SYCLOPS

Deliverable 4.3 – hipSYCL graph runtime & cross-device scheduling

GRANT AGREEMENT NUMBER: 101092877

Project acronym:	**SYCLOPS**
Project full title:	**Scaling extreme analYtics with Cross architecture acceLeration based on OPen Standards**
Call identifier:	**HORIZON-CL4-2022-DATA-01-05**
Type of action:	**RIA**
Start date:	**01/01/2023**
End date:	**31/12/2025**
Grant agreement no:	**101092877**

D4.3 – hipSYCL graph runtime & cross-device scheduling

Executive Summary: This deliverable outlines the work done in "Task 4.3: hipSYCL Graph Runtime" in WP4 of the SYCLOPS project that aimed to develop a graph-based runtime to enable the efficient execution of SYCL applications across heterogeneous processors. This work resulted in two key achievements. First, AdaptiveCpp added support for offloading calls to C++ parallel Standard Template Library (STL) algorithms, such as std::for_each and std::transform_reduce. Second, AdaptiveCpp implemented the MQS feature that automatically detects dependencies between kernels and intelligently distributes execution across different queues or devices. This makes AdaptiveCpp MQS the first framework capable of automatically utilizing multiple GPUs for programs written purely in standard C++. Overall, the development successfully achieved the goal of providing automatic multi-GPU programming and scheduling without requiring dedicated programming models, representing a major step forward for the programmability of multi-GPU systems in C++.

WP:	4		
Author(s):	Aksel Alpay		
Editor:	Raja Appuswamy		
Leading Partner:	UHEI		
Participating Partners:			
Version:	1.0	**Status:**	Draft
Deliverable Type:	Other	**Dissemination Level:**	PU
Official Submission Date:	06-Oct-2025	**Actual Submission Date:**	30-Sep-2025

Disclaimer

The SYCLOPS consortium consists of the following partners:

No.	Partner Organisation Name	Partner Organisation Short Name	Country
1	EURECOM	EUR	**FR**
2	INESC ID - INSTITUTO DE ENGENHARIADE SISTEMAS E COMPUTADORES, INVESTIGACAO E DESENVOLVIMENTO EM LISBOA	INESC	**PT**
3	RUPRECHT-KARLS-UNIVERSITAET HEIDELBERG	UHEI	**DE**
4	ORGANISATION EUROPEENNE POUR LA RECHERCHE NUCLEAIRE	CERN	**CH**
5	HIRO MICRODATACENTERS B.V.	HIRO	NL
6	ACCELOM	ACC	FR
7	CODASIP S R O	CSIP	CZ
8	CODEPLAY SOFTWARE LIMITED	CPLAY	UK

Document Revision History

Version	Description	Contributions
0.1	Structure and outline	EUR
0.2	DPC++ technical contributions	CPLAY
0.3	AdaptiveCPP technical contributions	UHEI
1.0	Final draft	EUR

Authors

Author	Partner
Aksel Alpay	UHEI

Reviewers

Name	Organisation
Aleksandar Ilic	INESC
Vincent Heuveline	UHEI
Danilo Piparo	CERN
Nimisha Chaturvedi	ACC
Martin Bozek	CSIP

Statement of Originality

This deliverable contains original unpublished work except where clearly indicated otherwise. Acknowledgement of previously published material and of the work of others has been made through appropriate citation, quotation or both.

Table of Contents

Executive Summary

This deliverable outlines the work done in “Task 4.3: hipSYCL Graph Runtime” in WP4 of the SYCLOPS project that aimed to develop a graph-based runtime to enable the efficient execution of SYCL applications across heterogeneous processors. This work resulted in two key achievements.

1. **Highly Competitive Stdpar Implementation:** AdaptiveCpp added support for offloading calls to C++ parallel Standard Template Library (STL) algorithms, such as std::for_each and std::transform_reduce. This implementation generates code that is often highly competitive or outperforms vendor compilers (NVIDIA’s nvc++, AMD’s roc-stdpar, Intel’s icpx/DPC++) across tested mini-apps. Crucially, AdaptiveCpp introduced an optimization to detect and elide unnecessary synchronization (barriers), which is unique among known stdpar compilers and essential for performance in latency-sensitive applications like LULESH.

2. **Multi-Queue Scheduling (MQS):** AdaptiveCpp implemented the MQS feature to automatically leverage multiple devices. MQS automatically detects dependencies between kernels and intelligently distributes execution across different queues or devices. This makes AdaptiveCpp MQS the first framework capable of automatically utilizing multiple GPUs for programs written purely in standard C++. MQS demonstrated excellent scaling (over 3.7x speedup using four NVIDIA GPUs in ideal cases like BabelStream triad) while incurring minimal overheads (typically within 1–5%).

Overall, the development successfully achieved the goal of providing automatic multi-GPU programming and scheduling without requiring dedicated programming models, representing a major step forward for the programmability of multi-GPU systems in C++. All work done on AdaptiveCpp has been made publicly available in its Github repository.

1. Introduction

Figure 1 shows the SYCLOPS hardware-software stack consists of three layers: (i) infrastructure layer, (ii) platform layer, and (iii) application libraries and tools layer.

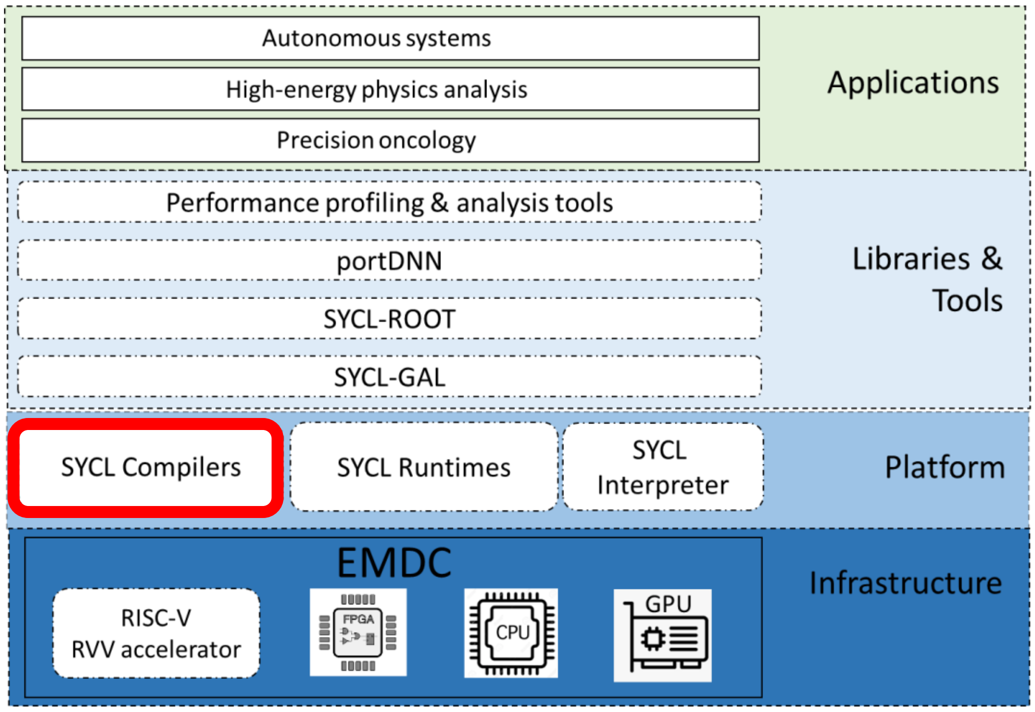


Figure 1. SYCLOPS architecture

Infrastructure layer: The SYCLOPS infrastructure layer is the bottom-most layer of the stack and provides heterogeneous hardware with a wide range of accelerators from several vendors.

Platform layer: The platform layer, provides the software required to compile, execute, and interpret SYCL applications over processors in the infrastructure layer. The second layer from the bottom, the platform layer, provides the software required to compile, execute, and interpret SYCL applications over processors in the infrastructure layer. SYCLOPS will contain oneAPI DPC++ compiler from CPLAY, and AdaptiveCpp, formerly known as hipSYCL, an open-source SYCL compiler toolchain from UHEI. In terms of SYCL interpreters, SYCLOPS will contain Cling from CERN.

Application libraries and tools layer: The libraries layer enables API-based programming by providing pre-designed, tuned libraries for various deep learning methods for the PointNet autonomous systems use case (SYCL-DNN), mathematical operators for scalable HEP analysis (SYCL-ROOT), and data parallel algorithms for scalable genomic analysis (SYCL-GAL).

This deliverable concerns the **SYCL compilers** part of the stack as highlighted in Figure 1, and covers the work done on the AdaptiveCPP compiler in the context of "*Task 4.2: SYCL Graph Runtime*" in *WP4* (M3-M33). The objective of this task was to develop a graph-based SYCL runtime that will enable efficient execution of SYCL applications across several heterogeneous processors.

This deliverable is structured as follows. Section 1 of this deliverable provides a high-level overview of the overall SYCLOPS architecture and positions this deliverable with respect to both components in the SYCLOPS stack and WP/tasks in the work plan. Section 2 provides a background by introducing the problem, the overall strategy of how the problem was tackled, and why C++ standard parallelism offloading was selected as programming model. Section 3 presents the support for C++ standard parallelism offloading that was added in the project, and shows that it highly competitive compared to vendor solutions, often outperforming them. Finally, Section 4 presents the multi-queue scheduling (MQS) framework that we have implemented on top of C++ standard parallelism, and showcases its performance using a number of benchmarks.

2. Background and Motivation

The goal of this work package was to enable automatic cross-device scheduling of kernels for the purpose of simplified multi-device programming. Independent kernels are to be scheduled and distributed automatically, however, every individual kernel is expected to still execute only on one device.

Such solutions already exist (e.g. StarPU), however, these approaches generally require the program to be written in custom programming models for the specific purpose of cross-device scheduling. This additional porting effort counteracts the desired increased simplicity for the programmer.

As a consequence, the motivation behind this work was that SYCL already provides an abstract data storage that is not tied to a specific device (the buffer-accessor model) and the ability to construct graphs of kernels in programs. Therefore, the idea arose that standard SYCL might be sufficient to express the desired functionality.

With such an approach, unlike existing solutions, SYCL programs would not have to be ported just to benefit from automatic cross-device scheduling, thus avoiding the additional porting effort.

Over the course of the project, several key insights were obtained that have led to a refinement of that approach, while still retaining the original goal of simplified muti-device scheduling without leveraging dedicated programming models for that purpose:

1. New code investments in SYCL generally strongly prefer the pointer-based USM (unified shared memory) model over the older buffer-accessor model, mainly due to performance concerns with the buffer-accessor model. For example, the widely used GROMACS molecular dynamics software has switched to USM after initially implementing SYCL support using the buffer-accessor model. There is no interoperability in SYCL between buffers and USM pointers. Therefore, in order to retain the goal of minimizing porting effort and maintain practical relevance, it was decided that our solution should not rely on the buffer-accessor model.

2. Submission of work in SYCL is expressed by submitting kernels to a queue. In the SYCL runtime model, queues are always tied to a single device. This also affects the API, e.g. there is a `queue::get_device()` member function. This function is of key importance for the USM model, because applications will use it to specify the data location when allocating USM memory. Experiments were conducted to add a multi-device queue; however, it was found that only a subset of the queue functionality could be meaningfully supported due to assumptions in the API about a 1:1 match between queues and devices. This subset also was not easy to clearly define and would likely have been confusing to users. Therefore, it was concluded that a more high-level, queue-less kernel submission mechanism would be the cleanest solution.

A queue-less submission mechanism for data parallel kernels is however roughly equivalent to the semantics of the existing `std::for_each` function from standard C++. Standard C++ also does not impose device-specific memory management, and the high-level, device-unaware nature of standard C++ allows for transparent multi-device scheduling.

Standard C++ parallelism is also already established as a model for offloading to accelerators (see e.g. NVIDIA's nvc++ compiler). This relies on detecting calls to C++ parallel standard template library (PSTL) algorithms, which are then offloaded and executed e.g. on a GPU. Offloading standard C++ parallel STL algorithms is often referred to as the *stdpar model* (standard parallelism).

Therefore, instead of implementing the functionality directly in SYCL, exposing the desired functionality at the higher-level of stdpar (which might then in turn rely on SYCL) is highly attractive. It also provides the additional advantage that the functionality would be even more accessible to end users, since standard C++ code is sufficient to leverage it – porting code to SYCL will not be necessary.

We also note that the SYCL specification builds on top of standard C++, and includes the C++ specification. Therefore, any valid standard C++ program (such as an stdpar program) is also a valid SYCL program.

Due to all these considerations, it was decided to add support for the stdpar model to AdaptiveCpp, and expose the multi-device scheduling functionality there.

In the following, we will first discuss how the stdpar model was implemented as a foundation, before then discussing how the automatic cross-device scheduling was implemented on top of stdpar. The latter is accomplished as part of AdaptiveCpp's new multi-queue scheduling (MQS) feature.

Our developments render AdaptiveCpp the very first solution that can automatically distribute code that is written simply in standard C++ across multiple GPUs, and it is also the first stdpar compiler able to utilize multiple GPUs from a single thread of execution.

3. Stdpar support in AdaptiveCpp

To facilitate this work (and with requests from users), we have added support for offloading calls to C++ parallel STL algorithms on top of our SYCL compiler and runtime infrastructure. This allows users to formulate their program in terms of high-level C++ algorithms such as `std::for_each`, `std::copy` or `std::transform_reduce`. Figure 2 illustrates the model at the example of a vector addition. The execution policy `std::execution::par_unseq` asserts to the compiler that the algorithm may safely be both parallelized and vectorized. This assertion also qualifies the algorithm for offloading to an accelerator.

Note that the code is standard C++ that can also be translated with any regular C++ compiler, even though an stdpar-aware compiler might leverage a GPU for acceleration.

```
#include <algorithm>
#include <execution>

void add(const std::vector<float>& a, const std::vector<float>& b
        std::vector<float>& c_out) {
  std::transform(std::execution::par_unseq,
                a.begin(), a.end(), b.begin(), c.begin(),
  [](auto v1, auto v2){
    return v1+v2;
  });
}
```

Figure 2: Vector addition in the stdpar model.

The AdaptiveCpp standard C++ algorithms are supported with our generic JIT compiler on all backends, including OpenCL via the construction kit, and thus also on RISC-V hardware. Our publication on the matter [1] discusses the implementation in detail.

Because C++ has a flat memory model, it is unaware that there might be multiple devices in the system with distinct memory spaces. Therefore, a primary challenge for the C++ standard parallelism model for offloading is that of making all system memory device-accessible. In some cases, e.g. when host and device are tightly integrated and share the same physical memory, or Linux HMM (heterogeneous memory management) is enabled, this might just work out of the box. In other cases, the compiler and runtime will have to remap all memory allocation and deallocation requests to device-aware variants. AdaptiveCpp does this by remapping allocations to `sycl::malloc_shared (unified shared memory, USM)`, which returns allocations that may automatically migrate between host and device. This remapping is non-trivial, and we refer to [1] for details.

AdaptiveCpp employs additional optimizations that are not implemented by C++ standard parallelism offloading compilers from hardware vendors (NVIDIA's nvc++, AMD's roc-stdpar, Intel's icpx/DPC++). This includes automatically emitting prefetch operations to migrate data used by kernels or an offloading heuristic that attempts to estimate whether it is worth offloading a C++ standard algorithm on the device, as opposed to the host.

In standard C++, code can assume that as soon as a parallel STL algorithm call returns, any changes made to memory are in principle immediately visible. In an offloading scenario, where parallel STL calls are mapped to kernel launches, this implies having to wait on the host after every parallel STL calls until the enqueued kernels have finished. In latency-sensitive code, this can cause severe performance overheads. Additionally, it makes it impossible to obtain performance improvements by distributing kernels across different GPUs, since there can be no overlap of different kernels.

Therefore, we have added the ability to the AdaptiveCpp compiler to detect and elide unnecessary barriers. For example, if there are two subsequent `std::for_each` calls, then a barrier in between them might not needed. This optimization is implemented by delaying synchronization as far as possible across all paths in the control flow graph, and then merging successive barrier calls. A detailed discussion can be found in [1]. AdaptiveCpp is, to our knowledge, the only stdpar compiler supporting this optimization.

Because in standard C++ parallelism math functions or other functionality like atomics needs to work in terms of functionality from the std:: C++ namespace, not the sycl:: namespace, we have added additional compiler transformations that remap C++ std math functions to AdaptiveCpp math builtins and std atomic functionality to AdaptiveCpp atomic builtins. This allows using std:: math builtins and std::atomic or std::atomic_ref to be used in device code.

Figure 3 illustrates the AdaptiveCpp standard C++ parallelism performance for three mini-apps relative to the respective hardware vendor compilers and their support for standard C++ parallelism offloading (nvc++, hipstdpar, icpx -fsycl-pstl-offload). The red line indicates performance parity, and the blue lines performance within 20%. XNACK refers to a hardware feature on AMD GPUs that is required to enable proper support for automatically migrating memory between host and device. Without, the ROCm stack runs in a degraded mode. In our experience, XNACK is rarely available on most production systems because it is not supported by all AMD GPUs, and requires non-standard Linux kernel boot parameters.

AdaptiveCpp outperforms the vendor compilers on all platforms for two out of three mini-apps, sometimes by an order of magnitude. As can be seen from the results, it is also less dependent on the availability of XNACK on AMD hardware.

Overall, the results indicate that AdaptiveCpp can generate highly competitive code on all three platforms in the standard C++ parallelism model, and is the only standard parallelism solution that can target them all robustly. These results were obtained using the generic JIT compiler on all hardware platforms.

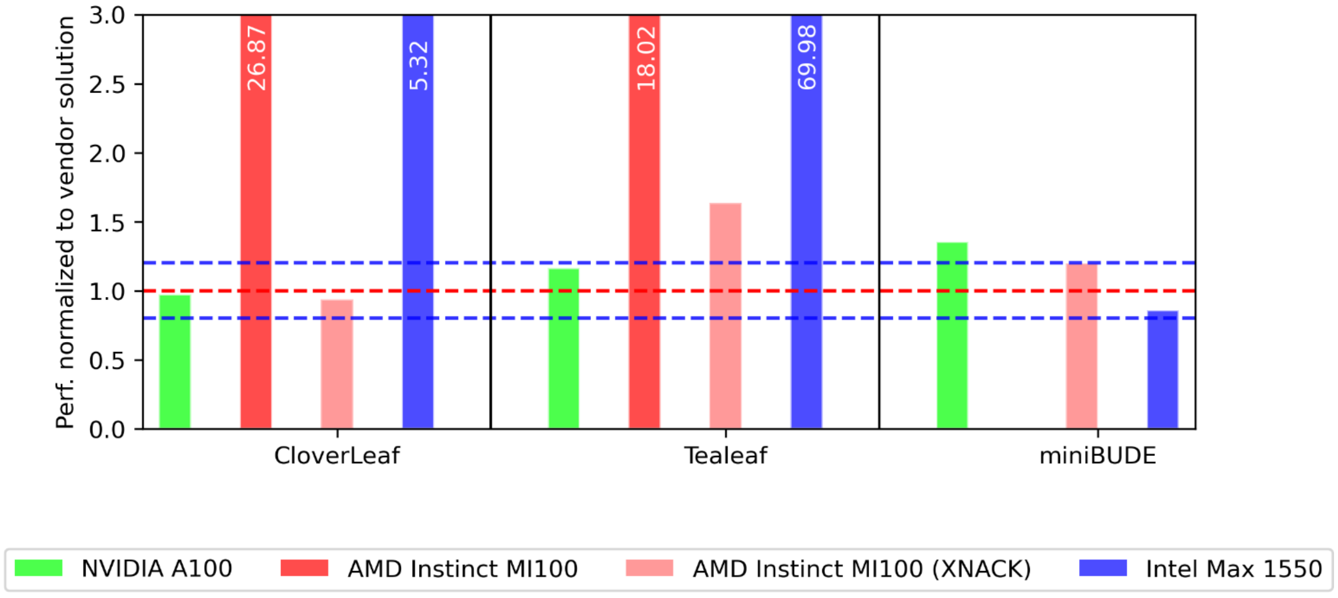


Figure 3: AdaptiveCpp stdpar speedup over vendor compiler

Figure 4 shows the performance AdaptiveCpp and nvc++ stdpar for the LULESH benchmark on NVIDIA A100 for various problem sizes. To the left, the speedup of nvc++ and AdaptiveCpp offloading compared to using the host PSTL can be seen. On the right, AdaptiveCpp's speedup over nvc++ is shown. In both plots, AdaptiveCpp substantially outperforms NVC++. For large problems, this is caused by an issue in the memory pool used by nvc++. For small problems, LULESH becomes highly latency-bound, and AdaptiveCpp's speedup is primarily caused by its ability to elide unnecessary synchronization. This demonstrates the importance of this optimization for performance.

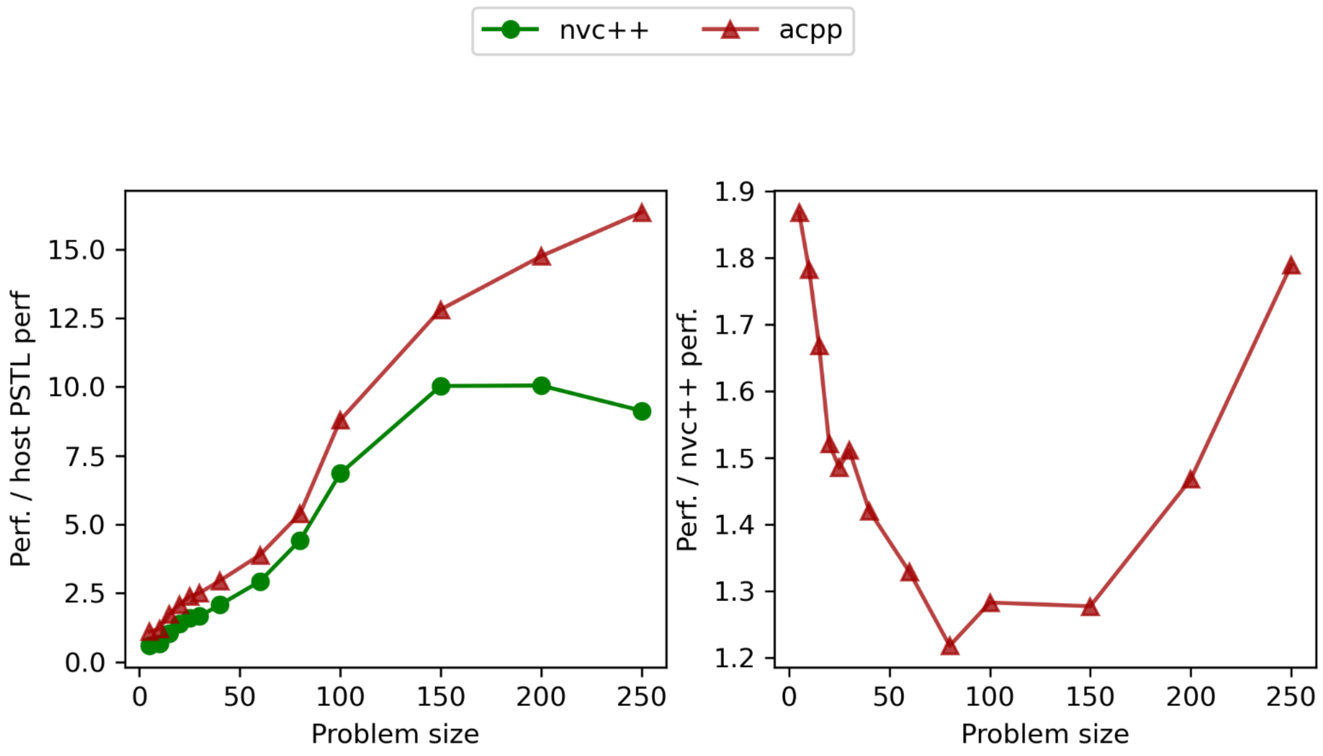


Figure 4: AdaptiveCpp and nvc++ Performance for LULESH on NVIDIA A100

Note that the stdpar model can also be used to implement more complex code patterns. For example, in [4], we have shown how it can be leveraged to implement nbody tree codes (including parallel tree construction), and demonstrated the competitiveness of both the model as well as AdaptiveCpp on hardware from multiple vendors.

4. Multi-queue scheduling (MQS)

In order to leverage multiple devices automatically, the multi-queue scheduling (MQS) feature was implemented in AdaptiveCpp. MQS can automatically detect dependencies between kernels, and distribute the execution of kernels across different queues, which may reside on the same device (for improved utilization) or on other devices.

The stdpar model does not provide explicit APIs for users to specify dependencies between kernels. Therefore, the compiler needs to figure out dependencies between kernels on its own. In AdaptiveCpp MQS, this is solved as follows:

1. a new compiler pass attempts to prove that a kernel does not perform indirect access. Indirect access refers to the situation when pointers are loaded from memory, for example if a linked-list data structure is used, and only the first node is passed to the kernel as direct argument. If indirect access is present, the compiler and runtime can in general no longer infer which allocations a given kernel might access. Consequently, in such a case the runtime must conservatively assume that the kernel might depend on all previous kernels. Such a scenario currently causes MQS to fall back to normal, single-device synchronous kernel submission. In practice however, indirect access is rare for many scientific computing workloads. Indirect access typically occurs in conjunction with complex, pointer-based data structures which are rarely used in the data parallel kernels that one might choose to express in the stdpar model.

2. Due to AdaptiveCpp's just-in-time compilation design, the generated kernel code is unavailable until the kernel is actually invoked. In order to be able to retrieve information about kernel properties such as indirect access, a per-application database (appdb) is maintained that records such information. Future application runs can then – already for the first kernel run – retrieve the needed data from the appdb.

3. AdaptiveCpp adds an additional memory management layer. This layer not only ensures that every memory allocation is GPU-accessible (in standard C++, i.e. the stdpar model, there is only a single memory space), it also tracks the locality of every allocation by recording on which device and on which queue an allocation was utilized last. It also allows to efficiently find the allocation to which any given pointer points to, even if the pointer is offset to the allocation base.

4. For every pointer kernel argument, AdaptiveCpp retrieves the corresponding allocation. A dependency is set up to the queues that have used this allocation previously. This effectively organizes kernels in a task graph by the allocations that they access. In theory, this might pessimize the dependency graph if not the entire allocation is actually used in the kernel. However, proving in the compiler which parts are actually accessed in the kernel is highly non-trivial and likely to return pessimized results as well to ensure correctness when in doubt. Additionally, a scheme that depends on dependency management below allocation granularity also must utilize much more complex data structures to maintain e.g. locality information, which can have an impact on runtime overheads.

In order to actually benefit from scheduling to multiple devices, work must have been enqueued asynchronously, that is, the synchronization elision optimization that was described in the previous section must have succeeded. Otherwise, the implicit barrier after every kernel launch prevents any overlap of kernels running on different devices.

In theory, one could wait for a dependency graph to build up and only lazily execute the graph at the point when synchronization is actually triggered. This has the advantage of having more information available for scheduling decisions, and many existing cross-device scheduling frameworks rely on this approach. However, for latency-bound applications, this method can substantially worsen performance because kernel execution can no longer be overlapped with host work. Therefore, since AdaptiveCpp aims to be a general purpose solution, MQS utilizes an eager submission design. This is sufficient for common cases, as we show later.

When a kernel is submitted and the MQS subsystem can prove the dependencies of the kernel as described above, it initiates the multi-queue scheduling logic. On every device, a predefined number of in-order queues (by default 4) are maintained. Consequently, MQS might obtain performance improvements by scheduling to a different device, or by scheduling to a different queue on the same device (potentially leading to concurrent execution of kernels).

The target queue is picked based on the accumulated estimated cost due to multiple effects:

1. The cost of the previously enqueued work to the same queue and device, and the dependencies of the kernel. These delay the expected finish time. This generally favors distributing work to other devices, and if that is not possible, to other queues.
2. The cost of migrating data. To this end, AdaptiveCpp estimates data transfer time using interconnect speeds and the overall size of the allocations used in a kernel. This favors keeping kernels on the same device.
3. Other minor effects, such as the small runtime overheads when synchronization constructs between different queues need to be inserted into the execution. If device load is still low, this causes the scheduler to submit a kernel to the same queue, if all its dependencies are also on that queue.

The cost of the kernel itself is estimated using a simplistic performance model, and the expected finish time is recorded. The estimated finish time is recorded and then potentially utilized for the cost calculation of the next kernel launch (e.g. as part of device load, queue load, or dependency cost estimate).

In practice, we found that the performance model does not need to be particularly sophisticated. In stdpar, due to the reliance on the synchronization elision optimization, submitted dependency graphs tend to be far smaller than in explicitly asynchronous models like SYCL. As such, cumulative errors in performance estimates generally play a lesser role. Additionally, in many cases, the decision is very clear as long as the right order of magnitude of the cost is estimated. In cases where the kernel cost and device load cost are very similar to the data transfer cost, even a better performance model might not help since there are many effects that cannot be modelled easily. For example, the kernel's control flow might depend on input values, or the actual finish time might depend on how the hardware schedules multiple kernels on different queues (will they be executed sequentially or concurrently, and in the latter case, how much will the resource sharing slow down individual kernels?).

When evaluating MQS, we found that many existing benchmarks do not lend themselves to MQS. This is not due to a deficiency in our work, but a simple consequence of the fact that many benchmarks only rely on few, or even just a single kernel. In this case, there are simply not enough independent kernels to distribute across multiple GPUs. A solution like MQS that can distribute kernels in a dependency graph across devices needs the program to expose a sufficiently wide dependency graph. Similarly, many benchmarks rely on iterative methods. In such a scenario, every kernel typically depends in a linear dependency chain on previous kernels. Again, such problems just do not expose a sufficient number of independent kernels that may be executed in parallel.

This problem is not new, and is one that programmers also face e.g. in the field of distributed computing with MPI. There, the problem is solved by decomposing the domain into multiple subdomains, and then invoking the same algorithms on independent subdomains.

The same strategy can be applied with MQS to generate more independent work if necessary – with the important distinction that the programmer does not need to explicitly assign the work to a device, they just need to ensure that sufficient independent work is available in the first place. Another reason for the absence of meaningful benchmarks that lend themselves to MQS is also that a solution with these capabilities for stdpar programs has so far simply not yet existed. Since applications and benchmarks are also constructed around opportunities that compilers and runtimes provide, applications have (rightly) made no effort so far to ensure that many independent kernels are available. If techniques like e.g. domain decomposition appear in benchmarks, they generally only do so in conjunction with MPI.

To overcome this, we have made some slight modifications to some existing benchmarks to exploit MQS better, and we have also developed new benchmarks. In this study, we have used the following benchmarks:

1. BabelStream [2], a memory benchmark, has been modified by decomposing the array into multiple subdomains on which a separate kernel is then invoked. Most of BabelStream's kernels are completely independent. It thus serves as a demonstration for best case scalability.

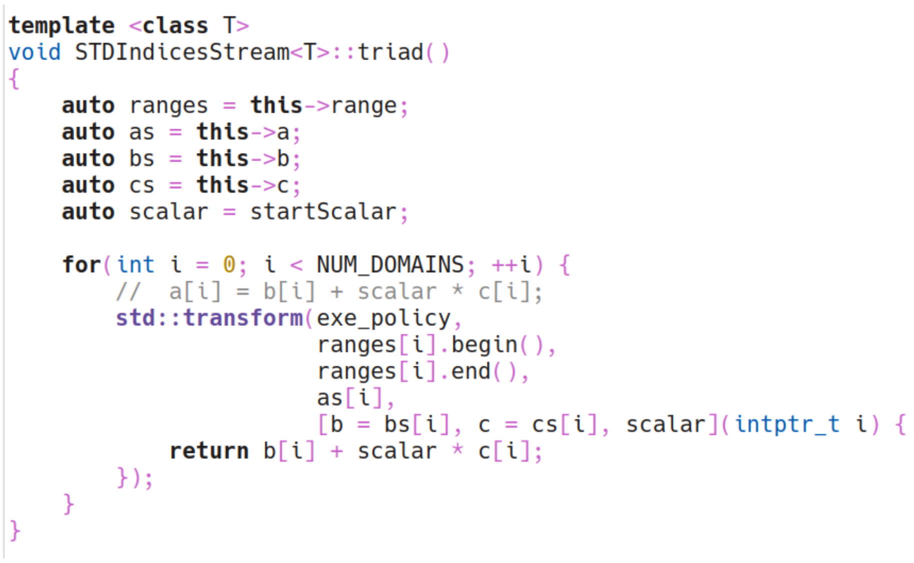
2. Mandelbrot: In this benchmark, the output image is decomposed into stripes, and the Mandelbrot fractal is calculated for each stripe. Then, a final gather kernel is launched to assemble the final image from the stripes. This benchmark tests the common case where independent work is followed by a gather-step.

3. Btree-search: Here, multiple parallel searches are conducted in a B-tree data structure. Every parallel search generates its own kernel sequence as the search descends into the tree. This tests the handling of pointer-based data structures with many small allocations and thus puts more pressure on the dependency management system.

4. Heat2d: This benchmark is based on the heat2d benchmark from the HeCBench suite [3] which we have modified to include domain decomposition. This mini-app solves the 2D heat equation using a simple stencil code. It is a proxy app for stencil codes with halo exchange along the domain boundaries, which is common for many scientific applications.

For illustrative purposes, Figure 5 shows the code patterns that were used in the BabelStream triad benchmark. The data arrays are split into one array per domain, and in a loop, one kernel per domain is submitted.

Note that `NUM_DOMAINS` does not need to match the number of GPUs present. It is simply a parameter to control the amount of independent kernels that are exposed to the MQS system. In practice, an application developer might e.g. set this to some maximum number of expected devices (most systems have 4 GPUs or less, but some rarer ones might have 8 or 16). The generated binary then will be able to fully leverage any system with devices up to the number of domains.



```
template <class T>
void STDIndicesStream<T>::triad()
{
    auto ranges = this->range;
    auto as = this->a;
    auto bs = this->b;
    auto cs = this->c;
    auto scalar = startScalar;

    for(int i = 0; i < NUM_DOMAINS; ++i) {
        //  a[i] = b[i] + scalar * c[i];
        std::transform(exe_policy,
                       ranges[i].begin(),
                       ranges[i].end(),
                       as[i],
                       [b = bs[i], c = cs[i], scalar](intptr_t i) {
            return b[i] + scalar * c[i];
        });
    }
}
```

Figure 5: Decomposition code pattern for triad benchmark in babelstream

The BabelStream case is particularly simple – here, a simple round-robin scheduling scheme might suffice. However, other benchmarks are more complex and require more elaborate scheduling logic. Heat2d e.g. first submits one kernel per domain to update the local values, and then submits multiple kernels to update the halos.

For all the tested application, MQS has selected the scheduling that would also be typically be picked by an experienced GPU programmer in an explicitly heterogeneous model: For the BabelStream pattern shown above, it distributes work similarly to a round-robin scheme. For mandelbrot, it distributes the stripes, and then performs the final gather step on the host. For heat2d, it runs the stencil kernels on the local domains in parallel on multiple GPUs, and then assigns the halo update kernels to the same GPU where the local data resides.

Figure 6 shows the speedup obtained by MQS over AdaptiveCpp with MQS enabled on one system with 4x NVIDIA Tesla V100, and another system with 4x NVIDIA A100. Results are shown for different numbers of GPUs exposed to the application using the `CUDA_VISIBLE_DEVICES` environment variable to simulate systems with different numbers of devices. The same application binary was used for all runs.

Results are normalized to the application compiled by AdaptiveCpp with MQS disabled. For reference, results using the NVC++ compiler from NVIDIA's HPC SDK are provided as well.

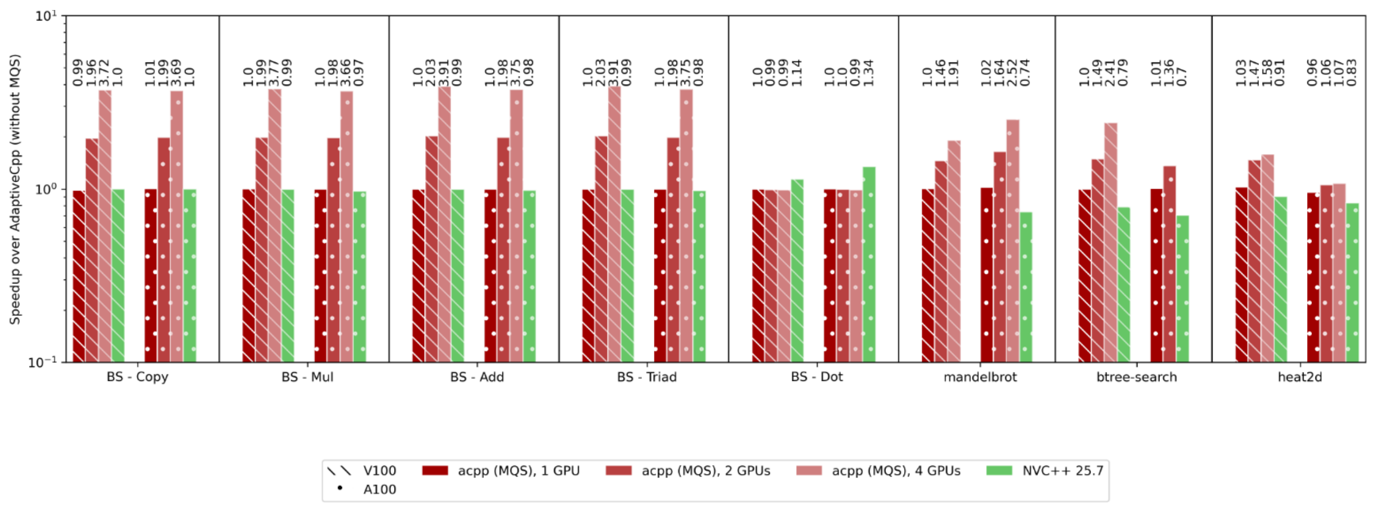


Figure 6: MQS speedup on two systems with 4x V100 and 4x A100 GPUs, respectively.

As expected, BabelStream scales very well, typically achieving a speedup of over 3.7x when using 4 GPUs. Only the dot kernel of BabelStream does not scale when enabling more GPUs.

This is because the dot benchmark uses `std::transform_reduce`, which is defined by the C++ standard to return the result of the reduction as return value of the function call. This however prevents synchronization elision, since AdaptiveCpp always needs to block until the kernel has completed so that it can return the result. While it still runs the kernels for the different domains on different devices for data locality reasons (the data was already distributed due to the other benchmarks, which babelstream runs before the dot kernel), the synchronization prevents overlapping of the execution on the different devices. This is not a defect in AdaptiveCpp MQS, but due to an unfortunate API in the C++ standard.

The mandelbrot kernel and btree-search benchmarks scale up to around 2.5x for four GPUs. This is expected for problems where data transfers play a role. On the A100 system, the results for the 4 GPU case of btree-search could not yet be obtained at the time of writing due to a driver issue on the node.

The heat2d benchmark scales less well. This is because the local stencil kernel is very small and simple, and runs very fast. Therefore, the data transfers between the GPUs during the halo exchange matter significantly. On the A100, this is worse because the local kernel runs even faster than on the V100. This effect depends on the problem size; a larger problem size causes better scaling since the halo exchange surface of the domain shrinks relative to the local domain as the problem size is increased.

Because the results are normalized to the performance of AdaptiveCpp with MQS disabled, the runs with one GPU can be used to estimate potential overheads of the MQS system. As can be seen from the plot, the MQS results with 1 GPU match the results of MQS disabled within 5% (and commonly within 1%). We conclude that overheads due to the MQS infrastructure will not be noticeable for typical applications.

We note that the MQS system is not device-dependent, and will work equally well on hardware from any vendor, if they provide a mature USM driver that can provide a single unified address space across all GPUs and efficient, implicit data transfers between all devices.

A current limitation is that MQS only works across devices from the same backend: For example, multiple CUDA GPUs work, bit mixing both AMD and NVIDIA hardware does not. This is primarily due to driver limitations. The flat memory model of standard C++ requires that every pointer must be accessible on host and all devices. When multiple backends are involved, this requires driver and hardware-level interoperability between different vendors (e.g., a pointer to an allocation is accessed on NVIDIA GPU, but the data is resident on an AMD GPU). This type of interoperability is not currently well-supported among drivers, but this might change in the future as more driver infrastructure is unified around technologies like e.g. Linux heterogeneous memory management (HMM).

In principle, MQS will work when different devices from the same backend are available (for example, multiple different models of CUDA GPUs). However, we did not focus on optimizing for this use case because it is not a common scenario in HPC systems. Additionally, the different performance of different GPUs generally requires a work balance scheme at the level of domain decomposition that MQS cannot control, because it cannot override how users invoke kernel submissions.

Overall, the results show that the MQS framework is able to efficiently leverage multiple devices in the same system automatically, if the program produces a sufficient amount of kernels that can be executed independently. It is the first solution able to accomplish that with just pure, standard C++. It therefore reaches the original goal of exposing multi-GPU programming and automatic multi-GPU scheduling without having to port the code to dedicated frameworks for that purpose. It is thus in our opinion a major step forward for the programmability of multi-GPU systems in C++.

5. Conclusion

This deliverable concludes the work done in "Task 4.3: hipSYCL Graph Runtime" of WP4 in SYCLOPS project. AdaptiveCpp MQS is the first framework that can automatically utilize multiple GPUs in a system for programs that are written in pure, standard C++. For the first time, this enables automatic multi-GPU scheduling without having to port the code to any explicit offload or graph scheduling APIs.

We have shown how AdaptiveCpp standard C++ offloading has been implemented, and how the MQS system was implemented on top of it, delivering speedups due to multi-device utilization. Performance scaled very well in ideal cases (e.g. BabelStream triad), and as expected for problems that included data transfers or communication. Nevertheless, there are many areas where the system could be improved further in the future. For instance, a key ingredient is the ability of the compiler to successfully elide unnecessary synchronization between stdpar kernels. There are however multiple cases where this might fail – e.g. there might be memory loads or stores in between the kernels where the optimizer cannot prove that they are not used inside kernels, or control flow might be too complex or escape the translation unit. The latter issue could be addressed by moving this compiler optimization pass to the link-time-optimization (LTO) stage. More elaborate pointer provenance analysis could help the optimization to be more robust with respect to loads/stores in between kernels.

Second, currently, the order of magnitude performance of typical PCIe interconnects is hardcoded. A more sophisticated approach would be to measure the interconnect between all device pairs, e.g. during the first run. This could be beneficial to adapt automatically to more complex topologies (e.g. systems where some pairs of devices have significantly faster interconnect than the rest) or systems that have particularly fast, vendor-specific interconnects (e.g. NVLink). This was not prioritized because compute time had not yet been granted on such systems at the time of writing.

Finally, the kernel performance model could be improved by taking into account runtime performance measurements. However, as mentioned previously, it is unclear to what extent a better performance model would even be beneficial, given that many other difficult-to-predict effects exist on accelerators.

All the work done on AdaptiveCpp has already been made publicly available in its Github repository[1]. We have also disseminated our work via technical blogs on the SYCLOPS website, and technical talks that can be found in the SYCLOPS Youtube channel.

[1] https://github.com/AdaptiveCpp/

References

[1] Aksel Alpay and Vincent Heuveline. 2024. AdaptiveCpp Stdpar: C++ Standard Parallelism Integrated Into a SYCL Compiler. In Proceedings of the 12th International Workshop on OpenCL and SYCL (IWOCL '24). Association for Computing Machinery, New York, NY, USA, Article 5, 1–12. https://doi.org/10.1145/3648115.3648117

[2] Deakin T, Price J, Martineau M, McIntosh-Smith S. Evaluating attainable memory bandwidth of parallel programming models via BabelStream. International Journal of Computational Science and Engineering. Special issue. Vol. 17, No. 3, pp. 247–262. 2018. DOI: 10.1504/IJCSE.2018.095847

[3] Z. Jin and J. S. Vetter, "A Benchmark Suite for Improving Performance Portability of the SYCL Programming Model," 2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Raleigh, NC, USA, 2023, pp. 325-327, doi: 10.1109/ISPASS57527.2023.00041. (https://ieeexplore.ieee.org/document/10158214)

[4] T. L. Cassell, T. Deakin, A. Alpay, V. Heuveline and G. B. Gadeschi, "Efficient Tree-based Parallel Algorithms for N-Body Simulations Using C++ Standard Parallelism," *SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Atlanta, GA, USA, 2024, pp. 708-717, doi: 10.1109/SCW63240.2024.00099. keywords: {Performance evaluation;ISO Standards;Software algorithms;Octrees;Graphics processing units;C++ languages;Scheduling;Hardware;Parallel algorithms;Standards;n/a},