# P0796r0: Supporting Heterogeneous & Distributed Computing Through Affinity


| | |
|---|---|
| **Date:** | 2017-10-16 |
| **Project:** | ISO JTC1/SC22/WG21: Programming Language C++ |
| **Audience** | SG14, SG1 |



| | |
|---|---|
| **Authors:** | Gordon Brown, Ruyman Reyes, Michael Wong, H. Carter Edwards, Thomas Rodgers |
| **Contributors** | Patrice Roy, Jeff Hammond, Simon Brand |
| **Emails:** | gordon@codeplay.com, ruyman@codeplay.com, michael@codeplay.com, hcedwar@sandia.gov, rodgert@twrodgers.com |
| **Reply to:** | michael@codeplay.com |


# Abstract


## Summary

This paper provides an initial meta-framework for the drives toward memory affinity for C++, given the direction from Toronto 2017 SG1 meeting that we should look towards defining affinity for C++ before looking at inaccessible memory as a solution to the separate memory problem towards supporting heterogeneous and distributed computing.


## Affinity Matters

Processor and memory binding, also called 'affinity', can help the performance of an application for many reasons. Keeping a process bound to a specific thread and local memory region optimizes cache affinity and reduces context switching and unnecessary scheduler activity. Since memory accesses to remote locations incur higher latency and lower bandwidth, control of thread placement to enforce affinity within parallel applications is crucial to fuel all the cores and to exploit the full performance of the memory subsystem on Non-Uniform Memory Architectures (NUMA).

Traditional homogeneous designs where memory is accessible at the same cost from all threads are difficult to scale up to the current computing needs. Current architectural trends move towards Non-Uniform Memory Access (NUMA) architectures where, although there is a coherent view of the memory, the cost to access it is not uniform. Memory affinity is especially useful in these systems. Using memory that is located on the same node as the processing unit helps to ensure that the application can access the data as quickly as possible.

In terms of traditional operating system behaviour, all processing elements of a CPU are threads, and they are placed using high-level policies that do not necessarily match the optimal usage pattern for a given application.

However, application developers must leverage the ***placement of memory*** and ***placement of threads*** in order to obtain maximum performance on current and future architecture.
For C++ developers to achieve this, native support for placement of threads and memory is critical for application portability. We will refer to this as the ***affinity problem***.

**Affinity** is defined as maintaining or improving the locality of threads and the most frequently used data, especially if the program behaviour is unpredictable or changes over time, or the machine is overloaded such that multiple programs interfere with each other.

Today, most OSes already can group processors according to their locality and distribute processes, while keeping threads close to the initial thread, or even avoid migrating threads and maintain first touch policy. But the fact is most programs can change their work distribution, especially in the presence of nested parallelism.

Frequently, data is initialized at the beginning of the program by the initial thread and is used by multiple threads. While automatic thread migration has been implemented in some OSes, the reality is that this has migration can cause high overhead. In an optimal case the operating system may automatically detect which thread access which data most frequently, or it may replicate data which is read by multiple threads, or migrate data which is modified and used by threads residing on remote locality groups.

The fact of it is that the OS may do a reasonable job, if the machine is not overloaded, and the first touch policy has been carefully used, and the program does not change its behaviour with respect to locality.

Imagine we have a code example using C++ STL container valarray using the latest C++17 parallel STL algorithm for_each, which applies the lambda to elements in the iterator range [begin, end) but using a parallel execution policy such that the workload is distributed in parallel across multiple cores on the CPU. We might expect the work to be fast, but because the containers of valarray are initialized automatically and automatically allocated on the master thread’s memory, we find that it is actually quite slow even when we have more than one thread.

```
// C++ valarray STL containers are initialized
// automatically and allocated on the master's memory
valarray<double> a(N), b(N), c(N);
//saxpying is slow
//Parallel foreach
std::for_each(par, std::begin(a), std::end(a),
[=](double b, double c){b[i]+scalar*c[i]});
// if we can migrate data at next usage and move pages close to next accessing thread
//using the affinity interface in future
...
//now faster, because data is local now
std::for_each(par, std::begin(a), std::end(a),
[=](double b, double c){b[i]+scalar*c[i]});
```

*Listing 1: Motivational example*

Now with the affinity interface we propose below and in future, we will hopefully find that there is significant increase in memory bandwidth when we have multiple threads by as much as 2x GB/s as thread count increases (using system call madvise on Sun systems to implement next touch policy to migrate the data close to the next executing thread).

The goal was that this would enable scaling up for heterogeneous and distributed computing in future. Indeed OpenMP [14] where one of the author participated in the design of its affinity model, has plans to integrate its affinity model with its heterogeneous model.[21]

## Background Research: State of the Art

The problem of effectively partitioning a system's topology is one which has been so for some time, and there are a range of third party libraries / standards which provides APIs to solve the problem. In order to standardise this process for the C++ standard we must carefully look at all of these. Below is a list of the libraries and standards which define an interface for affinity:

Portable Hardware Locality: https://www.open-mpi.org/projects/hwloc/
SYCL 1.2: https://www.khronos.org/registry/SYCL/specs/sycl-1.2.pdf
OpenCL 2.2: https://www.khronos.org/registry/OpenCL/specs/opencl-2.2.pdf
HSA: http://www.hsafoundation.com/standards/
OpenMP 4.0: https://www.cct.lsu.edu/mardigras14/abstracts#Wong
cpuaff: https://github.com/dcdillon/cpuaff
OpenMP 5.0: http://www.openmp.org/wp-content/uploads/openmp-TR5-final.pdf
Persistent Memory Programming: http://pmem.io/
MEMKIND: https://github.com/memkind/memkind
Solaris pbind(): https://docs.oracle.com/cd/E26502_01/html/E29031/pbind-1m.html
Linux sched_setaffinity(): https://linux.die.net/man/2/sched_setaffinity
Windows SetThreadAffinityMask():
https://msdn.microsoft.com/en-us/library/windows/desktop/ms686247(v=vs.85).aspx

Libraries such as the Portable Hardware Locality (hwloc) [9] provide a low level of hardware abstraction and offer a solution for the portability problem by supporting many platforms and operating systems. This and similar approaches may provide detailed hardware information in a tree-like structure. However, even some current systems cannot be represented correctly by a tree, where the number of hops between two sockets vary between socket pairs [14].

Some systems will provide additional user control through explicit binding of threads to processors through environment variables consumed by various compilers, system commands (e.g. Linux: taskset, numactl; Windows: start /affinity), or system calls for example Solaris has ***pbind()***, Linux has ***sched_setaffinity()*** and Windows has ***SetThreadAffinityMask()***.

# Problem Space

In this paper we describe the problem space of affinity for C++, the various challenges which need to be addressed in defining a partitioning and affinity interface for C++ and some suggested solutions:

- Querying a system's resource topology
- Querying the relative affinity of partitions
- Binding execution and allocation to a partition

Wherever possible, we also evaluate how an affinity based solution could be scaled to support both distributed and heterogeneous systems.

There are some additional challenges which we have been investigating but are not yet ready to be included in this paper and will be presented in a future paper:

- Migrating data from memory allocated in one partition to another
- Defining memory placement algorithms or policies

## Querying a System's Topology

The first task in allowing C++ applications to leverage memory locality is to provide the ability to query a ***system*** for its ***resource topology*** (commonly represented as a tree or graph) and traverse its ***execution resources***.

**Execution resource**

The capability of querying underlying ***execution resources*** of a given ***system*** is particularly important towards supporting affinity control in C++. The current proposal for executors [5] leaves the ***execution resource*** largely unspecified. This is intentional: ***execution resources***

will vary greatly between one implementation and another, and it is out of the scope of the current executors proposal to define those.

There is current work on extending the executors proposal to describe a typical interface for an ***execution context*** [8]. In this paper a typical ***execution context*** is defined with an interface for construction and comparison, and for retrieving an ***executor***, waiting on submitted work to complete and querying the underlying ***execution resource***.

Extending the executors interface to provide topology information can serve as a basis for providing a unified interface to expose affinity. This interface cannot mandate a specific architectural definition, and must be generic enough that future architectural evolutions can still be expressed.

**Level of abstraction**

An important consideration when defining a unified interface for querying the ***resource topology*** of a ***system*** is what level of abstraction should such an interface have and at what granularity the ***execution resources*** of the topology be described.

As both the level of abstraction of an ***execution resource*** and the granularity that it is described in will vary greatly from one implementation to another, it's important for the interface to be generic enough to support any level of abstraction. To achieve this we propose a generic hierarchical structure of ***execution resources***; each ***execution resource*** being composed of other ***execution resources*** recursively. Each ***execution resource*** within this hierarchy can be used to place memory (i.e allocate memory within the ***execution resource's*** memory region) or place execution (i.e. bind an execution to an ***execution resource's execution agents***) or both.

> *For example a NUMA system will likely have a hierarchy of nodes, each capable of placing memory and placing agents and a CPU + GPU system may have GPU local memory regions capable of placing memory but not capable of placing agents.*

| **Straw Poll** | Should the interface for querying a system's resource topology be completely abstract or should it provide specific components of the hardware architecture? |
|---|---|

**Representation**

Nowadays, there are various APIs and libraries that enable this functionality. One of the most commonly used is the Portable Hardware Locality (hwloc) [9]. Hwloc presents the hardware as a tree, where the root node represents the whole machine and subsequent levels represents different partitions depending on different hardware characteristics. The picture below shows the output of the hwloc visualization tool (lstopo) on a 2-socket Xeon E5300 server. Note that each socket is represented by a package in the graph. Each socket contain its own cache memories,

but both share the same NUMA memory region. Note also that different I/O units are visible underneath: Placement of these units w.r.t to memory and threads can be critical to performance. The ability of placing threads and/or allocating memory appropriately on the different components of this system is an important part of the process of application development, especially as hardware architectures get more complex. The documentation of lstopo [22] shows more interesting examples of topologies that can be encountered on today systems.

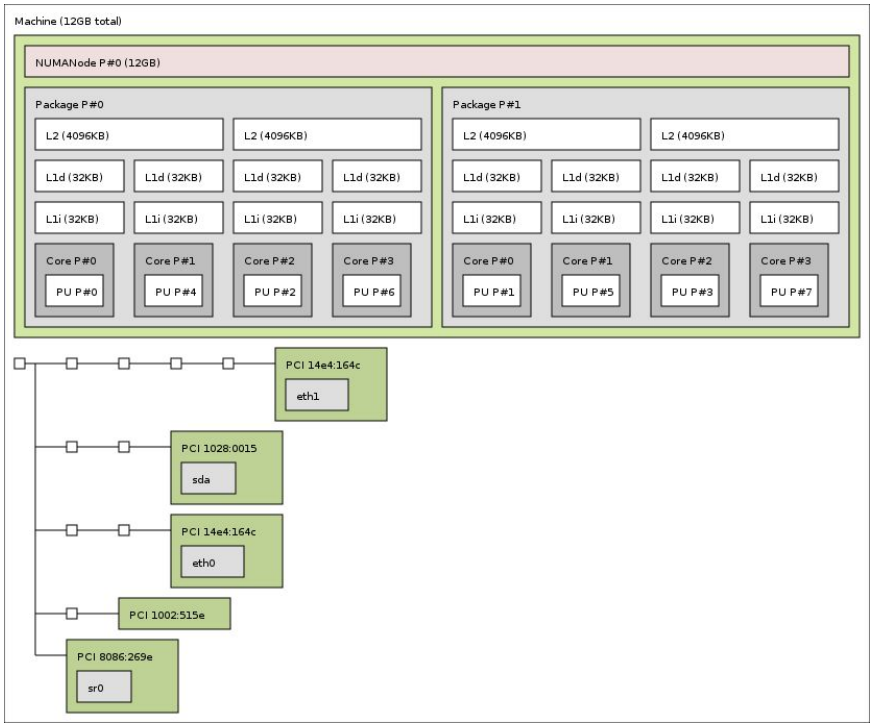


*Figure 1: Example system resource topology provided by hwloc*

However, systems are becoming increasingly non-hierarchical and a traditional tree based representation of a **system's resource topology** may not suffice anymore [18]. The HSA standard solve this problem by allowing a node in the topology to have multiple parent nodes [19]. This proposal in this paper currently focuses on a tree based solution for representing the **system's resource topology** however we wish to investigate other alternatives in a future paper.

| **Straw Poll** | Should the interface for querying a system's resource topology support non-hierarchical architectures.<br><br>*What kind of shape do we want for expressing the topology abstraction?* |
|---|---|

In the figure below (Figure 2) show an an example of how this could look in a C++ representation.

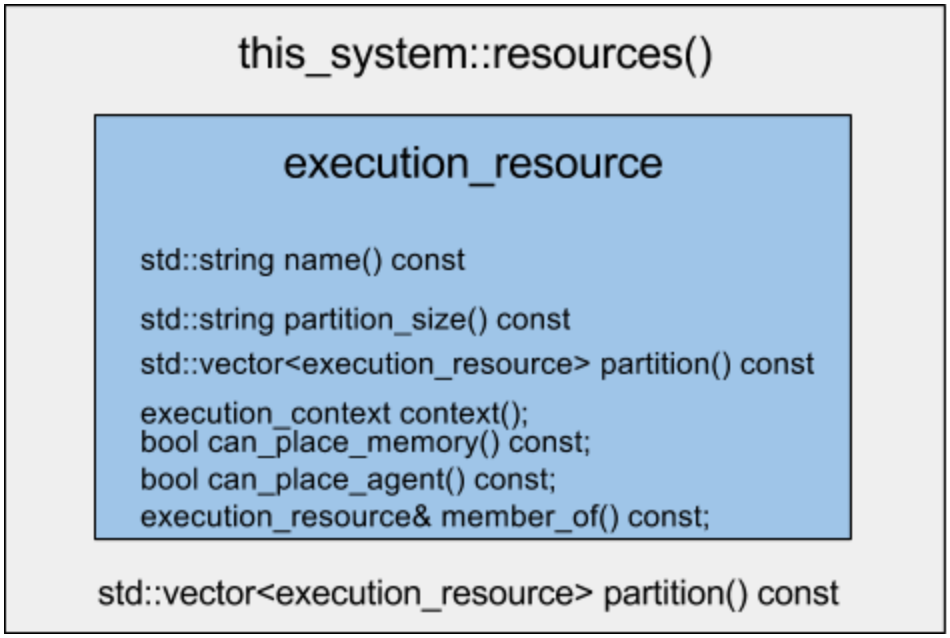


*Figure 2: Possible system hierarchy description*

**Extended Execution Resource Interface**

Below is a proposed interface for the generalization of the ***execution resource*** based on the definition of ***thread_execution_resource_t*** [8] with some extensions.

```
struct execution_resource {

  execution_resource() = delete;
  execution_resource(const execution_resource &) = delete;
  execution_resource(execution_resource &&) = delete;
  execution_resource &operator=(const execution_resource &) = delete;
  execution_resource &operator=(execution_resource &&) = delete;

  size_t concurrency() const noexcept;
  size_t partition_size() const noexcept;

  const execution_resource &partition(size_t i) const noexcept;
  const execution_resource &member_of() const noexcept;

  std::string name() const noexcept;

  bool can_place_memory() const noexcept;
  bool can_place_agent() const noexcept;

};
```

*Listing 2: Proposed extended execution resource interface*

The interface described above describes an execution resource as an object which cannot be user constructed, copied or moved, only referenced. It provides an interface for recursively querying the partitions and concurrency of it's child execution resources via the member functions ***concurrency***, ***partition_size*** and ***partition*** and it's parent execution resource via the member function ***member_of***. This interface is designed to match the design of ***thread_execution_resource_t*** [8]. Note that the resource is not limited to be an ***execution resource***, but also a general resource where no execution can take place but memory can be allocated such as off-chip memory.

> *The intention is that the actual implementation details of a resource topology are described in an execution context when required. This allows the execution resource objects to be lightweight objects that serve as identifiers that are only referenced.*

The interface also provides a member functions for querying whether the ***resource*** can place memory regions and place execution agents; ***can_place_memory*** and ***can_place_agents***, for querying an user-friendly name of the ***resource***; ***name***.

We may also wish to mirror the design of the executors proposal and have a generic query interface using properties for querying information about an ***resource***. It's expected that an implementation may provide additional non standard queries that are specific to that implementation.

| | |
|---|---|
| **Straw Poll** | Should the interface allow an execution resource to place memory, place agents or both?<br><br>*Is what is defined here a suitable solution?* |

**Querying the topology**

The interface for querying the ***resource topology*** of a ***system*** must be flexible enough to allow both querying all ***execution resources*** available under an ***execution context*** and querying the ***execution resources*** available to the entire system and constructing an ***execution context*** for a particular ***execution resource***. This important as many standards such as OpenCL [20] and HSA [19] require the ability to query the ***resource topology*** available in a ***system*** before constructing an ***execution context*** for executing work.

> *For example an implementation may provide an execution context for a particular execution resource such as a static thread pool or a GPU context for a particular GPU device or an implementation may provide a more generic execution context which can be constructed from a number of CPU and GPU devices queryable through the system resource topology.*

Below is a proposed interface for querying a ***system*** for its ***resource topology***.

```
namespace std::this_system {
  std::vector<execution_resource &> resources();
}
```

*Listing 3: Interface for querying the execution resources available within a system*

The ***resources*** function in the ***this_system*** namespace will return all ***execution resources*** available to the current system.

Below is an example of the interface for querying the ***execution resources*** available to the entire system and printing out the names of each ***execution resource***.

```
auto &resources = std::execution::this_system::resources();

for (auto &r : resources) {
    std::cout << r.name()  << std::endl;
}
```

*Listing 4: Example of querying the execution resources available within a system*

| **Straw Poll** | Should the interface provide a way of querying the system topology directly?<br><br>*Is what is defined here a suitable solution?* |
|---|---|

Below is a proposed extension to the ***execution context*** interface to allow an ***execution context*** to be constructed from an ***execution resource***.

```
struct execution_context {
  ...

  template <typename ExecutionResource>
  execution_context(ExecutionResource &&execResource);

  ...
};
```

*Listing 5: Extension to execution_context interface*

The ***execution context*** constructor described above allows constructing an ***execution context*** from any ***execution resource*** within a ***system's resource topology***. The constructed ***execution context*** can then execute work on any resource under that ***execution resource***.

Below is an example of how this extended interface could be used to construct an ***execution context*** from an ***execution resource*** which is retrieved from the ***system's resource topology***.

Once an execution context is constructed it can then still be queried for its execution resource and then that execution resource can be further partitioned.

```
auto &resources = std::execution::this_system::resources();

std::execution::execution_context execContext(resources[0]);

auto &execResource = execContext.resource();

// resource[0] should be equal to execResource

for (int i = 0; i < resource.partition_size(); i++) {
    std::cout << resource.partition(i).name()  << std::endl;
}
```

*Listing 6: Example of constructing an execution context from an execution resource*

| **Straw Poll** | Should the interface provide a way of creating an execution context from an execution resource?<br><br>*Is what is defined here a suitable solution?* |
|---|---|

## Binding Execution and Allocation to a Partition

When creating an ***execution context*** from a given ***execution resource***, the executors and allocators associated with it are bound to that ***execution resource***. For example: when creating an ***execution resource*** from a CPU socket resource, all executors associated with the given socket will spawn execution agents with affinity to the socket partition of the system.

```
auto cList = std::execution::this_system::resources();
// FindASocketResource is a user-defined function that finds a
// resource that is a CPU socket in the given resource list
auto& socket = findASocketResource(cList);
execution_contexteC{socket} // Associated with the socket
auto executor = eC.executor(myFunctor); // By transitivity, associated with the socket too
auto socketAllocator = eC.allocator(); // Retrieve an allocator to the closest memory node
std::vector<int, socketAllocator> v1(100);
std::generate(par.on(executor), std::begin(v1), std::end(v1), std::rand);
```

*Listing 8: Example of allocating with affinity to an execution resource*

The construction of an ***execution context*** on a component implies affinity (where possible) to the given resource. This guarantees that all executors created from that ***execution context*** can access the resources and the internal data structures requires to guarantee the placement of the processor.

Only developers that care about resource placement need to care about obtaining executors and allocations from the correct ***execution context*** object. Existing code for vectors and STL (including Parallel STL interface) remains unaffected.

If a particular policy or algorithm requires to access placement information, the resources associated with the passed executor can be retrieved via the link to the ***execution context***.

**Importance of topology discovery**

For traditional single CPU systems the execution resources reasoned about using standard constructs such as std::thread, std::this_thread and thread local storage. This is because the C++ memory model requires that  a system have **at least one thread of execution, some memory and some I/O capabilities.** This means that for these systems some assumptions can be made about the topology could be made during at compile-time, for example the fact that developers can query always the hardware concurrency available as there is always at least 1 thread or the fact that you can always use thread local storage.

This assumption, however, does not hold on newer more complex systems, and is particularly false in heterogeneous systems. In these systems, the even the available high level resources such as the number and type of devices available in a particular ***system*** is not known until the ***system's resource topology*** has been discovered which often happens as part of a runtime API [19] [20]. Furthermore the level of support these for querying the resource topology these devices may vary. This means the previous assumption that you can query thread concurrency at any stage of the program or the availability of a ***std::thread*** with local storage is no longer valid: Different devices may have different capabilities.

An interesting question which arises here is whether the system topology of an execution resource should be fixed on initialisation or allowed to be dynamic. Allowing a dynamic system topology allows components to go offline and become unavailable at runtime. If we do allow the system topology to be dynamic then we will need to provide a mechanism by which users can be notified of a topology change. However, providing this interface is out of the scope of this initial document.

Note that this is different from devices that go online or offline during execution: The devices themselves are online, they have not been found (or used) by the program until the appropriate discovery stage has been executed.

| | |
|---|---|
| **Straw Poll** | Should the interface allow a system's resource topology to be updated dynamically after initial initialisation?<br><br>*When do we enable the device discovery process? Can we change the system topology after executors have been created?*<br><br>*Should be provide an interface for providing a call-back on topology change?* |

**Lifetime considerations**

As the execution context would provide a partitioning interface which returns objects describing the components of the system topology of an execution resource it's important to consider the lifetime of these objects.

The objects returned from the partitioning interface would be opaque implementation defined objects which do not perform any scheduling or execution functionality which would be expected from an ***execution context*** and would not store any state related to an execution. Instead they would act simply as an identifier to a particular partition of the ***resource topology***.

For these reasons ***resources*** must always outlive any ***execution context*** which is constructed from them and any ***resource*** retrieved from an ***execution context*** must not be tied to the lifetime of that ***execution context***.

**Scaling to heterogeneous and distributed systems**

The initial solution should target systems with a single addressable memory region, i.e. a system which does not have discrete non-accessible memory regions such as a discrete GPU or FPGA. However in the interest of maintaining a unified interface going forward the initial solution should be designed with the latter in mind and should be scalable to support these systems in the future. In particular to support heterogeneous systems it's important that the abstraction allows the interface for querying the ***resource topology*** of the ***system*** in order to perform device discovery.

# Querying the Relative Affinity of Partitions

In order to make decisions about where to place execution or allocate memory in a given ***system's resource topology***, it is important to understand the concept of affinity between different ***execution resources***. This is usually expressed in terms of latency from resource a to b. Distance does not need to be symmetric in all architectures.

The relative position of two components in the topology is not necessary and indicative of their affinity. For example, two cores from two different CPU sockets may have the same latency to access to the same NUMA memory node.

| | |
|---|---|
| **Straw Poll** | Should the interface allow users to query the relative affinity between two execution resources?<br><br>*Do we want to implement a complete interface for affinity querying on C++ or do we leave this for library vendors?*<br><br>*Do we need to define terms such as latency on the C++ standard?*<br><br>*What should such an interface look like and should it be quantifiable?*<br><br>*Do we consider enough to show the number of "hops" for data to move from one resource to the other?* |

**Scaling to heterogeneous and distributed systems**

This feature could be easily scaled to heterogeneous and distributed systems as the relative affinity between components can apply to discrete heterogeneous and distributed systems as well.

# Future Work

**Migrating data from memory allocated in one partition to another**

In some cases for performance it is important to bind a memory allocation to a memory region for the duration of an a tasks execution, however in other cases it's important to be able to migrate the data from one memory region to another. This is outside the scope of this paper, however we would like to investigate this in a future paper.

| | |
|---|---|
| **Straw Poll** | Should the interface provide a way of migrating data between partitions? |

**Defining memory placement algorithms or policies**

With the ability to place memory with affinity comes the ability to define algorithms or memory policies which describe at a higher level how memory is distributed across large systems. Some examples of these are pinned, first touch and scatter. This is outside the scope of this paper, however we would like to investigate this in a future paper.

| **Straw Poll** | Should the interface provide standard algorithms or policies for distributing memory? |
|---|---|

# References


[1] P0234r0 Towards Massive Parallelism (aka Heterogeneous Devices/Accelerator/GPGPU) support in C++ with HPX
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0234r0.pdf

[2] P0236r0 Khronos's OpenCL SYCL to support Heterogeneous Devices for C++
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0236r0.pdf

[3] P0362r0 Towards support for Heterogeneous Devices in C++ (Concurrency aspects)
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0362r0.pdf

[4] P0363r0 Towards support for Heterogeneous Devices in C++ (Language aspects)
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0363r0.pdf

[5] P0443r2 A Unified Executors Proposal for C++
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0443r2.html

[6] P0567r1 Asynchronous Managed Pointer for Heterogeneous and Distributed Computing
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0567r1.html

[7] P0687r0: Data Movement in C++
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0687r0.pdf

[8] P0737r0: Execution Context of Execution Agents
https://github.com/kokkos/ISO-CPP-Papers/blob/master/P0737_ExecContext.rst

[9] Portable Hardware Locality
https://www.open-mpi.org/projects/hwloc/

[10] cpuaff
https://github.com/dcdillon/cpuaff

[11] OpenMP 5 Technical Report
http://www.openmp.org/wp-content/uploads/openmp-TR5-final.pdf

[12] Persistent Memory Programming
http://pmem.io/

[13] MEMKIND
https://github.com/memkind/memkind

[14] The Design of OpenMP Thread Affinity
https://link.springer.com/chapter/10.1007/978-3-642-30961-8_2

[15] Solaris pbind()
https://docs.oracle.com/cd/E26502_01/html/E29031/pbind-1m.html

[16] Linux sched_setaffinity()
https://linux.die.net/man/2/sched_setaffinity

[17] Windows SetThreadAffinityMask()
https://msdn.microsoft.com/en-us/library/windows/desktop/ms686247(v=vs.85).aspx

[18] Exposing the Locality of new Memory Hierarchies to HPC Applications
https://docs.google.com/viewer?a=v&pid=sites&srcid=bGJsLmdvdnxwYWRhbC13b3Jrc2hvcHxneDozOWE0MjZjOTMxOTk3NGU3

[19] HSA Foundation
http://www.hsafoundation.com/standards/

[20] OpenCL 2.2
https://www.khronos.org/registry/OpenCL/specs/opencl-2.2.pdf

[21] Affinity Matters
Euro-Par 2011 Parallel Processing: 17th International

[22] Portable Hardware Locality lstopo
https://www.open-mpi.org/projects/hwloc/lstopo/