AdaptiveCpp Stdpar: C++ Standard Parallelism Integrated Into a SYCL Compiler

Aksel Alpay
aksel.alpay@uni-heidelberg.de
Heidelberg University
Heidelberg, Germany

Vincent Heuveline
vincent.heuveline@uni-heidelberg.de
Heidelberg University
Heidelberg, Germany

ABSTRACT

Expressing data parallel programs using C++ standard parallelism is attractive not only due to the simplicity of the model, but also due to its highly idiomatic nature. This programming model, commonly referred to as stdpar, can also be used for accelerator programming by offloading calls to standard algorithms, and is supported by multiple vendors, such as NVIDIA with nvc++, AMD with roc-stdpar, and Intel with the new ICPX `-fsycl-pstl-offload` flag. We present AdaptiveCpp stdpar, a novel stdpar implementation that is part of the AdaptiveCpp SYCL implementation. AdaptiveCpp stdpar is the very first stdpar implementation to demonstrate performance across GPUs from Intel, NVIDIA and AMD, and allows users to start developing applications using C++ standard algorithms, and then progressively move to SYCL as more control is needed. We find that our solution outperforms all vendor stdpar compilers on HPC GPUs in the majority of tested applications, in some configurations by up to an order of magnitude. Furthermore, we show how AdaptiveCpp outperforms nvc++ in a latency-bound code for all tested problem sizes by up to 80% on NVIDIA A100 due to novel optimizations. Our stdpar implementation deviates from existing implementations by relying on a tighter integration with compiler and runtime, including e.g. dedicated optimization passes to elide synchronization, automatically prefetching required allocations, and an offloading heuristic.

CCS CONCEPTS

• **Software and its engineering → Runtime environments**; **Parallel programming languages**; **Compilers**.

KEYWORDS

C++, SYCL, compilers, parallelism, heterogeneity, parallelruntimes, HIP, CUDA, GPU, SPIR-V, LLVM, stdpar

ACM Reference Format:
Aksel Alpay and Vincent Heuveline. 2024. AdaptiveCpp Stdpar: C++ Standard Parallelism Integrated Into a SYCL Compiler. In *International Workshop on OpenCL and SYCL (IWOCL '24), April 08–11, 2024, Chicago, IL, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3648115.3648117

IWOCL '24, April 08–11, 2024, Chicago, IL, USA

ACM ISBN 979-8-4007-1790-1/24/04
https://doi.org/10.1145/3648115.3648117

1 INTRODUCTION

The C++ 17 [14] parallel standard template library (PSTL) provides parallel, high-level algorithms for common operations, such as `std::for_each`, `std::reduce` or `std::sort`. It is attractive to consider offloading calls to these algorithms to accelerators, such as GPUs. If the base language already provides parallel constructs, then offloading those to accelerators could simplify heterogeneous computing, since no code changes are required for standard C++ code to benefit from accelerators. This in turn could potentially lower the barrier of entry into heterogeneous computing. Furthermore, simply recompiling code using standard C++ parallel algorithms with an offloading compiler might result in speedups. The programming model of utilizing data-parallel standard C++ algorithms to target parallel hardware is commonly referred to as *stdpar*, and has been pioneered most notably by NVIDIA and their nvc++ compiler [8] [24] for NVIDIA hardware. More recently, other vendors have also started work on supporting the stdpar model: AMD has recently published roc-stdpar [12], and Intel's oneAPI ICPX compiler has added an experimental flag `-fsycl-pstl-offload` [7]. There are also discussions in the LLVM community about adding PSTL offload support using OpenMP offloading infrastructure[1]. Of the three vendor solutions, only roc-stdpar is open source, and the stdpar support is notably missing from the open-source oneAPI DPC++ compiler repository[2]. These vendor solutions generally share a similar design idea: A library written in a vendor-native programming model (thrust written in CUDA, rocThrust in HIP or oneDPL in SYCL) implements the algorithms, while the compiler and runtime are mostly unaware of the semantics of the stdpar model[3]. With these approaches, once the application is compiled with offloading enabled, offloading stdpar algorithms takes place independently of whether it is beneficial.

We argue that such a library-focused solution does not lead to optimal results in terms of both functionality and performance, and that instead the underlying compiler and runtime need to be integrated more tightly with the stdpar model, e.g. for optimizations or diagnostics. In this work, we present AdaptiveCpp stdpar: A novel stdpar implementation that is part of the AdaptiveCpp SYCL implementation and which was developed independently during a similar time frame as roc-stdpar and the stdpar support of ICPX.

Our contributions are:

- The first open-source stdpar implementation based on SYCL;

[1] https://discourse.llvm.org/t/rfc-openmp-offloading-backend-for-c-parallel-algorithms/73468
[2] https://github.com/intel/llvm
[3] apart from ensuring basic functionality, such as memory management

- A solution that can demonstrate performance on a wider range of hardware configurations[4] compared to existing vendor solutions and indeed, the first stdpar implementation to demonstrate performance across NVIDIA, AMD and Intel GPUs;
- The first stdpar implementation that departs from a library-focused design and instead integrates the stdpar model more deeply into compiler and runtime. This includes new optimization passes to detect and elide unnecessary barriers, automatic prefetching of required allocations, a pointer validation layer to handle e.g. capture-by-reference cases and an offloading heuristic to detect cases where offloading is not beneficial.
- A performance evaluation of our solution.

Our implementation is not focused on hardware from one particular vendor, and instead supports targeting CPUs as well as GPUs from Intel, NVIDIA and AMD. Adding stdpar support to a SYCL implementation like AdaptiveCpp is particularly interesting, because it allows users to start out application development using high-level, idiomatic stdpar algorithms, and then progressively move to SYCL kernels as more control is required while retaining portability.

1.1 SYCL

SYCL [16] is a modern single-source heterogeneous programming model based on pure C++. The most widely used implementations are arguably DPC++ [6] and AdaptiveCpp [1], both of which target a wide range of hardware. Both are open source, but DPC++ is also available as part of Intel's closed-source oneAPI product distribution with the compiler driver ICPX and additional proprietary features.

1.2 AdaptiveCpp

The work presented in this paper is implemented in AdaptiveCpp. AdaptiveCpp (see e.g. [1], [3]), formerly known as hipSYCL, is a SYCL implementation providing OpenMP [4], CUDA [22], ROCm [11], OpenCL [21] and Level Zero [13] backends. As such, it can target CPUs and GPUs from NVIDIA, AMD, and Intel, as well as potentially other OpenCL devices.

AdaptiveCpp on the one hand supports compilation-flows focused on interoperability, such as library-based host compilation, or integration into the clang CUDA or HIP toolchain. On the other hand, AdaptiveCpp provides a unique backend-independent generic single-pass compiler which relies on just-in-time (JIT) compilation of LLVM's [17] intermediate representation (IR). This compilation flow can currently target Intel, NVIDIA and AMD hardware [2] and is currently the only known compiler to implement the single-source, single compiler pass (SSCP) model as defined by the SYCL 2020 specification.

1.3 The Stdpar Model for Offloading

The C++ 17 standard defines several execution policies that can be passed as first argument to PSTL algorithms. These execution policies assert to the compiler the degree of parallelization that is safe to employ:

- `seq`: The algorithm must not be parallelized;
- `par`: The algorithm may be parallelized;
- `par_unseq`: The algorithm may be both parallelized and vectorized.

The `par_unseq` policy requires that it only be used with code that is explicitly vectorization-safe. E.g. memory allocation or locking mutex objects is forbidden for code executing with this policy. Since an accelerator like a GPU might utilize both parallelization and vectorization, the `par_unseq` policy usually fits best to an offloading use case. Indeed, the SYCL specification also assumes that code might be vectorized across work items, and that those might execute in lockstep. The restrictions that `par_unseq` imposes thus generally map well to SYCL device code restrictions.

Stdpar offloading solutions typically focus on the `par_unseq` policy. The only exception to our knowledge is nvc++, which also supports the `par` policy.

In order for stdpar offloading to work, there are two main prerequisites:

(1) A special stdpar compiler needs to detect which parts of the application need to be compiled for the target device without any help from user annotations;
(2) Because C++ standard parallelism is unaware of different disjoint memory spaces (such as host memory and a GPU's global memory), any necessary memory transfer needs to happen automatically and transparently.

The first point is already addressed in modern heterogeneous programming models such as SYCL, where compilers already support inferring which parts of the code need to be compiled for device. Consequently, SYCL is well-suited as a lower layer for an stdpar implementation.

The second point requires more effort, and is typically solved by intercepting calls to memory management functions, and ensuring that all allocations are accessible on both host and device[5]. See section 3 for details. Listing 1 illustrates the stdpar model.

We note that it is possible in the stdpar model to obtain an index denoting the position in the parallel iteration space, for example by defining a custom iterator type that merely counts the position in a range, and returns the index when dereferenced. This index can then be used similarly to a work item index in SYCL.

Furthermore, an `std::for_each` call with `par_unseq` execution policy is very similar conceptually to a SYCL `parallel_for` kernel, as in both cases, an operation is executed across a parallel iteration space in a single-program-multiple-data (SPMD) fashion.

2 GENERAL ARCHITECTURE

We intercept the C++ `algorithm`, `numeric` and `execution` headers to inject support for offload-aware PSTL algorithms. This is achieved by adding explicit overloads for the `par_unseq` execution policy. Currently, support focuses primarily on libstdc++ 11 or newer, although older versions may work as well.

Invoking one of the offload-capable overloads results in a call to a corresponding function from our lower-level `algorithms` library, which provides SYCL implementations of these algorithms. Functionality from the `algorithms` library can be invoked directly by

[4] e.g. AMD GPUs without support for the XNACK hardware feature

[5] This approach does not work for stack allocations, which can only be supported if system USM is available. See section 3 and section 5.2.

```
// Calculates the sum of x*x + 1 for all elements x in
    data
int f(std::vector<int>& data) {
  std::for_each(
    std::execution::par_unseq,
    data.begin(), data.end(), [](int& x){ x = x*x;});
  return std::transform_reduce(
    std::execution::par_unseq,
    data.begin(), data.end(), 0, std::plus{},
    [](auto val) { return val + 1; });
}
```

Code listing 1: Example stdpar code

users, if more control in terms of SYCL objects (e.g. `queue`, `event`) is required.

Generally, our implementations of PSTL algorithms are mapped to one or multiple SYCL kernel calls[6].

If an algorithm is called that does not have an offloading implementation, code will compile and execute correctly, but no offloading takes place. This design allows us to incrementally add offloading support to more algorithms.

The set of supported algorithms[7] in our implementation is currently still smaller compared to other stdpar solutions, such as nvc++. This is because we have focused on first creating an efficient overall stdpar infrastructure using dedicated compiler and runtime optimizations (see section 6). Additional algorithms can easily be added in the future, or ported from existing heterogeneous algorithm libraries such as thrust or oneDPL.

Our stdpar implementation is supported with any of AdaptiveCpp's compiler-based compilation flows, and can thus run on CPUs, as well as GPUs from NVIDIA, AMD and Intel. Using the generic single-pass compiler, a binary can be conveniently created that can offload stdpar algorithms to all supported devices. Enabling the stdpar functionality requires the `--acpp-stdpar` compiler flag.

3 MEMORY MANAGEMENT

3.1 The Need for Shared or System USM

The stdpar model inherits a flat memory hierarchy from C++, which is not aware of the multiple, distinct memory spaces that are encountered when accelerators such as GPUs are involved. Thus, memory needs to be accessible on the accelerator without using special memory allocation functions or explicit data transfers. Additionally, pointers may be dereferenced that were themselves loaded from memory pointed to by other pointers. This pattern is called *indirect access* and commonly occurs for example when pointer-based data structures such as trees or linked lists are used.

If indirect access is involved, the compiler and runtime will in general no longer be able to determine which allocations are used in kernels, and are thus unable to generate explicit data transfers in all cases. However, this problem can be addressed via unified shared memory (USM) allocations. SYCL distinguishes different kinds of USM allocations. The *shared* variant specifically refers to memory that can migrate automatically between host and device, and can be accessed on both. This migration is usually implemented at the driver and hardware level and does not involve the SYCL compiler or runtime. An implementation of shared USM typically involves the hardware generating a page fault if memory is accessed that is not currently present, and the device driver handling that page fault by triggering the migration of the involved memory pages.

Consequently, the typical strategy for memory management in stdpar implementations is to intercept all calls to memory allocation and deallocation functions and interposing memory management using shared USM memory[8]. This is not necessary however if *system* USM is available. System USM refers to the case when every host memory address is directly accessible by the device. This may be the case if host and device are tightly integrated (perhaps even sharing the same physical memory), if technologies like Linux heterogeneous memory management (HMM)[9] are involved, or if the target device is the host CPU itself. In a system USM scenario, no modifications to memory management are needed by stdpar implementations. AdaptiveCpp provides the `--acpp-stdpar-system-usm` flag, which disables the AdaptiveCpp stdpar memory management interposition layer. In the following, we will assume that system USM is not available unless explicitly noted otherwise.

3.2 Memory Management Interposition

If memory management is intercepted globally und unconditionally, problems arise in the SYCL runtime library and its backends. In this case, the SYCL runtime becomes both provider and consumer of USM allocations. This can cause deadlocks, stack overflows due to infinite recursion, or crashes during program shutdown if e.g drivers have unloaded and can no longer free USM memory.

We solve this problem using multiple strategies:

(1) Thread-local function call counters are used to prevent infinite recursion, and memory functions are potentially redirected to regular memory management functionality.
(2) Limiting the scope of memory allocation interposition. While memory deallocation functions such as `free()` are intercepted globally by providing appropriate symbols, memory allocation functions such as `malloc()` are only locally intercepted by replacing them in the LLVM IR of source files compiled with AdaptiveCpp using a new LLVM pass.

Compared to other stdpar implementations, AdaptiveCpp's design arguably introduces more complexities because it on the one hand supports multiple backends where we cannot control the internal behavior (e.g., internal CUDA or ROCm libraries), and the runtime itself depends on C++ standard library functionality such as `std::vector` or `std::shared_ptr`. The latter matters due to the C++ one definition rule (ODR): If both the application and runtime library contain definitions for the same symbol, the linker will pick one. This might happen for a function which performs memory allocations, such as the `std::vector` constructor. Ultimately, this can cause memory allocations to be intercepted in code that was not

[6]The exceptions are `fill` and copy algorithms, which if possible might be mapped to `queue::memset` and `queue::memcpy`, respectively
[7]A list of supported algorithms can be found here: https://github.com/AdaptiveCpp/AdaptiveCpp/blob/develop/doc/stdpar.md
[8]Note that this fails if pointers to stack memory locations are involved, which can easily happen e.g. by passing a lambda argument which captures by reference. See section 5 for a discussion of this case.
[9]https://www.kernel.org/doc/html/latest/mm/hmm.html

compiled with the AdaptiveCpp stdpar compiler, such as libraries used internally by our runtime library, or vice versa, cause function calls that should be intercepted to not be intercepted. To mitigate this, we have expanded the new LLVM pass with the functionality to

(1) Perform full call graph duplication for all callers of memory management functions;
(2) Add an ABI-tag into the symbol name of duplicated functions to denote whether allocations are intercepted. This is only possible if it is guaranteed that the symbol does not need to be exported. This is typically the case e.g. for all STL symbols, or symbols from AdaptiveCpp headers.

Furthermore, memory allocation calls that originate from within the internal AdaptiveCpp namespaces and headers are never replaced. This reduces the pressure on the AdaptiveCpp stdpar memory management layer during runtime operations, such as kernel submission.

In order to implement `realloc`, an stdpar implementation must be able to retrieve the allocation size for a given pointer.

We solve this by introducing an associative allocation tracking data structure. This data structure in particular is designed to allow retrieval of the allocation size and allocation start address of any pointer within a tracked allocation[10]. The intercepted functions for memory allocation and release update this allocation tracking data structure.

Our allocation tracking data structure is a prefix tree that operates on 4-bit characters of a 64-bit pointer key. The data structure is semi-lock-free, i.e. queries and inserts can be executed without locks.

Since memory deallocation functions are intercepted globally, it can happen that they are invoked with a pointer that does not originate from an interposed memory allocation function, and hence cannot be released using SYCL USM memory free functionality. This is handled by querying the allocation tracking data structure for every pointer, and thus deciding whether USM free or regular host free needs to be invoked.

4 EXECUTION MODEL

Stdpar operations are dispatched to a thread-local in-order SYCL queue. The device attached to the queue is the SYCL default device, and can thus be modified using the regular AdaptiveCpp mechanisms, such as the `ACPP_VISIBILITY_MASK` environment variable. Submitting work from multiple threads allows to express concurrency within one device.

Since neither calls to stdpar algorithms nor calls to memory allocation functions are provided with any information about the target device in the stdpar model, stdpar solutions are typically challenged when multiple offloading devices within one process need to be utilized. Our implementation is no exception and does not yet support this use case. This is primarily due to the additional complexities for memory management when multiple devices are involved, as every pointer still needs to be valid everywhere. In SYCL, where devices might potentially originate from different backends, this is a particularly challenging problem to solve.

On multi-device systems, users currently can e.g. utilize frameworks like MPI to achieve a mapping of one device per process.

In principle, every stdpar operation must be executed synchronously since the semantics of C++ standard parallelism allows programs to expect that all data is ready after a call to a PSTL algorithm returns. However, in section 6 we describe an optimization that allows AdaptiveCpp to elide that synchronization, if the effect of the elision is not observable.

5 CORRECTNESS

Ideally, any code that is correct C++ should work in stdpar offloading code. In practice, this is often in conflict with restrictions that typically govern device code. For SYCL, these restrictions include:

- Kernel lambda functions must capture by-value, not by reference;
- Host pointers may not be dereferenced on device, unless system USM is available;
- function pointers or virtual functions are not allowed;
- exceptions are not allowed;
- non-trivial types may only be passed as kernel arguments if they adhere to the SYCL *device copyable* concept, which requires specializing the `is_device_copyable` type trait for user-defined types;
- builtin functions such as math functions must only be invoked using their implementations in the `sycl::` namespace; calls to `std::` math functions or calls to functions in the global namespace from the C standard library are generally not allowed.

Stdpar implementations in principle must be able to handle these cases. This can be addressed by lifting device restrictions via dedicated SYCL extensions. However, this is not the only possible approach. For example, an implementation might also detect whether any functionality is used inside an stdpar call that cannot be supported on device, and if so, fall back to execute on the host. This may however require delaying or modifying the usual SYCL compiler diagnostics, such that usage of such functionality inside an stdpar call does not immediately trigger a compilation failure.

To our knowledge, so far there is no stdpar implementation that handles all of these cases. This is also in principle true for our current implementation. However, we have taken efforts to mitigate potential problems by handling those features that we believe are most likely to be encountered in practice in the context of parallel STL usage. This allows our implementation to handle cases that other stdpar implementations cannot handle.

5.1 Non-SYCL Builtin Functions

In order to address the usage of `std::` functions, we have added a new compiler pass to AdaptiveCpp's generic single-pass compiler that remaps calls to libc math functions to the AdaptiveCpp device compiler builtins. This allows calls to C and C++ standard math functions to work seamlessly inside stdpar (or SYCL) kernels. We note that our compiler can support calls to functions such as `::cbrt` that do not currently work in roc-stdpar due to lack of support in the clang HIP headers. A limitation in our present implementation

[10] Note that this implies that the data structure cannot be directly compared to a regular associative container like `std::map`. `std::map` maps a value to another value, while our data structure maps a range of key values to another value.

is that builtins for complex math are not yet implemented, i.e. it is not yet possible to use `std::complex` with AdaptiveCpp. This is planned for future releases.

5.2 Capture-by-Reference and Host Pointers

Furthermore, we allow our kernels to capture by reference. If `--acpp-stdpar-system-usm` was not used, all pointers passed as kernel arguments are validated using SYCL 2020 USM pointer information query mechanisms. If a pointer is found to not be a valid USM pointer for the target device, then AdaptiveCpp will emit a warning at runtime, and the PSTL algorithm will be executed on the host to guarantee correctness.

Listing 2 shows an example code which will compile and execute correctly with AdaptiveCpp stdpar because it detects the access to host memory at runtime, and consequently executes the code on the host instead of offloading. The other stdpar solutions fail in various ways: roc-stdpar crashes with a segfault and nvc++ emits a warning at compile time and then crashes due to an illegal memory access at runtime. ICPX does not compile the capture-by-reference case.

We note that, while our solution for handling invalid pointers is likely sufficient for many practical use cases, it is not bulletproof. If the pointer kernel argument is obscured from the compiler[11], then our validation layer will fail to detect cases where offloading is not possible. It is still unclear to what extent more compiler analyses could address these cases (e.g., reject offloading for all kernels relying on pointer casts or indirect access) without introducing too many false positives such that stdpar offloading support can remain useful.

```
int main() {
  int data [1024];
  // PSTL algorithms may access stack memory
  std::for_each(std::execution::par_unseq,
  &data[0], &data[1024], [=](auto& x){ x = 1; });

  // PSTL algorithms may capture by reference
  int c = 3;
  std::vector<int> data2(1024);
  std::for_each(std::execution::par_unseq,
  &data2[0], &data2[1024], [&](auto& x){ x = c; });
}
```

Code listing 2: Stdpar calls accessing stack locations.

5.3 Custom Data Types

Because AdaptiveCpp generally handles SYCL diagnostics more permissively compared to ICPX, our solution also allows using custom data types in stdpar kernels that do not satisfy the device-copyable trait, and do not specialize `sycl::is_device_copyable`. As a consequence, our solution can support cases that are legal in C++, which however ICPX rejects due to it strictly enforcing the SYCL 2020 rules. [12]

[11] e.g. via indirect access inside the kernel, or pointer-to-integer casts

[12] Nevertheless, it is true that in principle custom data types with complex copy semantics can become an issue when they are moved to device (e.g., it is ill-defined how many times an object is copied when it crosses the host-device boundary). What is less clear is the practical relevance of such cases, as it is likely that types with such complex copy semantics might already violate C++'s assumption that `par_unseq` code is vectorizable.

5.4 Memory Management Interoperability

A memory deallocation call in a program component that was not compiled with the stdpar compiler must be able to release USM memory allocated in an intercepted memory allocation function. This is because pointers might be released by a different program component than the one originally performing the allocation.

AdaptiveCpp solves this by intercepting the symbols for memory deallocation functions (`free()`, `operator delete` etc) for the entire program. Our implementation of these functions then queries our dedicated memory allocation tracking data structure to determine whether the pointer is a USM pointer or a host pointer. For host pointers, the regular host memory release functions are invoked, while USM pointers are freed using the SYCL USM mechanisms.

Listing 3 shows an example consisting of two translation units where this interoperability is important. This code works with AdaptiveCpp, nvc++ and ICPX, but crashes with roc-stdpar. Roc-stdpar does not intercept memory deallocations globally, but instead relies on the compiler replacing calls to those functions. Consequently, it does not support transferring memory ownership between different components of the program. We believe that this issue constitutes a notable limitation of roc-stdpar.

```
// a.cpp (compiled with stdpar compiler)
void release(int* ptr);
int main() {
  int* mem = new int[1024];
  std::for_each(std::execution::par_unseq,
    &mem[0], &mem[1024], [=](auto& x){x=1;});
  release(mem);
}
// b.cpp (compiled without stdpar compiler)
void release(int* ptr){ delete [] ptr; }
```

Code listing 3: Deallocation needs to work whether a translation unit was compiled with the stdpar compiler or not.

6 OPTIMIZATIONS

In addition to the functionality described in section 5 and the stdpar memory management interception, the AdaptiveCpp stdpar implementation includes a number of dedicated optimizations.

6.1 Submission Latency

The stdpar execution model is simpler than the SYCL model. While SYCL allows for arbitrary task graphs, in the stdpar model we know that all tasks submitted from the same thread will be executed sequentially. Additionally, we know that memory will be exclusively managed through USM pointers. Furthermore, the stdpar model never uses the `sycl::event` objects that are returned from kernel submissions.

We exploit this with two optimizations:

(1) All stdpar kernels are submitted using the AdaptiveCpp coarse-grained events extension[13] extension. This extension instructs the runtime to map `sycl::event` objects not to actual backend events, but instead to synchronize with the entire backend queue, if event synchronization is requested. This avoids having to create new backend events for every kernel launch.
(2) Additionally, we have introduced a new kernel submission model called *instant submission*. In the instant submission model, the AdaptiveCpp runtime will bypass its operation batching layer and bypass additional functionality that is needed to handle the buffer-accessor model in SYCL, but unneeded for stdpar. As a consequence, submitting an stdpar operation generally is expected to have lower latency compared to submitting a SYCL kernel.

6.2 Memory Pool

Calling USM memory management functions is typically more expensive than regular host memory management functions. Additionally, calls to USM memory management functions may imply some degree of synchronization with the device, e.g. to update the device page tables. These issues can affect application performance, if the application calls interposed memory management functions during performance-critical parts of the code. We address this by employing a memory pool that allocates a large amount of USM memory[14] at startup. The memory pool size is adjustable by users for optimization. All allocation requests are served from this memory pool. If the memory pool capacity is exceeded, the SYCL USM memory management functions are invoked directly. When running on a CPU device, the memory pool size is set to zero, since in that case, the mentioned benefits of this optimization do not apply, and utilizing the memory pool might risk incorrect placement of pages on NUMA systems.

Our memory pool always returns page-aligned memory such that migration of the allocation does not interfere with other allocations.

6.3 Synchronization Elision

For workloads consisting of short-running kernels, the stdpar model requiring synchronous behavior for every submitted kernel can become a performance limitation. To mitigate this, we have implemented a new LLVM optimization pass to elide synchronization where possible. To our knowledge, AdaptiveCpp is the only stdpar implementation that supports such an optimization.

Eliding synchronization is possible when the removal of synchronization does not lead to observable changes in the program semantics. This in particular excludes stdpar algorithms that are expected to return a result value, such as `reduce` or `transform_reduce` from the optimization. If however an iterator or `void` is returned instead (e.g. `for_each`), we consider the algorithm to be potentially non-blocking.

Our stdpar runtime maintains a counter of the number of operations that were enqueued since the last synchronization. Potentially non-blocking algorithms are finished with a call to a new `stdpar_optional_barrier` builtin. This builtin synchronizes the SYCL queue, if the number of enqueued operations is greater than zero. The LLVM transformation detects calls to the builtin and then

(1) Removes the builtin call from the stdpar algorithm definition, and reinserts it after the callsite of the stdpar algorithm;
(2) Then further moves the builtin call down the instruction sequence, taking all paths in the control flow graph until either
   - Another existing call to the builtin is encountered, in which case the current builtin call can be discarded, or
   - A condition requiring synchronization is encountered, in which case the builtin call is inserted right before the condition.

In principle, a condition requiring synchronization is a memory access on the host to the allocations used inside the kernel. However, due to potential indirect access in general it is challenging to prove that an allocation is certainly not used in a kernel. Thus, we currently consider any memory access a condition requiring synchronization. This includes loads, stores, LLVM builtins operating on memory (e.g. `memcpy`), and any externally defined functions as those might also access memory. The encounter of another stdpar call is explicitly not considered a condition requiring synchronization. Thus, this algorithm effectively delays synchronization for as long as possible, and potentially completely removes it in between subsequent stdpar algorithm calls.

One challenge is that stores may frequently appear in order to assemble stdpar call arguments. This happens commonly when function or lambda objects are passed to an stdpar algorithm. To avoid these stores triggering unnecessary synchronization, we detect if stores are used exclusively to assemble a call argument that is then passed to a subsequent stdpar call. In that case, the store can be ignored.

Furthermore, our control flow analysis currently only works within one LLVM function. Because of this, any instruction causing control flow to exit the current function also triggers synchronization in order to guarantee correctness. To mitigate the impact of this, the synchronization elision pass is run at the end of the optimization pipeline, when inlining passes have already run[15]. Furthermore, we run an additional dedicated inlining pass prior to the elision pass to specifically attempt to combine more stdpar calls in one LLVM function, and thus increase the likelihood of successful synchronization elision.

Since the current implementation is run as part of the regular optimization pipeline which operates on one translation unit at a time, it is challenged by programs where the stdpar calls are distributed across many translation units, and where there are only few stdpar calls per translation unit. This could be addressed in the future by moving the elision pass to the link-time-optimization (LTO) pipeline instead.

6.4 Automatic Prefetch

The performance of shared USM allocations can often be improved by providing hints to the driver about their usage. SYCL's `prefetch(void* ptr, size_t size)` can be used to inform the driver that

[13] https://github.com/AdaptiveCpp/AdaptiveCpp/blob/develop/doc/extensions.md#hipsycl_ext_coarse_grained_events
[14] By default, the memory pool is sized to 40% of the device global memory size.
[15] This is also beneficial, since at that point some loads and stores might have already been eliminated.

a memory region is about to be accessed by the device. The driver might then issue the data migration ahead of time, potentially transferring the entire memory region at once instead of waiting for page faults of individual memory pages. SYCL's `mem_advise(void* ptr, size_t size, int advise)` can be used to set additional hints about the expected usage pattern of the data.

In the context of stdpar, where it is expected that shared USM allocations are widely used, providing these hints automatically appears as an attractive optimization strategy. To our knowledge, no other stdpar implementation however implements this.

The AdaptiveCpp stdpar implementation supports automatically submitting prefetch hints for directly accessed allocations. This is achieved by introspecting all pointer-type kernel arguments, similarly to the pointer validation layer introduced in section 5. From the allocation tracking data structure, the base pointer of the allocation and allocation size is obtained, which are needed to emit a prefetch with a valid memory range.

However, whether a prefetch is actually emitted depends on the user-selected[16] prefetch mode. AdaptiveCpp currently supports multiple prefetch modes:

- *first* – a prefetch is only emitted for the very first time an allocation is used on device. To this end, the allocation tracking data structure also tracks how often allocations have been used by stdpar calls.
- *after-sync* – prefetches are emitted for the allocations used by the first kernel after a synchronization point. This can be a good choice if multiple groups of stdpar calls with elided synchronization are separated by host work on the allocations. In such a scenario, this prefetch mode emits prefetches once at the begining of every such batch of kernels.
- *always* – prefetches are unconditionally emitted
- *never* – automatic prefetches are disabled.
- *auto* – this is the default and currently synonymous with *first*. Adding more sophisticated heuristics in this mode is planned for future work.

Especially in the prefetch modes that tend to prefetch often such as *always*, the submission latency of prefetch operations can become relevant for the performance of short-running kernels. Prefetches are thus submitted by directly invoking internal functionality of the AdaptiveCpp runtime backends, and all SYCL layers are bypassed.

Automatically emitting `mem_advise` calls is the subject of future work and not yet supported.

6.5 Conditional Offloading

Depending on hardware and problem, offloading an stdpar algorithm might not always be the most performant choice. For example, for very small problem sizes, it may be better to simply execute the algorithm on the host and avoid the additional latency of offloading.

Consequently, an optimized stdpar implementation should attempt to estimate whether offloading is beneficial using appropriate heuristics or schedulers. This can be a very challenging problem, and has given rise to complex runtime systems which attempt to solve the problem in general, e.g. [5]. For the stdpar model, the problem is simplified due to the exclusively linear task graphs.

[16]Through environment variables or the `-acpp-stdpar-prefetch-mode` compiler flag

We have added an offload heuristic to the AdaptiveCpp stdpar implementation. To our knowledge, no other stdpar implementations supports such a conditional offloading mechanism.

We want to emphasize that we do not claim that our offloading heuristic delivers optimal results in all cases. It is however reasonably easy to implement and can deliver good results for common problems, as we show in section 8. We argue that stdpar implementations should provide conditional offloading support, not necessarily our heuristic in particular.

Our offloading heuristic is based on performance measurements at runtime. The runtime of both offloaded and non-offloaded parallel STL algorithms is measured. When offloading, the end-to-end time from kernel submission to synchronizing the queue is measured. This is important in order to capture offloading latencies. The expectation value of the runtime for each stdpar call is recorded in a file, which is continually updated with each application run. Stdpar calls are identified via a hash of algorithm type and all template types involved in the instantiation of the algorithm. The expectation value is maintained separately per problem size of the invocation of the stdpar algorithm. This expectation value is used during the application run to predict runtimes for a given stdpar algorithm and problem size. If no data is present yet for a given problem size, runtimes are predicted by interpolating (or extrapolating) from the closest measured problem sizes.

If no baseline knowledge about host performance is available yet, the heuristic currently offloads unconditionally. In order to achieve conditional offloading, the application must be run at least once with the environment variable `ACPP_STDPAR_HOST_SAMPLING` set. This will request a run where no offloading takes place, and thus causes host performance results to be gathered. The application runs for host sampling can be performed for smaller problems than production runs, in which case predicted runtimes will be extrapolated.

During the application run, it is recorded which stdpar calls succeed each other. When deciding whether to offload, our heuristic calculates the expected runtime for a predefined number of tasks in the succession chain by summing up their individual predicted runtimes. The cost of potential data transfers is estimated as well from the total data size of all directly accessed allocations of the stdpar calls in question.

The calculations to determine the offloading behavior are not free and can impact overall performance in latency-bound scenarios if they are executed too frequently. We thus conservatively enforce that reevaluation of the current offloading behavior is only allowed after at least 1 second and 128 executed stdpar algorithms. Users can change these defaults using environment variables, if desired.

7 EXPERIMENTAL SETUP

In the following, we will be presenting results obtained on the following systems:

- System 1: Linux 4.18, 2x AMD EPYC 7713, 64 cores each, 256GB RAM, 409.6 GB/s theoretical peak memory bandwidth.
- System 2: Linux 4.18 (HMM enabled), AMD EPYC 7543P, 32 cores, 256GB RAM, 4x AMD Instinct MI100. GPU peak memory bandwidth: 1229GB/s.

Benchmark	Run parameters
BabelStream	`-s 70000000`
CloverLeaf	`--file clover_bm64_300.in`
miniBUDE	`-w <32, 64, 128, ..., 1024> -p all`
TeaLeaf	In `tea.in` file: `x_cells=1024` and `y_cells=1024`
LULESH	`-s <problem size>`

Table 1: Chosen parameters for benchmarks

- System 3: Linux 4.18, AMD Epyc 7543P, 32 cores, 256GB RAM, 4x NVIDIA A100 40GB. GPU peak memory bandwidth: 1555 GB/s.
- System 4: Linux 5.15.0 (Ubuntu 22.04), 2x Intel Xeon Platinum 8480+, 56 cores each, 1024GB RAM, 4x Intel Data Center GPU Max 1550. GPU peak memory bandwidth: 3276 GB/s

The software stack includes:

- AdaptiveCpp commit f2c2960, built against LLVM 15 and libstdc++ 12, and invoked with `--acpp-targets='omp;generic'`.
- oneAPI 2024.0.2, Intel OpenCL GPU runtime 23.35.27191.9
- CUDA 12.1, NVHPC 23.5, NVIDIA driver 525.125.06
- ROCm 5.4.1, roc-stdpar commit[17] 8c57cd0. We use roc-stdpar's `--hipstdpar-interpose-alloc` flag.
- LULESH [15] commit[18] 29f6475
- BabelStream [10] commit[19] e5ad001
- CloverLeaf [18] commit[20] e243531
- TeaLeaf [20] commit[21] e70261c
- miniBUDE [23] commit[22] 6bfdb64

Benchmarks are compiled with `-O3 -march=native -DNDEBUG` flags. Benchmarks with cmake build systems are built in release configuration. Table 1 shows the parameters that were selected for the benchmark runs.

Both ICPX and AdaptiveCpp can target Intel GPUs either through OpenCL or Level Zero backends. For ICPX, we use the Level Zero backend as that is the default. For AdaptiveCpp however, the Level Zero backend is less mature than the OpenCL backend, and also does not yet implement prefetches. We thus use AdaptiveCpp's OpenCL backend for Intel GPU hardware. ICPX compiles with `-ffast-math` by default; we have disabled this to align with the other compilers.

8 RESULTS

8.1 Memory Performance: BabelStream

Due to the importance of the performance of shared USM allocations for the stdpar model, it is useful to first investigate basic memory performance with the BabelStream [10] memory benchmark. BabelStream is available in different models, including stdpar and SYCL. Figure 1 shows BabelStream results for AdaptiveCpp in its default configuration with the stdpar and SYCL models, and other compilers as a fraction of theoretical peak.

8.1.1 Host execution. While most of our optimizations are targeted at GPU offloading scenarios and this work thus focuses on comparing offloading performance, we have also included results for CPU execution through AdaptiveCpp stdpar. We observe a roughly $\approx 5\times$ performance increase when utilizing AdaptiveCpp on CPU over regular PSTL using clang 15 and libstdc++ 12 on an AMD host system. This is primarily due to NUMA issues in the TBB-based parallel STL implementation of libstdc++, which the AdaptiveCpp OpenMP backend avoids. This NUMA-related performance behavior is already well-known in the context of SYCL implementations with OpenMP backends versus TBB backends [9], and is hence not surprising. AdaptiveCpp's stdpar implementation can consequently also deliver substantial performance improvements even if the target device is the host CPU.

8.1.2 NVIDIA A100. On the NVIDIA GPU, we find that both AdaptiveCpp and nvc++ achieve a good fraction between 0.7 and 0.9 of theoretical peak. In most cases, performance between both is within 5%. For the dot product however, nvc++ achieves 90% of peak performance while AdaptiveCpp only delivers 72%. We attribute this to the more optimized device reduction implementation in thrust.

8.1.3 AMD Instinct MI100. For the implementation of shared USM allocations, the ROCm stack relies on a hardware feature called *XNACK*, which enables instruction replay in case of a page fault. Enabling XNACK support requires Linux HMM and non-standard Linux kernel options, and can thus not be enabled by unprivileged users, if it is not enabled on the host system. Production HPC systems in our experience rarely have XNACK enabled. Additionally, current consumer-grade AMD RDNA GPUs commonly lack XNACK hardware support entirely. We thus expect that most AMD users will not have access to an XNACK-enabled machine. In the absence of XNACK, on-demand page migration is disabled in ROCm, and shared USM allocations remain resident in host memory. This can cause severe performance degradation since every memory access of compute kernels needs to traverse the interconnect (e.g PCIe).

To address the limited availability of XNACK, Lin et al. [19] have proposed an `LD_PRELOAD` library called UTPX to implement userspace data migration for allocations that are not indirectly accessed. Notably, they also observed and exploited in UTPX that invoking a prefetch operation on a suitably aligned memory address can cause that memory to become device-resident, even if XNACK is not available, and thus avoid the performance penalty of missing XNACK support. This behavior is not documented by AMD.

Because AdaptiveCpp enables automatic prefetching as an optimization by default and because we ensure that all allocations are page-aligned, this ROCm behavior is also triggered by AdaptiveCpp.

This is reflected in our BabelStream results on AMD. The presented values are without XNACK support. As expected, roc-stdpar performs poorly at around 3% of theoretical peak, because allocations are not resident in device memory. AdaptiveCpp on the other hand manages to achieve roughly 80% of theoretical peak performance.

[17] https://github.com/ROCmSoftwarePlatform/roc-stdpar
[18] https://github.com/illuhad/LULESH
[19] https://github.com/uob-hpc/babelstream
[20] https://github.com/uob-hpc/cloverleaf
[21] https://github.com/uob-hpc/tealeaf
[22] https://github.com/illuhad/minibude

When enabling XNACK, performance increased to roughly 80% of peak for roc-stdpar as well. However, in this case BabelStream results failed validation for both AdaptiveCpp and roc-stdpar. Common ROCm debug settings such as `AMD_SERIALIZE_COPY=3` were tried, but did not help. Results with XNACK enabled are thus omitted from the plot. It is both surprising and concerning that ROCm's on-demand page migration solution already fails for such a simple code as BabelStream.

8.1.4 Intel Max 1550. On the Intel Max 1550, the program compiled by ICPX with the `-fsycl-pstl-offload=gpu` flag was unable to run successfully and crashed inside the destructor of the `sycl::handler`. We suspect that the memory allocation interposition layer interfered with internals from the SYCL headers. In order to still obtain reference results using ICPX, we leveraged that BabelStream's std-data model supports directly calling Intel's oneDPL algorithm library, and allocating memory explicitly using `sycl::malloc_shared`. While this stdpar "emulation" can provide an interesting data point for comparison, it is potentially a simplified case for drivers due to the lower amount of shared USM allocations.

We find that AdaptiveCpp performance closely matches the oneDPL performance within 1%, except for the dot kernel. This kernel seems to not be optimized well in the AdaptiveCpp case. However, forcing ICPX to also use the OpenCL backend in line with AdaptiveCpp reduces the oneDPL performance by ≈ 30%, narrowing the gap. This indicates that there is substantial variance in performance between different backends in ICPX, and potentially substantial variance in driver quality.

8.1.5 Comparison to SYCL. AdaptiveCpp results using the BabelStream SYCL model are provided as well. This implementation of BabelStream relies on explicit device allocations using the SYCL buffer-accessor model and can thus be used to determine whether the shared USM allocations in the stdpar model add overhead.

The presented data illustrates that the AdaptiveCpp stdpar performance has its own performance characteristics, independently from SYCL. For example, when running on the AMD Epyc 7713 CPUs, stdpar outperforms SYCL by 2.3× for the triad benchmark. This is due to the overall lower submission latency for stdpar kernels. The SYCL implementation of the benchmark on the other hand needs a larger problem size to saturate performance on this hardware.

The performance of the SYCL dot-Kernel is particularly poor on CPU compared to the stdpar version of the code. This is because the SYCL version of BabelStream provides its own reduction implementation, which utilizes a GPU-optimized heuristic to determine the work group sizes and number of work groups. This does not map well to AdaptiveCpp's CPU backend.

On the tested GPUs, SYCL and stdpar generally provide similar performance, indicating that the reliance on shared USM allocations in the stdpar model does not necessarily add overheads. In the case of triad on the AMD GPU, the stdpar version is even 5% faster than its SYCL counterpart.

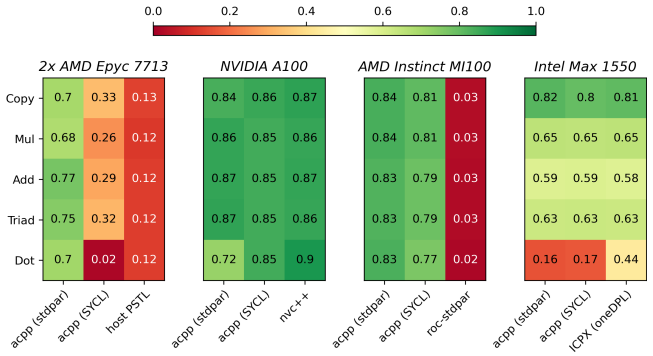

Figure 1: Memory bandwidth as a fraction of theoretical peak for BabelStream's std-data model (acpp: AdaptiveCpp).

8.2 Mini-Apps: Stdpar versus Native Models

Figure 2 shows results for the mini-apps CloverLeaf [18], TeaLeaf [20] and miniBUDE [23]. These benchmarks are available for many programming models, including stdpar and vendor-native models. This allows us to investigate how stdpar compares to native models.

CloverLeaf is a 2D hydrodynamics mini-app on a structured grid, TeaLeaf solves the heat conduction equation, and miniBUDE is a compute-bound molecular dynamics application.

For the AdaptiveCpp stdpar results, the various supported prefetch modes were tried. In all cases, the best performing mode was either the default prefetch mode *first* or *never*. We thus display results for these two modes. However, these mini-apps are generally structured such that the benefit of our optimizations presented in section 6 is mostly limited: There are little synchronization elision opportunities due to the code structure, and data migration and allocation work is mostly limited to the application start. As such, these results should be seen as performance baseline.

Results are normalized by the performance of the native, vendor-supported model: CUDA for NVIDIA, HIP for AMD, and SYCL (using ICPX) for Intel. miniBUDE supports two important optimization parameters: The work group size and the number of poses per work item (ppwi). We have made runs with all supported values for ppwi and power-of-two work group sizes, and include results only for the best combination. The stdpar model does not allow configuring the work group size, and hence does not take this parameter into account. miniBUDE segfaulted with roc-stdpar without XNACK, and is hence missing in the roc-stdpar non-XNACK results.

On the Intel Data Center GPU Max 1550, runtimes with AdaptiveCpp were excessively long if the memory pool was enabled. For comparison, on Intel UHD 620 and 630 iGPUs, this behavior was not observed. This is likely because current Intel drivers always migrate entire allocations (and thus the entire pool) on discrete GPUs. For the purpose of this study, we have disabled the memory pool for the runs on the Intel GPU.

We also include results for the host PSTL on the respective systems using clang 15 and libstdc++ 12 without offloading. For all benchmarks, there was at least one stdpar implementation that was faster than host PSTL execution, and for AdaptiveCpp this was always the case. This indicates that the value proposition of the

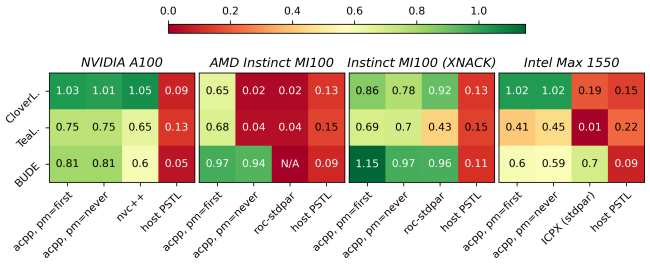

Figure 2: Stdpar performance normalized to native models and compilers. (pm: AdaptiveCpp prefetch mode, CloverL.: CloverLeaf, TeaL.: TeaLeaf, BUDE: miniBUDE.)

stdpar offloading model – simply recompile regular C++ code to utilize GPUs – can indeed work and deliver speedups.

On NVIDIA A100, AdaptiveCpp outperforms nvc++ for both TeaLeaf and miniBUDE by 15% and 35% respectively. Only for CloverLeaf do we find a modest lead for nvc++ of 2%. Across the three benchmarks, AdaptiveCpp's stdpar implementation delivers between 75% and 103% of CUDA performance.

On the AMD Instinct MI100, we again observe the pattern that AdaptiveCpp can deliver acceptable performance (between 65% and 97% of HIP performance, depending on the benchmark) in the absence of XNACK. At the same time, roc-stdpar does not provide any meaningful performance in this case, and does not achieve more than 4% of HIP performance, with miniBUDE not running at all. If XNACK is enabled, AdaptiveCpp still outperforms roc-stdpar by 20% for miniBUDE and 63% for TeaLeaf, achieving between 70% and 115% of HIP performance. roc-stdpar on the other hand leads by 7% in CloverLeaf, but achieving overall only between 43% and 96% of HIP performance.

Unlike the BabelStream case, the binaries generated by ICPX's `-fsycl-pstl-offload` flag could be run successfully for the three mini-apps on the Max 1550. However, performance was very poor for TeaLeaf and CloverLeaf. For CloverLeaf, AdaptiveCpp's stdpar implementation matched the ICPX SYCL performance (slightly exceeding it by 2%), while the ICPX stdpar performance was only able to reach 19% of the performance of the native model. Similarly, where AdaptiveCpp stdpar reached 45% native performance in TeaLeaf, ICPX only reached 1%. There currently appear to be serious performance issues in ICPX's stdpar support for memory-bound applications, potentially even preventing it from reaching practical usability in extreme cases (e.g. TeaLeaf). Only for the compute-bound miniBUDE was ICPX competitive. When using again the oneDPL stdpar "emulation", ICPX performs similarly to AdaptiveCpp. The issue is thus likely caused by the ICPX stdpar memory management layer. Across all three benchmarks, AdaptiveCpp stdpar performance was between 45% and 102% of ICPX SYCL performance, while ICPX stdpar was only between 1% and 70%.

These results indicate that the AdaptiveCpp stdpar implementation is highly competitive against vendor solutions, achieving noticable leads (up to an order of magnitude) for two out of the three applications on all platforms. At the same time, it did not show any of the severe performance issues that some of the vendor solutions encountered.

8.3 LULESH

LULESH [15] is a shock hydrodynamics mini-app that is particularly interesting for optimizing stdpar implementations, because it challenges compilers, runtimes and drivers in multiple ways: It submits many small kernels, thus being highly-latency sensitive, performs frequent allocations and deallocations during the computation, and extensively relies on indirect memory access.

Figure 3 shows LULESH results for NVIDIA A100 for various problem sizes, and figure 4 shows results for AMD Instinct MI100. In the plots, we adopt the abbreviations of `pm` for the prefetch mode, `se` for synchronization elision and `mp` for memory pool. Results are normalized by the performance of the host PSTL (clang 15, libstdc++ 12). For the very large problem sizes of 200 and greater, due to excessive runtime, host PSTL performance numbers were estimated by aborting the computation after 500 iterations, and extrapolating to the required number of iterations for the full solution obtained from the much faster GPU runs.

The AdaptiveCpp offloading heuristic was calibrated using host sampling runs for problem sizes of 5 and 10.

8.3.1 NVIDIA A100. From the results on the NVIDIA GPU, we conclude that the memory pool is a key optimization: With all stdpar-specific optimizations including the memory pool disabled (pm=never,se=off,mp=off), performance scales only to roughly twice the host PSTL performance. If the memory pool is enabled, performance for small and medium problem sizes closely matches nvc++ performance. However, for very large problems beyond a problem size of 150, nvc++ performance does not scale beyond $\approx 10\times$ host PSTL performance, while AdaptiveCpp continues to scale at least up to the largest tested problem size, where it delivers a $16\times$ speedup over host execution. In the regime of these large problem sizes, calls to memory allocation and deallocation functions appear in the profiling timeline of the nvc++-compiled application. For runs with smaller problem sizes, these function calls are not present. This indicates that nvc++ likely utilizes a memory pool which however is too small to handle the larger problems, and thus direct calls to memory management functions are generated in this case. The observed performance impact underlines the importance of avoiding direct calls to USM memory allocation and deallocation functions in performance-critical code paths.

Enabling synchronization elision yields up to $\approx 80\%$ additional performance for the smallest problem size, and around 30% performance for a medium problem size of 100. For LULESH, our synchronization elision algorithm was highly effective, and manages to elide roughly 80% of barriers. This results in AdaptiveCpp substantially outperforming nvc++ for all tested problem sizes – in the most extreme cases, by $\approx 80\%$ for very small and very large problems.

Automatically prefetching was detrimental for performance for all prefetch modes. The additional prefetches increase the host-side overhead in an already highly latency-sensitive application. Furthermore, due to LULESH's reliance on indirect access, the compiler might not even see the actually important large data allocations, instead prefetching small allocations that only store pointers to

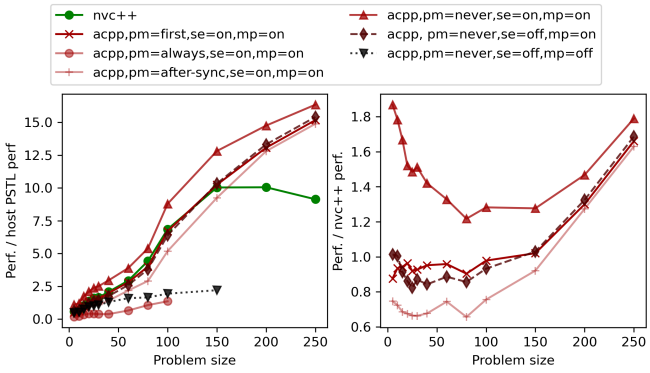

Figure 3: LULESH performance on NVIDIA A100.

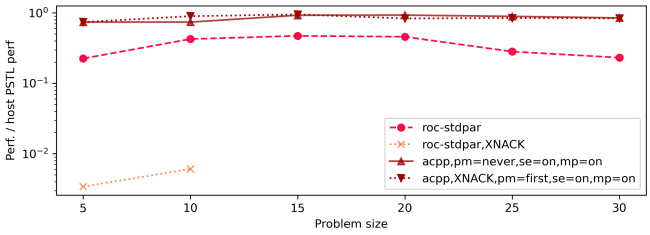

Figure 4: LULESH performance on AMD Instinct MI100.

the data. Thus, page faults still happen and there is little benefit to prefetching.

8.3.2 AMD Instinct MI100. On the AMD Instinct MI100, LULESH performance was poor: Due to excessive runtime, problem sizes larger than 30 were not run. With roc-stdpar and without XNACK, performance never exceeded 47% of host PSTL performance. With XNACK enabled, performance was even worse and below 1% of host performance. Only the smallest problem sizes of 5 and 10 were run due to excessive runtime in the XNACK-enabled configuration. The ROCm stack seems to have severe problems handling LULESH, likely related to the application's frequent memory allocations and deallocations, indirect memory accesses and page faults.

The same performance behavior can in principle also be seen with AdaptiveCpp. However, the offloading heuristic detects the problem and AdaptiveCpp decides thus to not offload. In this way, it can recover between ≈ 70% and ≈ 90% of host performance, depending on the problem size. With XNACK disabled, any prefetch mode other than *never* resulted in a crash due to an illegal memory access on device. It is not known what causes this, as attempts to create minimal reproducers so far have failed. In any case, the poor performance of LULESH also highlights the need for an offloading heuristic that can avoid offloading in cases where offloading is clearly detrimental.

8.3.3 Intel Max 1550. ICPX was unable to compile LULESH. This is because LULESH uses capture-by-reference extensively in calls to PSTL algorithms, and even though the captured references point to shared USM memory and can thus be accessed on device, ICPX reports this as an error. This is in line with SYCL, but overly strict in the stdpar model. This underlines the need for a different handling of diagnostics for stdpar code.

AdaptiveCpp on the other hand successfully compiles LULESH. However, when running on the Max 1550 GPU, the program hangs inside the Intel OpenCL implementation during kernel processing. Because we have seen similar hangs even for simple cases like BabelStream prior to upgrading the OpenCL driver to the version that was ultimately used, we suspect that this might be caused by immature drivers.

To validate LULESH with AdaptiveCpp on Intel hardware, we successfully ran LULESH on two Intel iGPUs (Intel UHD 620 and UHD 630) where this issue was not observed. However, performance was below host performance on this hardware, and the offloading heuristic thus stopped offloading.

9 CONCLUSION

We have presented the AdaptiveCpp stdpar implementation, which supports offloading standard C++ parallel algorithms to any device supported by AdaptiveCpp, including Intel, NVIDIA, and AMD GPUs. Our work allows users to start heterogeneous application development at the highly idiomatic and productive level of regular C++ parallel STL algorithms, and then progressively introduce lower-level SYCL functionality as more control is needed during optimization efforts, without sacrificing portability.

Our approach is highly competitive compared to vendor solutions in the form of Intel's ICPX, NVIDIA's nvc++ or AMD's roc-stdpar, and it introduces multiple novel features that these alternatives do not provide. This includes a pointer validation layer to support capture-by-reference, support for automatically prefetching required allocations, the ability to elide unnecessary barriers, and an offloading heuristic to avoid offloading when this would be detrimental.

We have discussed a robust implementation of a memory interposition layer to ensure all allocations of the program are accessible on device, and we have demonstrated using Babelstream that there is not necessarily a performance impact on memory performance compared to SYCL code using explicit device allocations – in the case of BabelStream, the difference was within 5% percent. Additionally, our stdpar solution can provide substantial (≈ 6× for triad) performance increases on CPUs with strong NUMA effects compared to common host PSTL implementations such as libstdc++.

On AMD GPUs, roc-stdpar depends on the XNACK hardware feature, which is however poorly accessible due to lack of hardware support in most consumer GPUs, and it often not being enabled in production HPC systems. It is therefore likely that most users will not have XNACK available. We have demonstrated how AdaptiveCpp can deliver performance in the absence of XNACK due to the automatic prefetching optimization. In this scenario, it outperforms AMD's roc-stdpar by an order of magnitude.

Using three mini-apps, we have shown that AdaptiveCpp can provide speedups for code relying on C++ parallel STL idioms across hardware from three GPU vendors simply by recompiling. The same was not always true for the vendor solutions running on their own hardware. For example, Intel's new ICPX stdpar support performed very poorly on memory-bound applications and does not appear to us to be useful in practice for such workloads. However, this is still

a very new feature in ICPX, and it will be interesting to monitor its progress in the future.

Our implementation has outperformed the vendor stdpar solutions on each platform for the majority of tested applications, achieving good fractions of the performance of native programming models and compilers: 75%-103% on NVIDIA A100, 65%-115% on AMD Instinct MI100 and 45%-102% on Intel Data Center GPU Max 1550. The minimum of the performance range was consistently higher for AdaptiveCpp than the vendor compilers on all platforms.

Using a complex application, LULESH, we have demonstrated how our optimizations, such as synchronization elision, have allowed us to outperform nvc++ on NVIDIA A100 for all problem size regimes by up to ≈80%. In this case, the automatic prefetch optimization was detrimental for performance. This implies that it might be worthwhile in the future to investigate mechanisms to automatically estimate the benefit of prefetching.

On AMD Instinct MI100, LULESH has highlighted severe problems in AMD's implementation of shared USM allocations. In this case, offloading has led to substantial performance loss, especially when XNACK was enabled. However, AdaptiveCpp's offloading heuristic has detected the performance loss, and took the decision to not offload, thus avoiding roc-stdpar's performance drop of two orders of magnitude. This illustrates that PSTL offloading may be detrimental, and implementations should attempt to estimate whether offloading is actually beneficial. However, our experience was limited to one AMD GPU architecture, Linux version and ROCm version. A future study investigating the behavior on newer AMD stacks might provide valuable information about the state and prospects of shared USM allocations on AMD hardware. The fact that NVIDIA, AMD, and Intel are now moving towards support for the stdpar model promises that such issues might decrease in the future.

We have demonstrated how AdaptiveCpp is the first stdpar implementation to provide performance across GPUs from AMD, NVIDIA, and Intel. Coupled with SYCL as a portable lower-layer when more control is desired, we believe it can be a powerful tool both for experienced users, as well as new users who wish to enter the world of accelerated computing.

ACKNOWLEDGMENTS

This project has received funding from the European Union's HE research and innovation programme under grant agreement No 101092877 (SYCLops project). This work used the Isambard UK National Tier-2 HPC Service (http://gw4.ac.uk/isambard/) operated by GW4 and the UK Met Office, and funded by EPSRC (EP/ P020224/1). We gratefully acknowledge resources provided by Intel on the Intel Developer Cloud. We would like to thank Tom Lin for inspiring discussions about the subject.

REFERENCES

[1] Aksel Alpay and Vincent Heuveline. 2020. SYCL beyond OpenCL: The Architecture, Current State and Future Direction of HipSYCL. In *Proceedings of the International Workshop on OpenCL* (Munich, Germany) *(IWOCL '20)*. Association for Computing Machinery, New York, NY, USA, Article 8, 1 pages. https://doi.org/10.1145/3388333.3388658

[2] Aksel Alpay and Vincent Heuveline. 2023. One Pass to Bind Them: The First Single-Pass SYCL Compiler with Unified Code Representation Across Backends. In *Proceedings of the 2023 International Workshop on OpenCL* (, Cambridge, United Kingdom,) *(IWOCL '23)*. Association for Computing Machinery, New York, NY, USA, Article 7, 12 pages. https://doi.org/10.1145/3585341.3585351

[3] Aksel Alpay, Bálint Soproni, Holger Wünsche, and Vincent Heuveline. 2022. Exploring the Possibility of a HipSYCL-Based Implementation of OneAPI. In *International Workshop on OpenCL* (Bristol, United Kingdom, United Kingdom) *(IWOCL'22)*. Association for Computing Machinery, New York, NY, USA, Article 10, 12 pages. https://doi.org/10.1145/3529538.3530005

[4] OpenMP ARB. 2022. *OpenMP*. https://www.openmp.org/

[5] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. 2009. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. In *Euro-Par 2009 Parallel Processing*, Henk Sips, Dick Epema, and Hai-Xiang Lin (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 863–874.

[6] Intel Corporation. 2022. Intel oneAPI DPC++/C++ Compiler. https://www.intel.com/content/www/us/en/developer/tools/oneapi/dpc-compiler.html

[7] Intel Corporation. 2023. *Intel oneAPI Base Toolkit Release Notes*. https://www.intel.com/content/www/us/en/developer/articles/release-notes/intel-oneapi-toolkit-release-notes.html

[8] Graham Lopez David Olsen and Bryce Adelstein Lelbach. 2020. *Accelerating Standard C++ with GPUs Using stdpar*. https://developer.nvidia.com/blog/accelerating-standard-c-with-gpus-using-stdpar/

[9] Tom Deakin, James Cownie, Wei-Chen Lin, and Simon McIntosh-Smith. 2022. Heterogeneous Programming for the Homogeneous Majority. In *2022 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. 1–13. https://doi.org/10.1109/P3HPC56579.2022.00006

[10] Tom Deakin, James Price, Matt Martineau, and Simon McIntosh-Smith. 2016. GPU-STREAM v2.0: Benchmarking the Achievable Memory Bandwidth of Many-Core Processors Across Diverse Parallel Programming Models. In *High Performance Computing*, Michela Taufer, Bernd Mohr, and Julian M. Kunkel (Eds.). Springer International Publishing, Cham, 489–507.

[11] Advancend Micro Devices. 2022. *ROCm*. https://github.com/RadeonOpenCompute/ROCm

[12] Advanced Micro Devices. 2023. *ROCm Standard Parallelism Runtime Implementation*. https://github.com/ROCmSoftwarePlatform/roc-stdpar

[13] Intel. 2022. *Level Zero*. https://github.com/oneapi-src/level-zero

[14] ISO. 2017. *ISO/IEC 14882:2017 Information technology — Programming languages — C++* (fifth ed.). International Organization for Standardization, Geneva, Switzerland. 1605 pages. https://www.iso.org/standard/68564.html

[15] Ian Karlin, Jeff Keasler, and Rob Neely. 2013. *LULESH 2.0 Updates and Changes*. Technical Report LLNL-TR-641973. 1–9 pages.

[16] Khronos SYCL working group. 2021. *SYCL 2020 rev 4 specification*. Standard. Khronos Group, Inc, Beaverton, OR, USA. https://www.khronos.org/registry/SYCL/specs/sycl-2020/pdf/sycl-2020.pdf

[17] Chris Lattner and Vikram S. Adve. 2004. LLVM: a compilation framework for lifelong program analysis & transformation. *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* (2004), 75–86.

[18] Wei-Chen Lin, Tom Deakin, and Simon McIntosh-Smith. 2022. Evaluating ISO C++ Parallel Algorithms on Heterogeneous HPC Systems. In *2022 IEEE/ACM International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. 36–47. https://doi.org/10.1109/PMBS56514.2022.00009

[19] Wei-Chen Lin, Simon McIntosh-Smith, and Tom Deakin. 2024. Preliminary report: Initial evaluation of StdPar implementations on AMD GPUs for HPC. arXiv:2401.02680 [cs.DC]

[20] Simon McIntosh-Smith, Matthew Martineau, Tom Deakin, Grzegorz Pawelczak, Wayne Gaudin, Paul Garrett, Wei Liu, Richard Smedley-Stevenson, and David Beckingsale. 2017. TeaLeaf: A Mini-Application to Enable Design-Space Explorations for Iterative Sparse Linear Solvers. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*. 842–849. https://doi.org/10.1109/CLUSTER.2017.105

[21] Aaftab Munshi. 2009. The OpenCL specification. In *2009 IEEE Hot Chips 21 Symposium (HCS)*. IEEE, 1–314.

[22] NVIDIA. 2022. *CUDA Toolkit Documentation*. https://docs.nvidia.com/cuda/index.html

[23] Andrei Poenaru, Wei-Chen Lin, and Simon McIntosh-Smith. 2021. A Performance Analysis of Modern Parallel Programming Models Using a Compute-Bound Application. In *High Performance Computing*, Bradford L. Chamberlain, Ana-Lucia Varbanescu, Hatem Ltaief, and Piotr Luszczek (Eds.). Springer International Publishing, Cham, 332–350.

[24] Michael Wolfe. 2021. Performant, Portable, and Productive Parallel Programming With Standard Languages. *Computing in Science & Engineering* 23, 5 (2021), 39–45. https://doi.org/10.1109/MCSE.2021.3097167