IWOCL 2024

The 12th International Workshop on OpenCL and SYCL

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

Aksel Alpay, Universität Heidelberg

Aksel Alpay, Universität Heidelberg, Vincent Heuveline, Universität Heidelberg

APRIL 8-11, 2024 | CHICAGO, USA | IWOCL.ORG

Can heterogeneous computing be expressed directly with C++?

Note that SYCL asks a closely related question:

Can heterogeneous computing be expressed with pure C++ syntax, and an additional C++ API?

and is thus a continuation of SYCL's line of thought!

- C++ 17 parallel STL (PSTL) provides mechanisms to express data parallel computation: `std::for_each`, `std::transform`, `std::transform_reduce`, `std::fill`, `std::copy`...

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


std::vector<T> data = ...
std::for_each(Policy, data.begin(), data.end(), [](auto& x){ x += 1;});
```

`Policy` may be:

- `std::execution::seq` – algorithm must be executed sequentially
- `std::execution::par` – algorithm may be parallelized
- `std::execution::par_unseq` – algorithm may be parallelized and vectorized. Only vectorization-safe code is allowed inside the algorithm (e.g. no locks)

- Especially the `par_unseq` policy maps well to data-parallel accelerators like GPUs
- It is attractive to consider offloading such data parallel C++ constructs to data parallel accelerators
  - Lower barrier of entry into heterogeneous computing
  - Highly idiomatic
  - Perhaps get speedup for existing regular C++ code simply by recompiling with offloading compiler?
- This programming model is typically referred to as stdpar (standard parallelism)
- Stdpar as offloading model was notably pioneered by NVIDIA's nvc++ compiler for NVIDIA hardware
- Recently, other vendors have been proposing their own solutions: AMD roc-stdpar, Intel `icpx -fsycl-pstl-offload=gpu`
- In a similar time frame, the AdaptiveCpp project also started working on stdpar

Stdpar implementations

UNIVERSITÄT HEIDELBERG ZUKUNFT SEIT 1386

Typical stdpar implementation design: Vendor model compiler → vendor model algorithm library

- nvc++ → thrust (CUDA)
- roc-stdpar → rocThrust (HIP)
- icpx -fsycl-pstl-offload=gpu → oneDPL (SYCL)

Additionally,

- Compilers generally unaware of the stdpar model except to enable basic prerequisistes (memory management, kernel outlining)
- Once the code has been compiled, existing stdpar compilers generally offload unconditionally

Our contributions

Meet **AdaptiveCpp stdpar**: Stdpar support integrated into the AdaptiveCpp SYCL implementation.

- First stdpar implementation that is both open-source and based on SYCL;
- First stdpar implementation to demonstrate performance across Intel, NVIDIA and AMD GPUs;
- First stdpar implementation to diverge from the library-focused design for performance and functionality benefits
  - Tighter integration of the stdpar model with the compiler
  - Additional, new optimizations/features, including synchronization elision, automatic prefetching of data, an offload heuristic and a pointer validation layer
- Substantial perf improvements over other stdpar compilers

Stdpar implementations in comparison

Implementation	Supported hardware	Open source?	Based on
NVC++	CPUs,NVIDIA GPUs	No	CUDA+thrust
roc-stdpar	AMD GPUs	Yes	HIP+rocThrust
icpx	Intel (others?)	No	SYCL+oneDPL
AdaptiveCpp	CPUs, Intel GPU, NVIDIA GPU, AMD GPU	Yes	SYCL +own algorithms library +compiler extensions +runtime extensions

- AdaptiveCpp stdpar is not focused on hardware from one vendor
  - By default generates a universal binary that targets all supported devices (CPUS/Intel GPUs/NVIDIA GPUs/AMD GPUs)
- Start app development in high-level C++ standard parallelism, progressively move to SYCL as more control is needed for optimization

Implementation

General architecture

- Provide custom `algorithm`, `execution`, `numeric` headers
- Add new overloads for offload-capable algorithms for `par_unseq` policy
  - Set of offload-aware algorithms is still smaller than for competing solutions[1]
  - Initial goal was to provide an innovative framework; quantity in terms of algorithms can always be improved later
- Algorithms where offload is not implemented will work, but run on the host.
- Header interception and additional compiler logic enabled using `--acpp-stdpar`

[1] https://github.com/AdaptiveCpp/AdaptiveCpp/blob/develop/doc/stdpar.md

Memory management

- C++ has a flat memory hierarchy, and is unaware of multiple distinct memory spaces (e.g host vs device memory)
- $\rightarrow$ Memory needs to be available on device without the user calling special memory allocation functions or explicit data transfers
- Compiler/runtime in general cannot determine all allocations that might be used on device, e.g.
  - indirect access: Additional pointers are loaded from memory, e.g. in pointer-based data structures (linked lists, trees, …)
  - Pointers to allocations might be disguised as integers

SYCL 2020 unified shared memory (USM) memory to the rescue?

- SYCL System USM: All host memory addresses are directly accessible on device.
  - Might be available e.g. if host/device are tightly integrated, the device is the host CPU, Linux HMM
  - No further action is needed, stdpar memory management solved!
  - But only rarely available in practice…
- SYCL Shared USM: Memory automatically migrates between host and device as needed
  - Typically implemented with hardware emitting page faults, and driver migrating memory pages
  - No explicit data transfers needed, but special memory allocation/deallocation functions required: `sycl::malloc_shared()`, `sycl::free()`
  - More widely available (AMD, Intel, NVIDIA)

For generality, an stdpar implementation cannot assume that system USM is available. **In the following, we assume shared USM**.[2]

[2]For the system USM case, AdaptiveCpp supports `--acpp-stdpar-system-usm` which disables the additional shared USM memory management handling

Supporting stdpar through shared USM requires a **memory management interposition layer**:

- Intercept all allocations/deallocations (`new`, `delete`, `malloc`, `free`, `realloc`,...) and reroute to shared USM
- This is how all stdpar implementations generally work
- Major limitation: Only works with heap allocations. What happens if a user passes in a stack pointer?

AdaptiveCpp:

- Allocation handled by locally replacing function calls with compiler
- Deallocation handled by globally intercepting symbols → Can deallocate both USM and regular memory everywhere

This is **very** tricky to get right![3]

Challenges (examples)

- Stack overflows/infinite recursion due to intercepted memory management inside SYCL
- Negative performance impact on submission latency
- ODR-resolved functions may cause local interposition to not trigger correctly
- Memory allocation/deallocation requests when drivers are unavailable (early during program startup or late during shutdown)

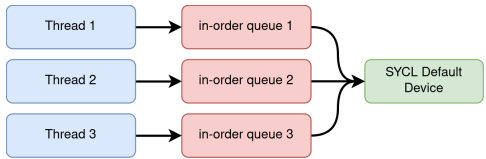
AdaptiveCpp solutions (examples)

- Conditionally disable interposition when recursing, or when runtime is unavailable
- Semi-lock-free allocation tracking data structure that can be used to determine whether a pointer is USM independently from the SYCL runtime
- Disable allocation interposition inside call graph of SYCL headers/`sycl::` functions
- Full call graph duplication for the interposed and non-interposed allocation cases, insertion of ABI tags to distinguish symbols

[3] especially for multi-backend SYCL implementations

Execution model

- C++ does not provide any information on the target device in its API
  - → All stdpar implementations are challenged when multiple devices need to be used
  - AdaptiveCpp stdpar uses SYCL default device[1]
- thread-local in-order SYCL queues



- C++ stdpar model requires waiting after every kernel launch
  - AdaptiveCpp is the only stdpar implementation that can detect and elide unnecessary synchronization (more later)

[1]controllable with `ACPP_VISIBILITY_MASK`. Multiple devices can be used e.g. via MPI.

Correctness

Seamless integration into C++ requires that all C++ features allowed in `par_unseq` algorithms work. This clashes with device code limitations, e.g. for SYCL:

- Kernel lambdas must capture by-value, not by reference
- Host pointers may not be dereferenced on device (unless system USM)
- function pointers, virtual functions not allowed
- exceptions not allowed
- non-trivial types may only be passed as SYCL kernel arguments if they adhere to the SYCL device-copyable concept and specialize `sycl::is_device_copyable`
- builtin functions (e.g. math functions) need to be from `sycl::` namespace, e.g. `std::sin()` is not allowed

Two categories of solutions:

1. Add extensions to support these features on device
2. Detect whether unsupported functionality is used, and if so, don't offload (might require delayed/different compiler diagnostics)

No stdpar implementation currently solves all of these restrictions.
AdaptiveCpp stdpar attempts to address/mitigate **the most common issues**. → AdaptiveCpp stdpar can handle cases other implementations cannot handle.

Correctness examples

- AdaptiveCpp supports `std::` math builtins in device code
  - New LLVM pass that remaps libc builtins to AdaptiveCpp builtins
  - roc-stdpar only supports this partially (e.g. `std::cbrt` does not work)
- AdaptiveCpp supports capture-by-reference
  - AdaptiveCpp validates all kernel pointer arguments
    - If a host pointer is used, the algorithm is not offloaded.
  - icpx does not compile kernels with capture-by-reference, nvc++ and roc-stdpar crash if the pointer is a host pointer
- AdaptiveCpp supports non-trivial data structures
  - icpx does not allow such types in kernels unless `sycl::is_device_copyable` is specialized
- AdaptiveCpp supports memory ownership transfer to program components not compiled with stdpar compiler
  - this can happen easily, e.g. if `std::shared_ptr` is used both by the stdpar-compiled program and external libraries
  - roc-stdpar crashes in this case (only intercepts deallocation locally)

Optimizations

- Simpler execution model than SYCL; bypass some unneeded SYCL layers (e.g. DAG construction)
  - → Lower submission latency than SYCL
- USM allocation/free is more expensive than regular allocation/free
  - → Introduce USM memory pool and serve allocations from pool using custom allocator
- Automatic emission of `queue::prefetch()` calls for allocations used in kernels
  - Emitting single data transfer may be more efficient than having separate data transfers for each page fault
  - Supported prefetch modes
    - `always` - always prefetches
    - `never` - prefetching is disabled
    - `first` - (default) only prefetch the first time an allocation is used on device
    - `after-sync` - only prefetch for the first operation after a barrier

Optimizations: Synchronization elision


```
1 for(int i=0; i < num_iters; ++i) {
2   std::for_each(par_unseq, data.begin(), data.end(), ...);
3   std::transform(par_unseq, data.begin(), data.end(), data.begin(),...);
4 }
5 access_results(data);
```

- stdpar algorithms in above could be executed asynchronously until results are accessed
- Waiting after every stdpar call can be expensive!
- AdaptiveCpp detects such unnecessary barriers and elides them by delaying synchronization for as long as possible in new LLVM pass
- Currently only works within one TU; calls to functions defined externally prevent elision
  - Move optimization to LTO pipeline?
- Does not work for algorithms that need to directly return a result (e.g. `transform_reduce`)

Optimizations: Offloading heuristic

- Offloading is not always beneficial (e.g. latency for small problems)
  - $\rightarrow$ Introduce offloading heuristic
- At runtime, record the execution chain of operations to predict the next ones
- Predicts offloaded/non-offloaded runtime for the next N operations
  - Maintain database with previous kernel runtimes
  - Both for offloaded and non-offloaded case – needs host run to calibrate performance
- Estimates data transfer cost using allocations passed as kernel arguments

Note: This heuristic worked well for our use cases, but we do not claim that it is perfect!

Optimizations: Comparison

Optimization	AdaptiveCpp	icpx	nvc++	roc-stdpar
Memory pool	Yes	?	Yes	Yes
Synchronization elision	Yes	No	No	No
Automatic prefetch	Yes	No	No	No
Offloading heuristic	Yes	No (?)	No	No

Evaluation

Setup

Test hardware:

- System 1: 2x AMD Epyc 7713 (Isambard P3)
- System 2: AMD Epyc 7543P, 4x AMD Instinct MI100 (Isambard P3)
- System 3: AMD Epyc 7543P, 4x NVIDIA A100 (Isambard P3)
- System 4: 2x Intel Xeon Platinum 8480+, 4x Intel Data Center GPU Max 1550 (IDC)

Software:

- AdaptiveCpp `f2c2960` built against LLVM 15/libstdc++ 12, using generic single-pass compiler
- oneAPI 2024.0.2
- CUDA 12.1, NVHPC 23.5
- ROCm 5.4.1, roc-stdpar `8c57cd0`

AMD GPUs and XNACK

- AMD GPUs depend on hardware feature called XNACK for shared USM
- Important for instruction retry in case of page fault
- Without XNACK, ROCm maps shared USM to device-accessible host-memory
  - Every memory access needs to traverse PCIe…
- XNACK is elusive:
  - Most consumer GPUs lack hardware support
  - Not enabled on most HPC systems
  - Needs non-standard Linux kernel arguments (cannot be enabled by unprivileged users)
- We are lucky, our system supports XNACK
- In practice, most systems currently do not → non-XNACK performance is more important than XNACK performance!

Babelstream

BabelStream[1] supports the stdpar model – useful to investigate the impact of the stdpar shared USM interposition layer!

- Compare to SYCL version of the code with explicit device allocations
- XNACK results failed to validate – only non-XNACK results are shown on AMD
- the `icpx -fsycl-pstl-offload`-compiled BabelStream crashed inside internal SYCL header code.
  - As a workaround, we present results with direct calls to oneDPL and explicit `sycl::malloc_shared()` calls. This is a simpler problem for drivers and not exactly the same!

[1] Tom Deakin et al. (2016): GPU-STREAM v2.0: Benchmarking the Achievable Memory Bandwidth of Many-Core Processors Across Diverse Parallel Programming Models.

Babelstream

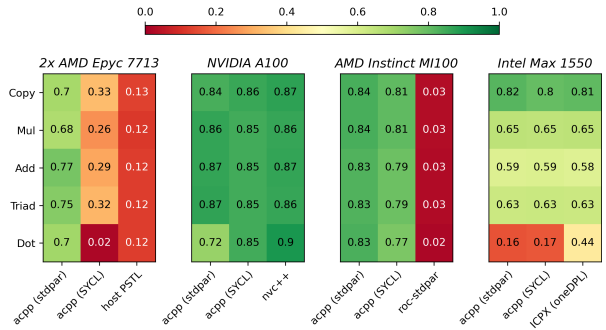


Figure: BabelStream perf as a fraction of theoretical peak for AdaptiveCpp and vendor compilers

- …outperforms SYCL on CPU (lower overhead)
- roc-stdpar is not competitive without XNACK
- acpp does not need XNACK for perf! (**auto-prefetch!**)
- stdpar shared USM can be very efficient!

Mini-apps

- miniBUDE[1]: Compute-bound molecular docking mini-app
- CloverLeaf[2]: 2D Hydrodynamics mini-app
- TeaLeaf[3]: Heat equation solver

Investigate how stdpar compilers perform compare to native vendor model:

- nvcc-compiled CUDA on NVIDIA;
- hipcc-compiled HIP on AMD;
- icpx-compiled SYCL on Intel

All AdaptiveCpp prefetch modes were tested; the best were always either `first` or `never`.

[1] Poenaru et al. (2021): A Performance Analysis of Modern Parallel Programming Models Using a Compute-Bound Application.

[2] Lin et al. (2022): Evaluating ISO C++ Parallel Algorithms on Heterogeneous HPC Systems.

[3] McIntosh-Smith et al. (2017): TeaLeaf: A Mini-Application to Enable Design-Space Explorations for Iterative Sparse Linear Solvers.

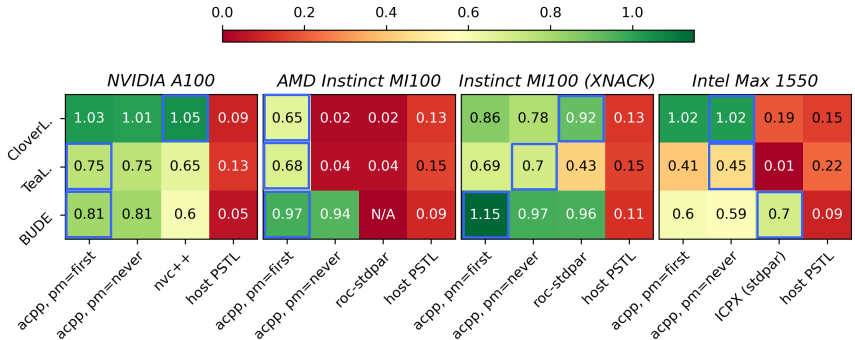


Figure: Stdpar performance normalized to native model. Fastest results are highlighted.

- AdaptiveCpp outperforms vendor stdpar models for 2/3 apps on all systems
- …Sometimes by an order of magnitude
- AdaptiveCpp delivers reliable performance, always faster than host PSTL
- icpx stdpar is not competitive except for compute-bound miniBUDE → issue in memory interposition layer?

LULESH[1]

- Shock hydrodynamics mini-app
- Very challenging for stdpar:
  - Frequent allocations and deallocations
  - Lots of indirect access
  - Latency-sensitive → sensitive to synchronous stdpar operations

[1] Karlin et al. (2013): LULESH 2.0 Updates and Changes

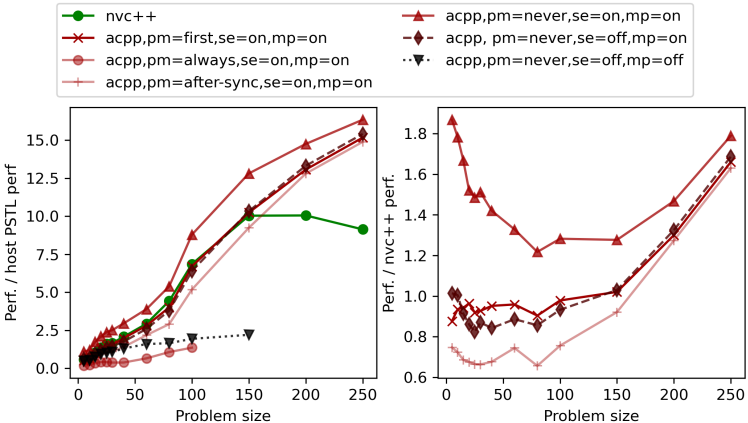


Figure: LULESH on A100 (pm: prefetch mode, se: synchronization elision, mp: memory pool)

- Memory pool is an important base optimization!
- Synchronization elision allows AdaptiveCpp to outperform nvc++ by up to 80%. (≈ 80% of barriers elided)
- Prefetching is detrimental for this app

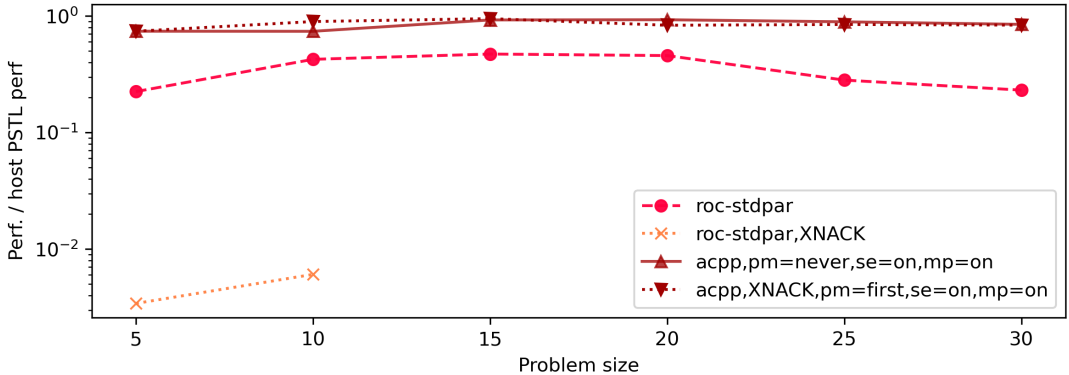


Figure: LULESH on Instinct MI100

- ROCm stack severely challenged
- Crash if prefetches are used
- XNACK performance even worse
- AdaptiveCpp detects the issue and decides to not offload
  - Offloading heuristic is important!

What about LULESH on Intel?

- ICPX refuses to compile LULESH (capture-by-reference)
- AdaptiveCpp compiles, but hangs inside Intel driver. Potential driver issue?
  - Verified to run fine on Intel UHD 620 and 630 iGPU
  - (Results are not exciting there; slower than host PSTL so offloading heuristic decides to not offload)

Conclusion

- Integration of heterogeneous computing directly into C++ is possible
- Requires deep compiler and runtime integration for best results
- AdaptiveCpp is the first stdpar implementation to attempt this
  - synchronization elision, automatic prefetch, direct calls to lower-level runtime functionality…
- Outperforms vendor stdpar solutions for majority of mini-apps on all platforms, and nvc++ by 80% on A100 with LULESH
- Unlike roc-stdpar, performs well without XNACK (the expected case!)
- None of the perf weaknesses that roc-stdpar and icpx exhibited
- Additional compiler improvements after paper submission. Expect 10-20% faster kernels with AdaptiveCpp 24.02…

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.