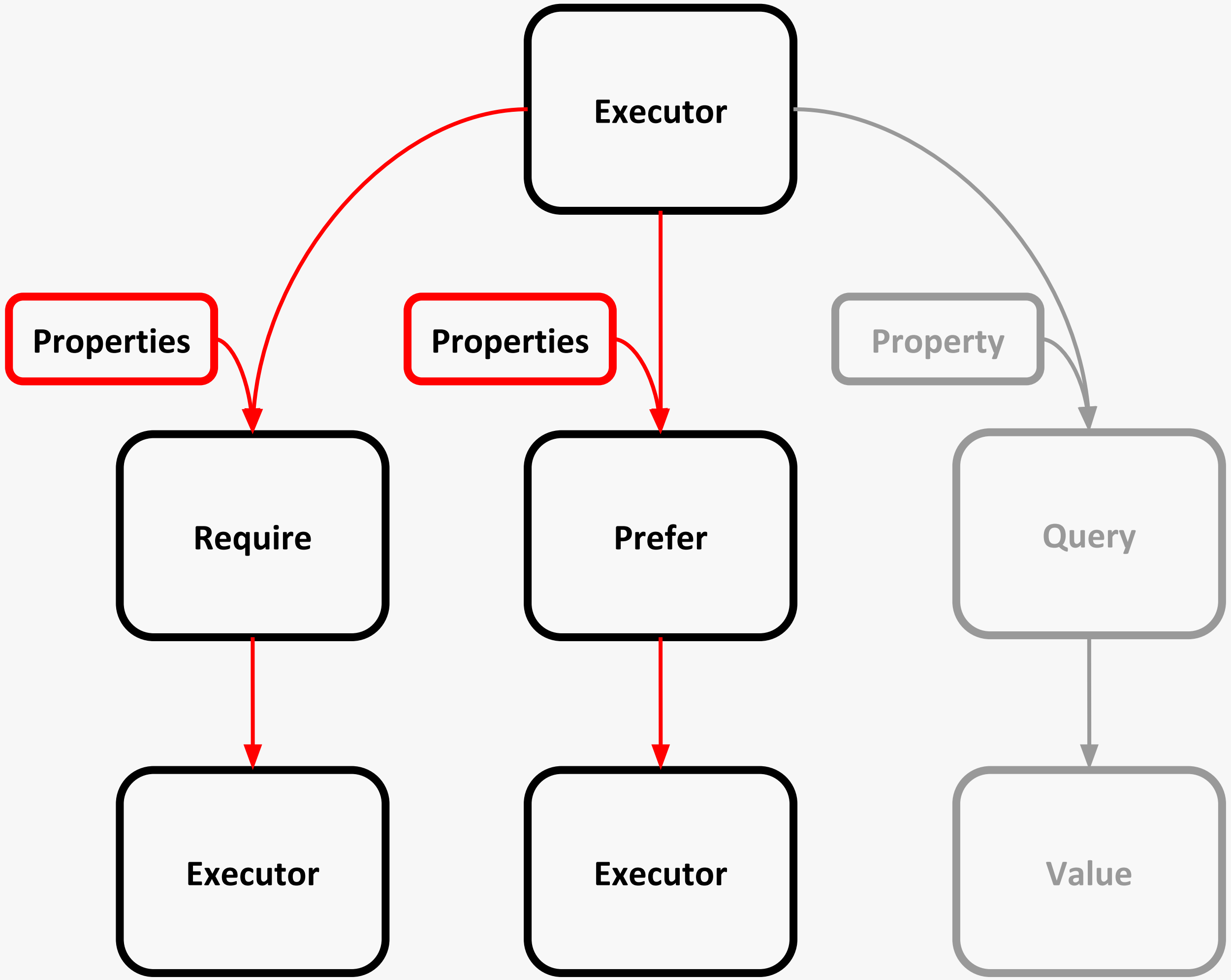
- Performing a prefer returns an executor that may have the requested properties
    - If the properties are already supported the same executor is returned
    - If the properties are not supported the executor will simply return the original executor



- Performing a query returns the current value of a specific property
  - In many cases this value will be a boolean
  - In some cases this query can be performed at compile-time if property::static_query_v is available



- Properties that are successfully requested via require or prefer can be supported in two ways
  - An executor implementation can natively support the property
  - An executor can support a property via an adaptation



```
execution::oneway_executor exec;

auto newExec = execution::require(exec, twoway);

auto fut = newExec.twoway_execute([&]() {
  return func();
});
```

Require

```
execution::oneway_executor exec;

auto newExec = execution::require(exec, twoway);

auto fut = newExec.twoway_execute([&]() {
  return func();
});
```

Require

```
execution::possibly_blocking_executor exec;

auto newExec = execution::prefer(exec, never_blocking);

newExec.execute([&]() {
  func();
});
```

Prefer

```
execution::oneway_executor exec;

auto newExec = execution::require(exec, twoway);

auto fut = newExec.twoway_execute([&]() {
  return func();
});
```

Require

```
execution::possibly_blocking_executor exec;

auto newExec = execution::prefer(exec, never_blocking);

newExec.execute([&]() {
  func();
});
```

Prefer

```
execution::possibly_blocking_executor exec;

auto newExec = execution::prefer(exec, never_blocking);

auto isNeverBlocking = execution::query(newExec, never_blocking);
```

Query

Execution functions

	One-way	Two-way
Single	`execute()`	`twoway_execute()`
Bulk	`bulk_execute()`	`bulk_twoway_execute()`

```
{
  execution::oneway_executor exec;
  exec.execute([&]() {
    func();
  });
}
```

Single One-way

```
{
  execution::oneway_executor exec;
  exec.execute([&]() {
    func();
  });
}
```

Single One-way

```
{
  execution::twoway_executor exec;
  auto fut = exec.twoway_execute([&](){
    return func();
  });
}
```

Single Two-way

```
{
  execution::oneway_executor exec;
  exec.execute([&]() {
    func();
  });
}
```

Single One-way

```
{
  execution::twoway_executor exec;
  auto fut = exec.twoway_execute([&](){
    return func();
  });
}
```

Single Two-way

```
{
  execution::bulk_executor exec;
  exec.bulk_execute([&](size_t index,
    auto &s){
    func(i, s);
  }, shape, sharedFactory);
}
```

Bulk One-way

```
{
  execution::oneway_executor exec;
  exec.execute([&]() {
    func();
  });
}
```

Single One-way

```
{
  execution::twoway_executor exec;
  auto fut = exec.twoway_execute([&](){
    return func();
  });
}
```

Single Two-way

```
{
  execution::bulk_executor exec;
  exec.bulk_execute([&](size_t index,
    auto &s){
    func(i, s);
  }, shape, sharedFactory);
}
```

Bulk One-way

```
{
  execution::bulk_twoway_executor exec;
  auto fut = exec.bulk_twoway_execute(
    [&](size_t index, auto &r, auto &s){
    func(i, r, s);
  }, shape, resultFactory, sharedFactory);
}
```

Bulk Two-way

P1019r0 Integrating Executors with Parallel Algorithms

```
{

  vector<int> data = { 4, 9, 5, 1, 3, 9, 5, 0, 3, 5, 1, 3 };

  execution::static_thread_pool pool;
  auto exec = pool.executor();

  sort(execution::par.on(exec), data.begin(), data.end());

}
```

P0967r2 Supporting Heterogeneous & Distributed Computing through Affinity

Why does C++ need affinity support?

- All systems are inherently heterogeneous
  - Desktop systems commonly have compute capable GPUs
  - Server systems commonly have multiple CPU nodes or CPU + {GPU, FPGA, DSP, TPU, etc } nodes
  - Mobile and embedded systems commonly have GPUs and/or often other specialised chips
- Many systems are distributed
  - HPC server and cloud systems have a distribution of a large number of interconnected nodes

- Memory access is no longer simple
    - Distributed memory regions across NUMA nodes
    - Hierarchical GPU memory regions
    - On-chip shared memory
    - Off-chip DMA transfers
    - Shared virtual memory through cache coherency
    - High Bandwidth Memory (HBM)

- Affinity is supported through many C++ libraries / standards
  - Hwloc (Portable Hardware Locality)
  - OpenMP
  - MEMKIND
  - Cpuaff
  - Persistent Memory Programming
  - OpenCL / SYCL
  - HSA
  - Platform specific solutions: Windows / Linux / Solaris
  - Chapel, X10, UPC++
  - TBB
  - HPX
  - MADNESS

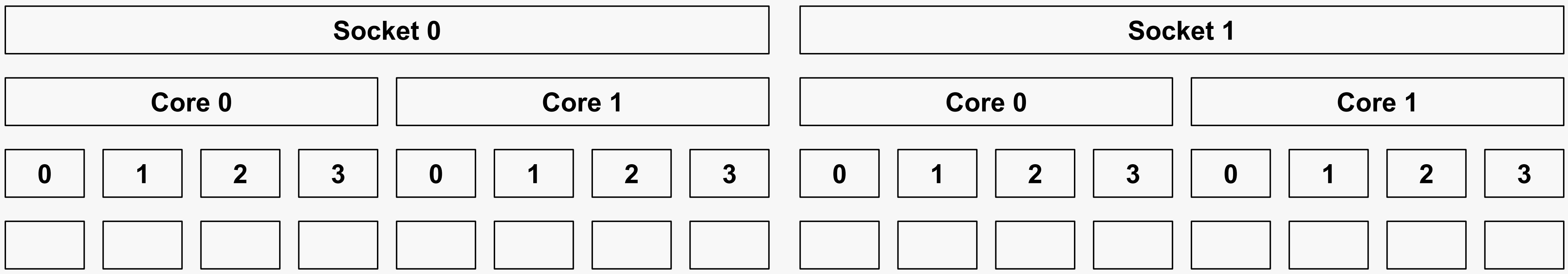
What are we proposing?

- Define an interface for discovering and querying affinity
  - Solution needs to be able to discover all resources within a system and query relative affinity between them
  - Solution needs to provide memory and process affinity
- Integrate closely with the unified executors proposal
  - Solution must align as closely as possible with the direction of the executors design
- Ensure scalability to heterogeneous and distributed systems
  - Solution needs to consider the limitations of heterogeneous and distributed systems to ensure scalability

- Executor properties
  - A collection of executor properties for describing affinity binding guarantees
  - Low granularity, relies on implementation applying property in an optimal way
  - Designed for users who are not particularly familiar with the architecture or for users who do not need to fine tune their code for affinity
- Execution resource topology
  - A framework for describing, discovering and querying the execution resources available within the system
  - High granularity, allows for fine grained control over affinity binding
  - Designed for users who have a high understanding of the architecture and for users who are implementing libraries or algorithms and need to fine tune their code for affinity

Bulk execution affinity properties

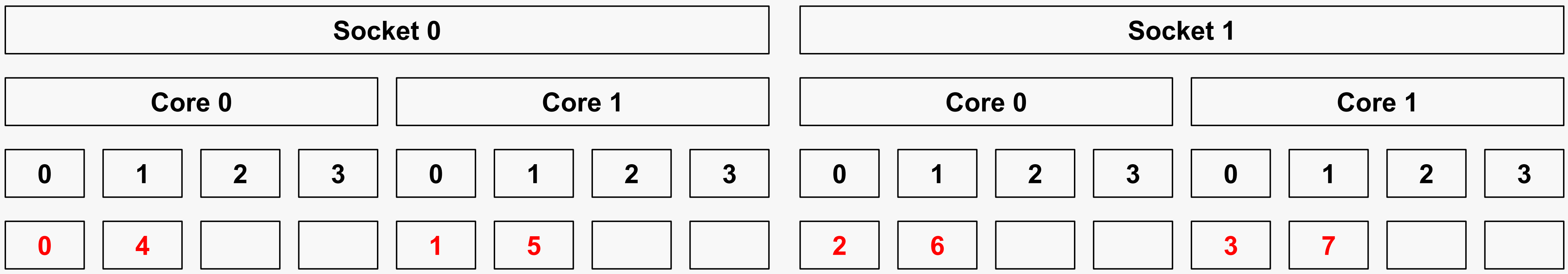
- Executor property which requires that an executor provide a particular guarantee of affinity binding pattern
  - Pattern can be **none**, **balanced**, **scatter** or **compact**
  - Requires that each execution agent be bound to a particular execution resource before the callable is called.
  - Binding must be consistent across all invocations of **bulk_execute**, **bulk_twoway_execute** or **bulk_then_execute**.



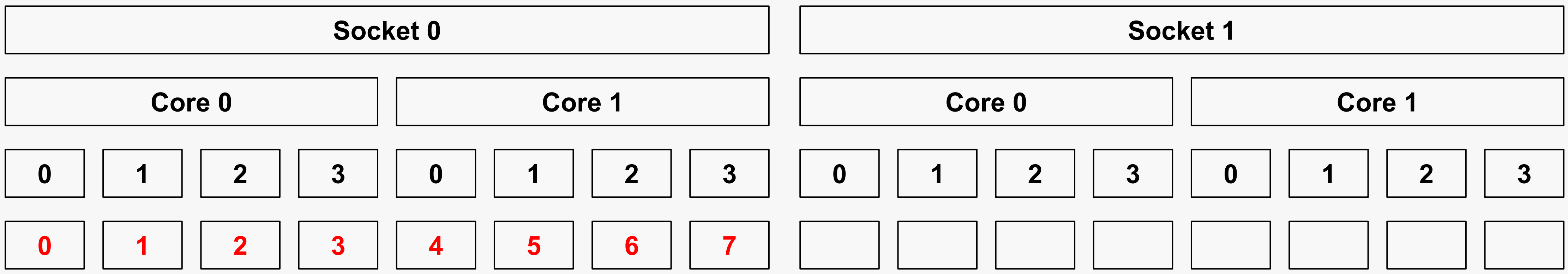
Socket 0
Socket 1
Core 0
Core 1
Core 0
Core 1
0
1
2
3
0
1
2
3
0
1
2
3
0
1
2
3

Socket 0								Socket 1							
Core 0				Core 1				Core 0				Core 1			
0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?

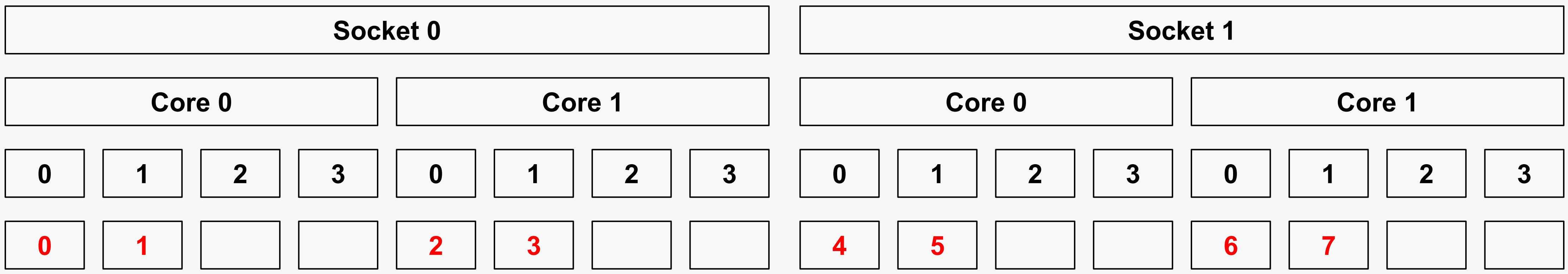
```
{
  auto exec = execution::execution_context{execRes}.executor();

  auto affExec = execution::require(exec, execution::bulk,
    execution::bulk_execution_affinity.none);

  affExec.bulk_execute([](std::size_t i, shared s) {
    func(i);
  }, 8, sharedFactory);
}
```




```
{
  auto exec = execution::execution_context{execRes}.executor();

  auto affExec = execution::require(exec, execution::bulk,
    execution::bulk_execution_affinity.scatter);

  affExec.bulk_execute([](std::size_t i, shared s) {
    func(i);
  }, 8, sharedFactory);
}
```




```
{
  auto exec = execution::execution_context{execRes}.executor();

  auto affExec = execution::require(exec, execution::bulk,
    execution::bulk_execution_affinity.compact);

  affExec.bulk_execute([](std::size_t i, shared s) {
    func(i);
  }, 8, sharedFactory);
}
```



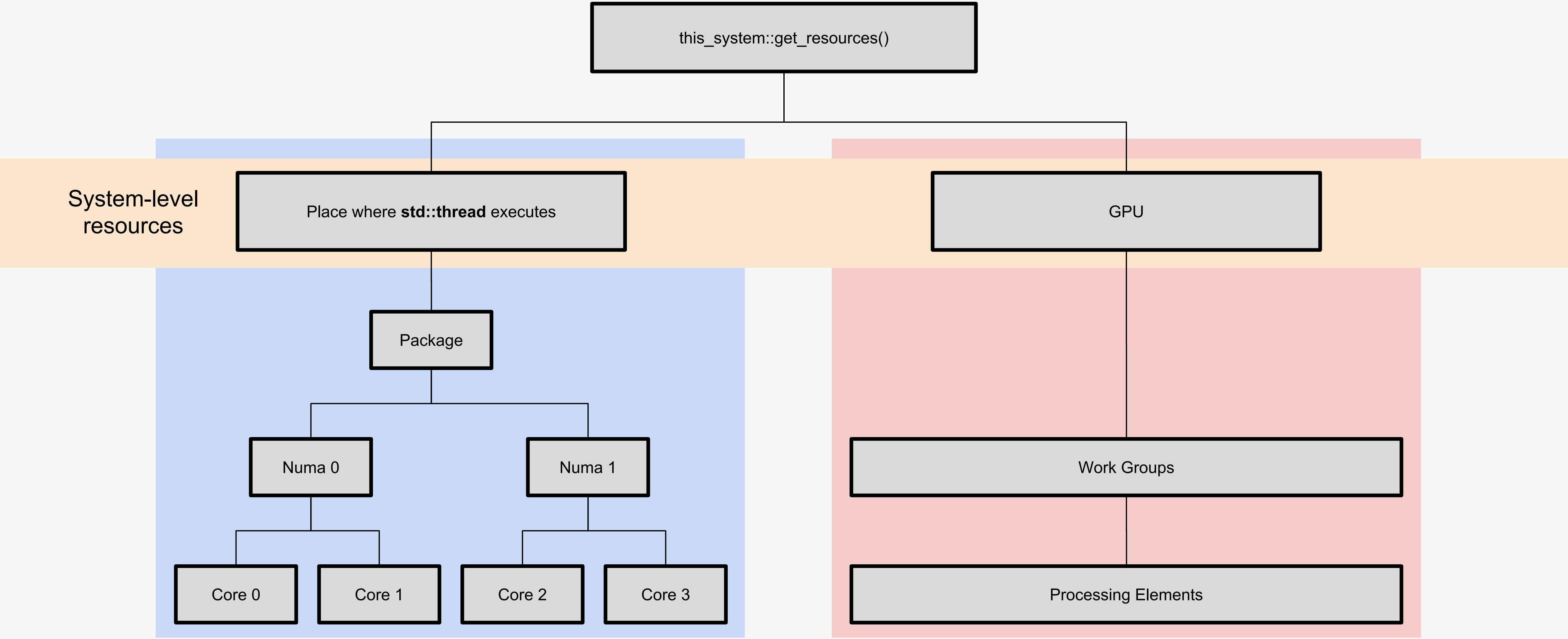

```
{
  auto exec = execution::execution_context{execRes}.executor();

  auto affExec = execution::require(exec, execution::bulk,
    execution::bulk_execution_affinity.balanced);

  affExec.bulk_execute([](std::size_t i, shared s) {
    func(i);
  }, 8, sharedFactory);
}
```

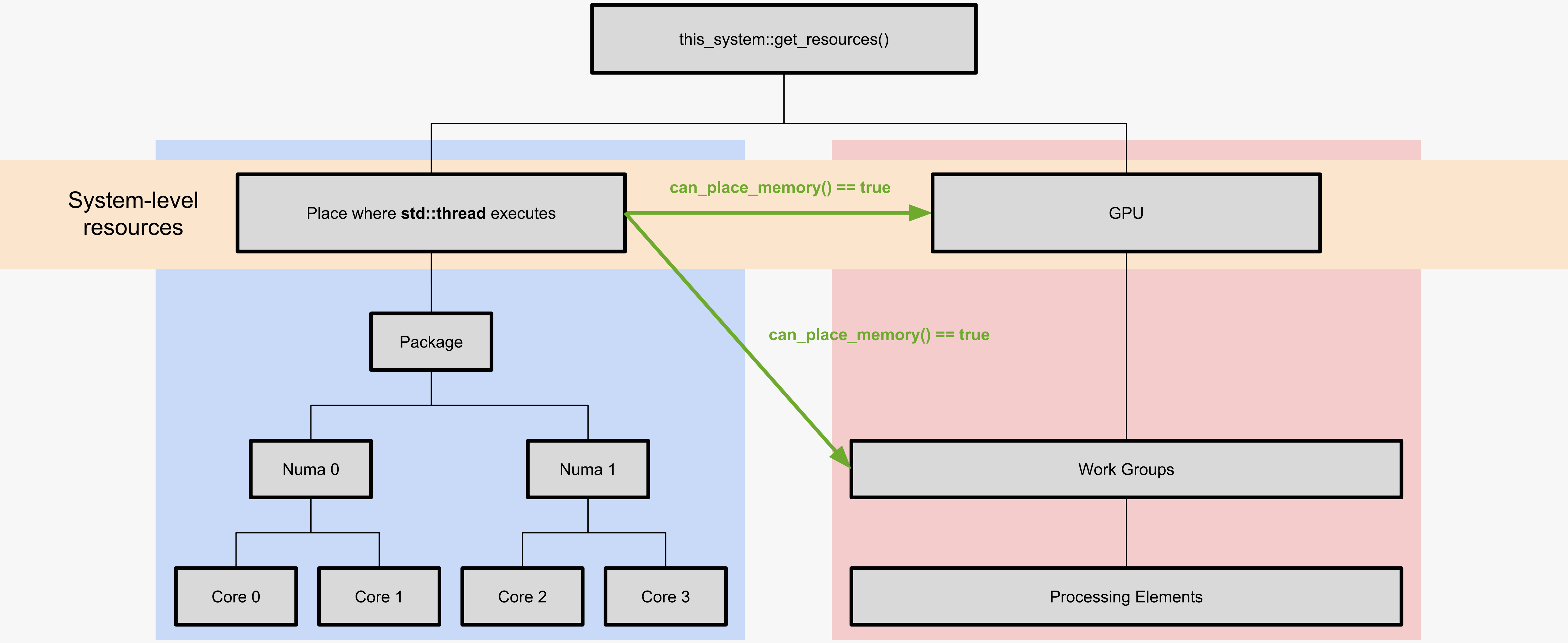
```
{

  vector<int> data = { 4, 9, 5, 1, 3, 9, 5, 0, 3, 5, 1, 3 };

  execution::static_thread_pool pool;
  auto exec = pool.executor();

  sort(execution::par.on(exec)
    .with(execution::bulk_execution_affinity.scatter),
      data.begin(), data.end());

}
```

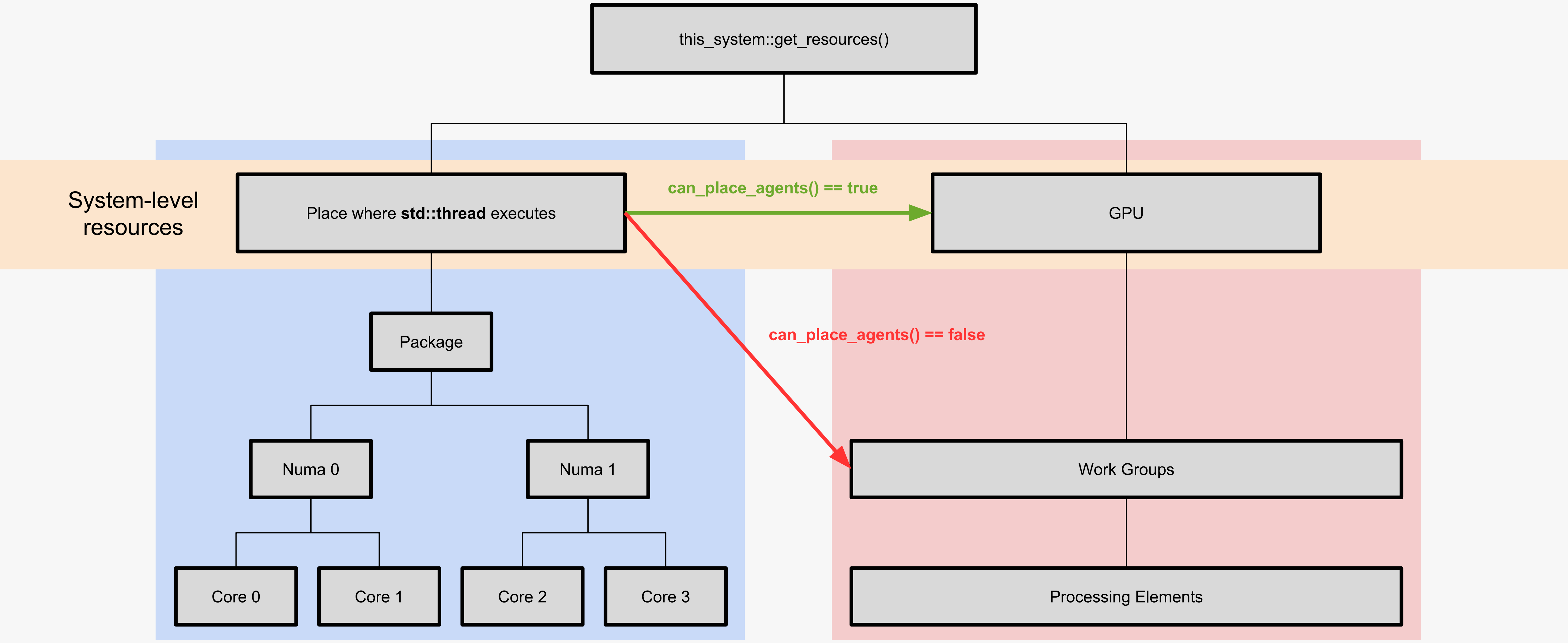
```
{

  vector<int> data = { 4, 9, 5, 1, 3, 9, 5, 0, 3, 5, 1, 3 };

  execution::static_thread_pool pool;
  auto exec = pool.executor();

  sort(execution::par.on(exec)
    .with(execution::bulk_execution_affinity.scatter),
      data.begin(), data.end());

}
```

Not yet proposed

Execution resource topology


this_system::get_resources()
System-level resources
Place where std::thread executes
GPU
Package
Numa 0
Numa 1
Work Groups
Core 0
Core 1
Core 2
Core 3
Processing Elements


this_system::get_resources()
System-level resources
Place where **std::thread** executes
can_place_memory() == true
GPU
Package
can_place_memory() == true
Numa 0
Numa 1
Work Groups
Core 0
Core 1
Core 2
Core 3
Processing Elements

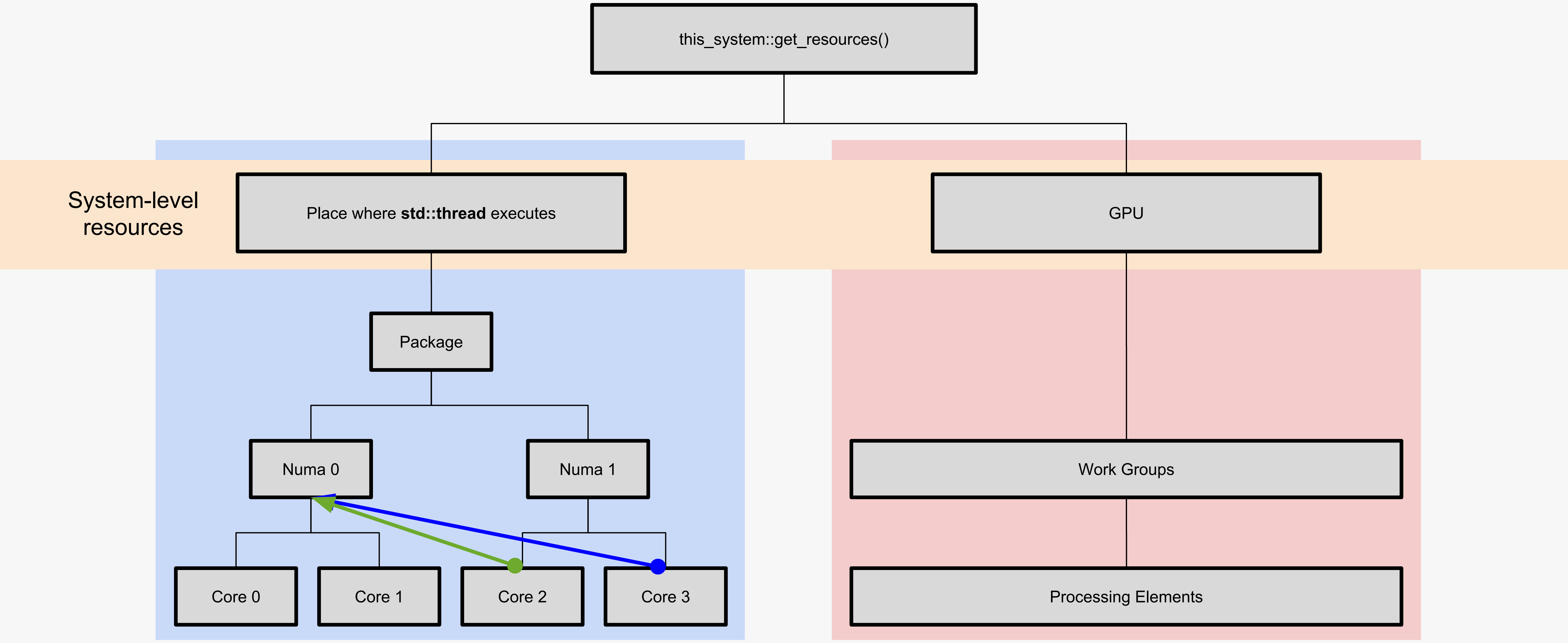

this_system::get_resources()
System-level resources
Place where **std::thread** executes
can_place_agents() == true
GPU
can_place_agents() == false
Package
Numa 0
Numa 1
Work Groups
Core 0
Core 1
Core 2
Core 3
Processing Elements

```
{

  auto systemLevelResources =
    std::execution::this_system::get_resources();

  // output names of member resources
  for (auto res : systemLevelResources) {
    std::cout << res.name() << ”\n”;

}
```

Querying relative affinity of execution resources

- An affinity_query is constructed from two execution resources
  - An affinity query represents an operation:- read, write, copy, move, map
  - An affinity query represents a metric:- latency, bandwidth, capacity, power consumption
- Some operations have restrictions on the parameters
  - Requires `can_place_memory()` or `can_place_agents()` to be `true`
- Two `affinity_query` objects can be compared
  - Comparison operators return the relative affinity as a magnitude
- The native metric can be queried directly
  - By calling `native_metric()`

```
/* Query latency of reading memory in A from a task executing in B*/
auto readBfromA = affinity_query<affinity::read, affinity::latency>(A, B);

/* Query latency of reading memory in A from a task executing in B*/
auto readCfromA = affinity_query<affinity::read, affinity::latency>(A, C);

/* Relative latency of reading B over reading C */
auto isBMoreCostly = readBFromA > readCFromA;

/* Relative latency of reading C over reading B */
auto isCMoreCostly = readCFromA > readBFromA;
```



relativeLatency = affinity_query<read, latency>(core2, numa0) > affinity_query<read, latency>(core3, numa0)

Binding execution & allocation

- An `execution_context` can then be used to execute work and allocate memory
  - An `execution_context` provides an executor to execute work on the execution resources it represents
  - An `execution_context` provides an allocator to allocate memory with affinity to the execution resources it represents
- For example:
  - The `execution_context` of a NUMA node `execution_resource` may allow you to allocate memory and execute work with affinity to that node
  - An `execution_context` of the local memory region of a GPU `execution_resource` may allow you to allocate memory with affinity to that node but not to execute work

```
/* If an execution resource can place agents */
if (execResource.can_place_agents()) {

  /* Construct an execution context from an execution resource */
  execution_context execContext(execResource);

  /* Retrieve the executor for the execution context */
  auto exec = execContext.executor();

   /* Execute user function using the executor of the execution context */
   exec.bulk_oneway_execute([](size_t index){
       some_function();
   }, size);
}
```

```
/* If an execution resource can place memory */
if (execResource.can_place_memory()) {

    /* Create an alias to the execution context allocator type */
    using allocator_t = vendor_a::execution_context::allocator_type;

    /* Construct an execution context from an execution resource */
    auto execContext = vendor_a::execution_context(execResource);

    /* Retrieve the memory resource from an execution resource */
    auto memoryResource = execContext.memory_resource();

    /* Construct a  pmr vector using a polymorphic allocator with the memory
    resource of the execution context */
    auto myVector = std::pmr::vector(allocator_t{memoryResource});
}
```

We're Hiring!
codeplay.com/careers/

codeplay®
THE HETEROGENEOUS SYSTEMS EXPERTS

Thank you for Listening

@codeplaysoft info@codeplay.com codeplay.com