June 2024

UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

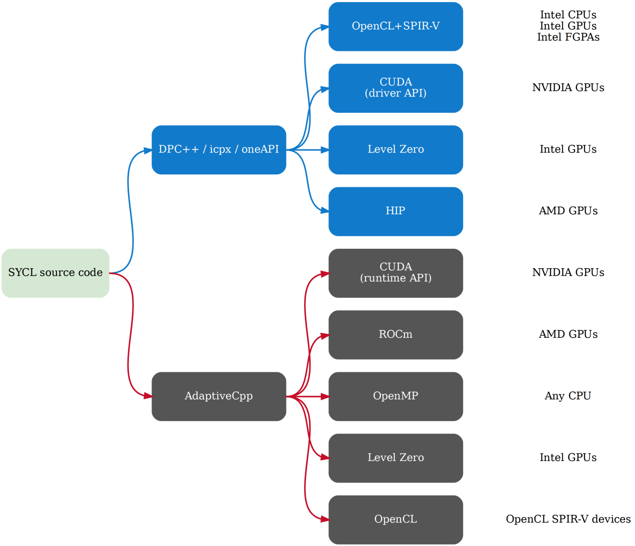
# AdaptiveCpp: A modern compiler and runtime stack


Aksel Alpay
Heidelberg University

# AdaptiveCpp

- Independent
- community-driven
- Portable
- open-source[a]
- supports SYCL and standard C++ parallelism offloading

[a] `https://github.com/AdaptiveCpp/AdaptiveCpp`

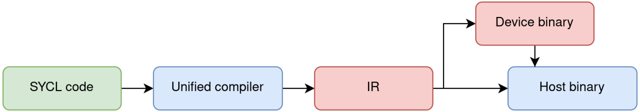


# AdaptiveCpp compiler overview

Covering different use cases:

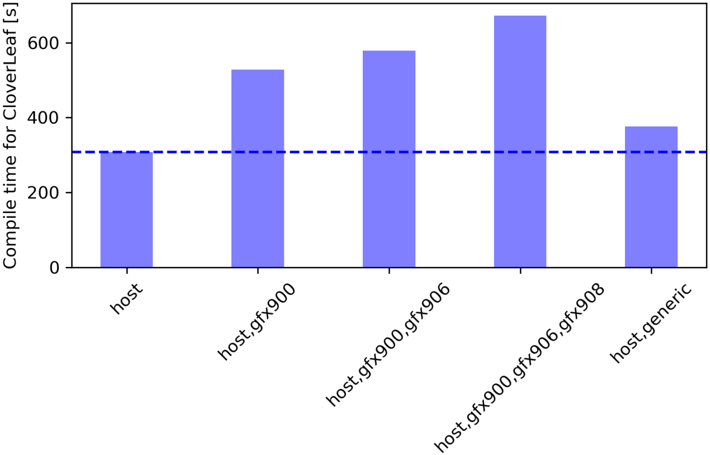
1. Deployment simplicity (at expense of functionality):
   - **Library-only**, no dedicated compiler support
   - Available for OpenMP, nvc++
2. Interoperability, code migration:
   - **Run as part clang CUDA/HIP toolchains**
   - mix-and-match SYCL and CUDA/HIP code in kernels
   - minimal AdaptiveCpp-specific compiler support
3. Performance, portability, functionality:
   - **generic JIT compiler**
   - default, main compiler
   - fully aware of AdaptiveCpp semantics
   - best performance, best compile times, best binary portability

# Generic JIT compiler




- Unified code representation across backends
- generic JIT targets:
  - native host, amdgcn, nvptx64, SPIR-V
- Modular runtime backend plugins
  - OpenMP, CUDA, HIP, OpenCL, Level Zero
- Only needs to parse code once, independently of devices that code will be executed once
  - Other compilers, e.g. DPC++, parses the code once for host plus once for every generated device binary format (SPIR-V, PTX, amdgcn ISA)

# Generic JIT compiler - compile times



# AdaptiveCpp is unique

UNIVERSITÄT HEIDELBERG ZUKUNFT SEIT 1386

AdaptiveCpp is the only SYCL implementation…

- with a unified host-device compiler (only needs to parse the code once in generic JIT mode)
- that can generate a binary that can offload to CPUs and "all the GPUs" from a single compilation step
  - Specifiying targets is not needed: `acpp -o test test.cpp`
- that has a unified code representation across backends
- that has a unified JIT infrastructure across backends
- that can automatically include runtime knowledge in kernel code generation
- that supports any LLVM-supported CPU with full JIT capabilities
- (... and more, e.g. HIP/CUDA source interoperability)

# Why community-driven compilers make sense

UNIVERSITÄT HEIDELBERG ZUKUNFT SEIT 1386

- Vendor-specific hardware knowledge is not required to build a compiler frontend or programming model
- Compiler backends (which require vendor-specific knowledge) are freely available (e.g. LLVM)
- **Nowadays, anybody can build high-performance compiler stacks!**

- We can just build the compilers ourselves, for the programming models we want to use!
- Compilers and programming models should not have to be subject to vendor politics.
- Vendor lock-in concerns are real, but can be easily avoided by not relying on vendor compilers

AdaptiveCpp is **only vendor-independent** production-grade SYCL and standard C++ offloading solution

- Ensures vendor independence of SYCL as an open standard
- Make SYCL resilient to vendor politics

# Rising popularity

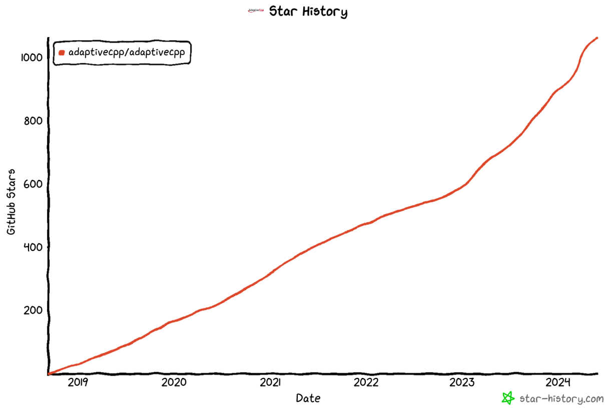
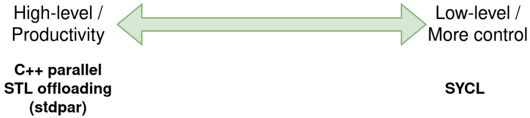



**Figure:** Number of github stars over time

- `https://github.com/adaptivecpp/adaptivecpp`
- Growing discord server: `https://discord.gg/s2p7Vccwh3` (link in AdaptiveCpp readme)

# For all programmers

UNIVERSITÄT
HEIDELBERG
ZUKUNFT
SEIT 1386




- Choose your desired abstraction model
- Mix-and-match within the same application
- E.g., start with C++ PSTL algorithms, drop to SYCL where more control is needed.
  - They complement each other, both may be needed for a full coverage of all needs

## For high-level use cases: C++ parallel STL offloading (stdpar)

# Stdpar

UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

- C++ 17 parallel STL (PSTL) provides mechanisms to express data parallel computation: `std::for_each`, `std::transform`, `std::transform_reduce`, `std::fill`, `std::copy` ⇒ We can offload those!
  - Very idiomatic and productive
  - Perhaps get speedup by recompiling existing C++ program?
- 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
- nowadays AMD (roc-stdpar) and Intel (icpx) support it as well for their hardware
- AdaptiveCpp is currently the **only stdpar solution that can target all of Intel/NVIDIA/AMD GPUs** robustly (including creating a binary that can offload to any of those)

# Stdpar example code

```
// No GPU-specific memory management needed
std::vector<float> v1 = get_input1(problem_size);
std::vector<float> v2 = get_input2(problem_size);
std::vector<float> result(problem_size);
// Triad will be offloaded to GPU
// Note: No way to manage devices, or asynchronous execution
std::transform(std::execution::par_unseq,
  v1.begin(), v1.end(), v2.begin(), result.begin(),
  [](float x, float y){
  return x + y;
});
// Implicit barrier at the end
```

Note: This is a very basic example; the stdpar model is more powerful (e.g. free indexing)

# AdaptiveCpp Stdpar Offloading

- How does it work (rough idea)?
  - Intercept all memory management (exception: systems with true unified shared memory, e.g. Grace-Hopper, MI300A, HMM)
  - Make all allocations GPU-accessible (e.g. `sycl::malloc_shared`)
  - Intercept and offload calls to C++ PSTL functions, if possible
- Optimizations not implemented by other PSTL offloading solutions
  - Automatic prefetching
  - Optimized memory pool
  - Offloading heuristic
  - Synchronization elision

See my paper for details. A. Alpay et al.: AdaptiveCpp Stdpar: C++ Standard Parallelism Integrated Into a SYCL Compiler

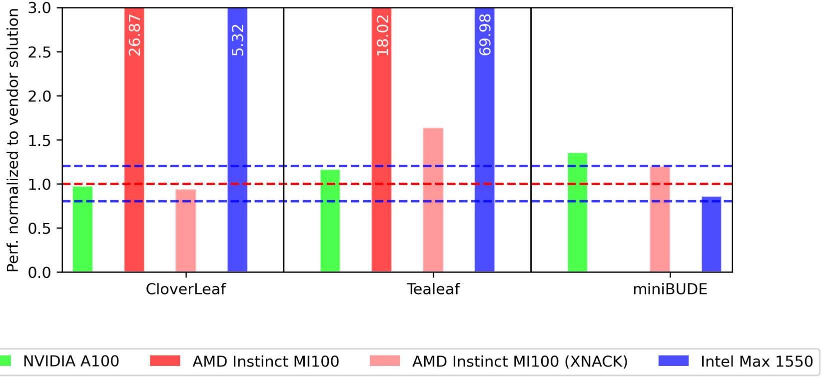



**Figure:** AdaptiveCpp stdpar perf normalized to vendor stdpar compiler (nvc++, roc-stdpar, icpx)

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

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

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

# AdaptiveCpp vs nvc++ with LULESH

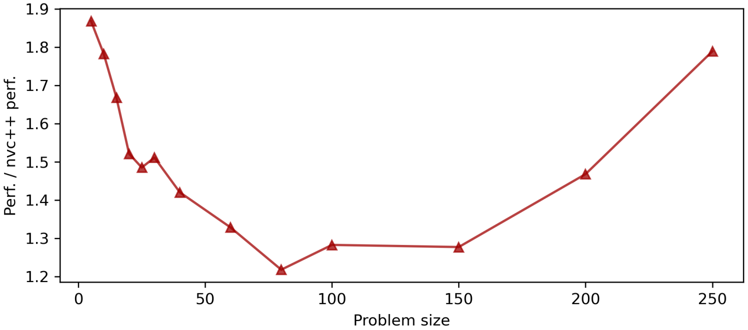



**Figure:** AdaptiveCpp speedup over NVC++ on NVIDIA A100 for LULESH[1] stdpar

- Memory pool issues with nvc++ for large problem sizes
- AdaptiveCpp synchronization elision yields large advantage for small problems

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

## For low-level use cases: Runtime modification of kernels

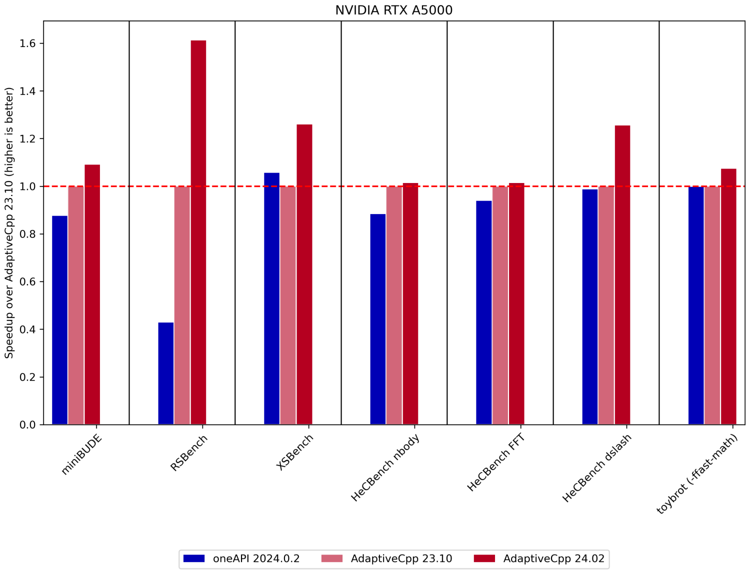
AdaptiveCpp is the only SYCL implementation with a unified JIT compiler across all backends. Leverage it!

# Automatic JIT-time optimizations

Enabled if `ACPP_ADAPTIVITY_LEVEL` $> 0$

- Hard-wiring of work group sizes
- Optimizer is informed about the configuration of the kernel launch at runtime
- Improved register scheduling
- Other optimizations, e.g. different code paths if problem size fits in 32bit integers

Note: Persistent kernel cache on disk. No overheads for future application runs once kernels have been JIT-compiled




Note: Different compiler defaults! (e.g. icpx uses `-ffast-math` by default)

# Specialization constants



Only SYCL implementation to support specialization semantics uniformly across all targets:

```
sycl::queue q;
sycl::specialized<float> scaling_factor = // some runtime value
float* data = ...
q.parallel_for(range, [=](auto idx){
  // The JIT compiler will treat the value of scaling_factor as a constant at JIT-time.
  // E.g, if scaling_factor is 1 at runtime, the compiler may generate an empty kernel.
  data *= scaling_factor;
});
```

Enabled if `ACPP_ADAPTIVITY_LEVEL` > 1:

- Automatic detection of invariant kernel arguments & hardwiring them as constants during JIT-time

# Dynamic functions

UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

- Replace functions with definitions selected at runtime
- Runtime kernel assembly
- Can be used to e.g.
  - build your own, user-controlled kernel-fusion engine
  - “JIT-time polymorphism”

# Dynamic functions II

UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

```
SYCL_EXTERNAL void operator_a(sycl::item<1> idx, int* data) { ... }
SYCL_EXTERNAL void operator_b(sycl::item<1> idx, int* data) { ... }

void run(sycl::item<1> idx, int* data);

int main() {
  sycl::queue q;
  int* data = ...
  sycl::jit::dynamic_function_config dfc;
  dfc.define_as_call_sequence(&run,
    {&operator_a, &operator_b});
  q.parallel_for(myrange, dfc.apply([=](sycl::item<1> idx){
    run(idx, data);
  }));
  q.wait();
}
```

# AdaptiveCpp 24.06 incoming



Stay tuned! AdaptiveCpp 24.06 scheduled to be released in early July…

- Improved SYCL 2020 compliance and features (e.g. reductions)
- Advanced JIT programming: Dynamic functions, detection of invariant kernel arguments…
- Improved parallel STL offloading support (`std:.atomic/std::atomic_ref` in device code, more execution policies)
- Latency improvements in runtime library
- Additional compiler optimizations