Cppless: Single-Source and High-Performance Serverless Programming in C++

MARCIN COPIK, ETH Zurich, Switzerland
LUKAS MÖLLER, ETH Zurich, Switzerland
ALEXANDRU CALOTOIU, ETH Zurich, Switzerland
TORSTEN HOEFLER, ETH Zurich, Switzerland

The rise of serverless computing introduced a new class of scalable, elastic and widely available parallel workers in the cloud. Many systems and applications benefit from offloading computations and parallel tasks to dynamically allocated resources. However, the developers of C++ applications find it difficult to integrate functions due to complex deployment, lack of compatibility between client and cloud environments, and loosely typed input and output data. To enable single-source and efficient serverless acceleration in C++, we introduce *Cppless*, an end-to-end framework for implementing remote functions which handles the creation, deployment, and invocation of serverless functions. *Cppless* is built on top of LLVM and requires only two compiler extensions to automatically extract C++ function objects and deploy them to the cloud. We demonstrate that offloading parallel computations, such as from a C++ application to serverless workers, can provide up to 59x speedup with minimal cost increase while requiring only minor code modifications.

CCS Concepts: • **Software and its engineering → Compilers**; *Cloud computing*; • **Computer systems organization → Cloud computing**.

Additional Key Words and Phrases: serverless, function-as-a-service, faas, cloud computing, c++, llvm, compiler

ACM Reference Format:
Marcin Copik, Lukas Möller, Alexandru Calotoiu, and Torsten Hoefler. 2025. Cppless: Single-Source and High-Performance Serverless Programming in C++. *ACM Trans. Arch. Code Optim.* 1, 1, Article 1 (January 2025), 30 pages. https://doi.org/10.1145/3747841

1 Introduction

Serverless functions have taken cloud systems by storm. Stateless, short-lived, and isolated functions execute on dynamically allocated cloud resources, and the programming model of Function-as-a-Service (FaaS) hides the software and hardware stacks of the cloud from the user. Functions offer a highly scalable and elastic offloading of computations to dynamically allocated parallel workers, with up to 6000 new function containers in a minute on commercial cloud platforms [1]. While serverless has initially gained popularity in web development and API integration, functions have been recently used for parallel and compute-intensive tasks such as data analytics, machine learning training, compilation, and high-performance computing [2–7].

In the pay-as-you-go system of FaaS, users are charged for each millisecond of active computation in a function. In a traditional deployment to virtual machines in Infrastructure-as-a-Service (IaaS), optimizations primarily improve the responsiveness and throughput of services. However, these improvements might not immediately lead to decreased costs when rescaling the deployment is not feasible. On the other hand, serverless functions immediately benefit from optimized code and shortened runtime as every millisecond saved directly decreases the cost of running an application

Authors' Contact Information: Marcin Copik, ETH Zurich, Switzerland, marcin.copik@inf.ethz.ch; Lukas Möller, ETH Zurich, Switzerland, mail@lukas-moeller.ch; Alexandru Calotoiu, ETH Zurich, Switzerland, acalotoiu@inf.ethz.ch; Torsten Hoefler, ETH Zurich, Switzerland, torsten.hoefler@inf.ethz.ch.

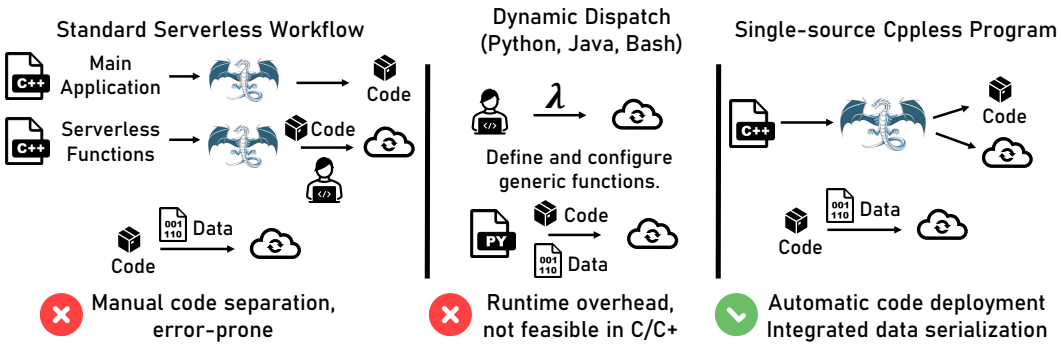

Fig. 1. Compilation in *Cppless*: single-source programming with automatic deployment at compilation time.

in the cloud. Shaving off seconds from functions is crucial in FaaS to provide lower costs than a persistent IaaS deployment for large and complex workloads [8, 9]. Hence, it is important that services are implemented on efficient backends and gain from the performance of compiled languages.

Even though serverless has been shifting to larger and more compute-intensive workloads, it is still dominated by languages such as JavaScript and Python. These languages make it particularly easy to implement serverless functions, thanks to embedded support for serialization, JSON, HTTP request handling, and an easy deployment mode with function code extraction at runtime. However, achieving high performance in these high-level and dynamic languages is difficult and requires programmers to offload hot spots to lower-level languages with the help of native extensions. Interpreted languages are particularly affected by high costs of cold startups (Section 5.2.4). Furthermore, many parallel and high-performance libraries and frameworks are already implemented in C/C++ and could be used for parallel computations in the cloud. Unfortunately, the drive toward a high-performance serverless cloud is hindered by the complex deployment model and lack of native integration of serverless functions into C++ applications.

Serverless C++ functions require developers to split functions from the main application and compile them separately, deploy to the cloud using interfaces that are not standardized and differ for each cloud platform, and execute through a vendor-specific REST and RPC APIs. While managed languages can use the bytecode and runtime introspection mechanisms to automatically extract function code from an application, existing C++ language capabilities are not sufficient for such a task. This problem is aggravated by the lack of compatibility between client and cloud software and hardware environments, and the conversion of statically typed data structures from and into the loosely typed JSON format (Section 2.2). This results in a convoluted process and a high entry barrier for serverless functions in high-performance applications. To benefit from serverless acceleration, parallel C++ applications need a framework that keeps the application and function code together to achieve high productivity, while avoiding code bloat and cloud vendor lock-in.

We resolve the aforementioned limitations by introducing *Cppless*, a single-source programming model and an end-to-end compiler for serverless functions in C++ (Figure 1). *Cppless* accelerates parallel C++ applications by shifting compute-intensive tasks to serverless functions, which are automatically created, deployed, and invoked by the framework. Similarly to GPU programming frameworks like CUDA and SYCL, which embed kernel code within C++ applications, we include remote serverless functions as regular function objects and C++ lambda functions. In the example of a Ray-Tracer application, using serverless with *Cppless* requires only 7 lines of offloading code and 18 lines of serialization instructions more than a multithreaded implementation, and provides up to 8.26x speedup against local computation on a virtual machine with 16 vCPUs. Obtaining

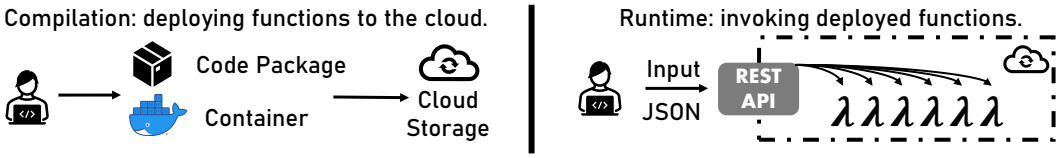

Fig. 2. **Development workflow in serverless**: functions are shipped to the cloud during deployment. At runtime, the main applications invoke previously created functions.

the result with functions costs less than 0.0027$ and is delivered in one second, while a virtual machine could take several minutes to boot, and keeping online even a small general-purpose virtual machine `t2.medium` with two virtual CPUs costs 0.0536$ per hour - almost twenty times more.

Cppless achieves these performance, cost, and productivity results by combining the serverless and non-serverless program parts in a single source code. To that end, we introduce the concept of **alternative entry points** to redirect the compilation of a single translation unit into multiple targets. Once the compiler detects a serverless function code inside a C++ translation unit, it splits the compilation path for the selected function to generate a separate executable (Section 3). To dispatch C++ lambda functions to the cloud, we extended the compiler with two additional features: **reflection** to access capture variables and **unique naming scheme** for these anonymous functions. *Cppless* automatically deploys alternative entry points to the cloud as serverless functions. A serialization library helps pack input data into binary format, and the framework encapsulates the vendor-specific process of deploying and invoking functions.

The *Cppless* compiler is built on top of the LLVM framework and requires only a few modifications to the `clang` codebase (Section 4). To seamlessly integrate serverless functions into applications, we implemented a high-level and templated user interface, which internally uses the language extensions added to support single-source programming. With just a few lines of code added to the application, users can offload native C++ computations to elastic and scalable serverless functions while retaining **the same compilation workflow**. With a set of microbenchmarks and parallel applications (Section 5), we demonstrate that *Cppless* enables efficient parallel offloading to the serverless cloud without compromising the productivity and safety of C++ programming.

In this paper, we make the following contributions:

- Serverless programming model for efficient offloading computations in C++ to the cloud.
- A C++ toolchain extended with alternative entry points and serializable lambda functions, allowing users for straightforward embedding of serverless functions into their applications. The toolchain is available on an open-source license,[1] and is accompanied by a paper artifact.[2]
- A C++ framework providing high-level abstractions that hides the complexity of cloud provider APIs, demonstrated with the example of AWS Lambda.

2 Why *Cppless*?

Serverless functions provide high scalability but introduce a division between functions shipped to the cloud and the main application code (Figure 2). While several frameworks have been developed to integrate functions and applications using them, they targeted high-level, interpreted, and dynamically-typed languages (Section 6). In *Cppless*, we attempt to achieve the same goal for C++: a standard-compliant interface for single-source serverless programming. To that end, we

[1] https://github.com/spcl/cppless [2] https://doi.org/10.5281/zenodo.15778386

first examine the low popularity of C++ in serverless. Then, we identify the unique challenges of bringing the statically typed and compiled C++ to the serverless world.

2.1 Programming Languages in Serverless

Serverless computing has been dominated by dynamically typed and interpreted languages like Python and JavaScript, covering 70% of serverless invocations [10] and 85% of open-source serverless applications [11]. The serverless landscape differs fundamentally from the rest of the programming world, as shown in the most recent TIOBE index [12]. There, Python and JavaScript achieved the first and sixth positions with ratings of 16.33% and 3.01%, respectively. On the other hand, C and C++ occupy the second and third place, respectively, achieving a cumulative rating of 19.5%. This difference is not caused by a lack of C++ applications that can benefit from serverless computing, as shown in the examples of Monte-Carlo simulations [6, 13] and distributed scientific analyses for large data processing [14]. Instead, we hypothesize that the low adoption is caused by insufficient support for integrating functions into C++ codebases and seamlessly deploying to the cloud.

Productivity FaaS is a new programming model that requires developers to adapt their codebases and toolchains to cloud deployment. A taxonomy of challenges experienced by serverless developers contains two categories directly related to *Cppless*: serverless application implementation and deployment [15], When considering the implementation of serverless applications, developers particularly struggle with integrating the deployment package and invoking functions through the cloud API. Deploying to FaaS causes issues in building a code package according to the size restrictions and expected format and deploying it to the cloud with available CLI tools. Thus, a productive programming framework needs to provide solutions for both categories to ease the burden and allow developers to focus on their application's logic and performance.

2.2 C++ Challenges in Serverless

Integrating serverless functions into C++ applications is difficult because of three major differences and challenges that are resolved in the *Cppless* framework.

Challenge #1: Compile-Time Dispatch Interpreted languages such as Python and Java permit extracting function code and embedding it with the input data into a POST request, allowing for dynamic function dispatch at runtime. On the other hand, C++ requires that the function code is compiled ahead of time, and it lacks runtime introspection and reflection mechanisms that could reliably and efficiently discover all dependencies of a selected function. Shipping all linked libraries would quickly lead to transmitting dozens and hundreds of megabytes, creating major performance overheads. Instead, the C++ program must be restructured to allow for separate compilation of the serverless functions and extended with serialization and invocation interface. Then, the results of the compilation should be uploaded to the cloud once the compilation is finished. This way, users can execute their C++ applications immediately after finishing the build process.

Challenge #2: Server Environment *Serverless* functions still execute on a *server*, which can be easily hidden in high-level and interpreted languages. However, this is not the case for compiled languages such as C++, where both the underlying architecture and ABI compatibility are of concern. For example, the user code might be running on an ARM notebook and link against the `libc++` and `libc` standard libraries, while the function code will execute in a Linux container on an x86 server, with `libstdc++` and `musl` available as implementations of the C++ and C standard libraries, respectively. Thus, more is needed than just to split the function code into a separate shared library; the application can no longer be compiled in a single environment with the same configuration. Languages such as C++ require a new programming model that will handle function code as a separate entity, while hiding this complexity from the user behind a uniform interface.

```cpp
double pi_estimate(int n); // Implementation of the Monte Carlo estimation

double compute_pi() {
  const int n = 100000000, np = 128;

  cppless::aws_dispatcher dispatcher;
  auto aws = dispatcher.create_instance();
  using config = lambda::config<cppless::lambda::with_memory<512>>;

  std::vector<double> r(np);
  // Define lambda function offloaded to serverless cloud
  auto fn = [=] { return pi_estimate(n / np); };
  for (auto& result : r) {
    // Invoke remote cloud functions. Returned values are written to the vector
    cppless::dispatch<config>(aws, fn, result);
  }
  cppless::wait(aws, np);  // Wait for all invocations to finish

  return std::reduce(r.begin(), r.end()) / np;
}
```

Fig. 3. Offloading parallel PI computation to AWS Lambda with *Cppless*. The program launches 128 functions in parallel to process 100 million samples.

Challenge #3: Static Typing Cloud platforms accept function input in a JSON format through a RESTful interface, which requires a dedicated conversion from strongly typed data structures. This is in contrast to microservices built on top of RPC communication, which hides the network transport of input arguments from the developer with the help of frameworks like Protobuf in gRPC and Transport in Thrift. Languages like Python and JavaScript come with native JSON support and serialization of objects to this format; C++ has no standard (de)serialization procedures.

C++ ensures type safety through compile-time verification. However, when function implementation is not single-source, and clients are connected to serverless environments only by loosely typed JSON, this compile-time safety is lost. When separated implementations of the invoking application and *serverless* function are joined only by the loosely typed JSON, type verification becomes the responsibility of the serialization runtime. JSON provides a limited possibility of encoding types, e.g., it is not possible to distinguish between integers of different sizes. This forces developers to implement manual JSON schemas, complex error handling, or adopt specialized serialization libraries that can properly encode diverse C++ types. This approach effectively shifts type checking from compilation to runtime, increasing the time and size overhead.

3 Designing a C++ Compiler for Serverless Functions

We begin by introducing the *Cppless* interface from a user perspective (Section 3.1). *Cppless* is designed as a group of C++ language extensions (Section 3.2), implemented in the LLVM compiler (Section 4). The extended compiler is combined with deployment tools to provide a seamless, single-step, and productive process of deploying C++ software to serverless (Section 3.3). For details on the design and implementation of *Cppless*, we refer readers to the work of Möller [16].

3.1 Parallel Computing with *Cppless*

We use the classic example of parallelizing Pi estimation (Figure 3) to demonstrate how users can implement serverless functions with *Cppless*, while not making their code dependent on additional, third-party cloud frameworks and keeping the function code united with the main application.

```
template <class Func>
struct ProcessBridge {
  int operator()() {
    auto alt_entry_name = gen_id<Func>(); // Spawn execution of the alternative entry point.
    return spawn(alt_entry_name);
  }
  __attribute((entry)) int main(int, char**) {
    Func func;
    deserialize(read_http_request(), func);
    respond_http(func());
  }
};
```

Fig. 4. `ProcessBridge` presents a simplified example of connecting a user-defined function object with an invocation of a remote serverless function.

In order to execute the computations on serverless functions, the user creates an instance of a dispatcher configured for the selected cloud system (line 7). In this case, we consider executing C++ functions as AWS Lambda functions. Within a dispatcher instance, all invocation requests share the same network resources, such as HTTP sessions, and each request is assigned a unique local identification. To invoke the function concurrently across 128 instances of an AWS Lambda function, the user calls the `cppless::dispatch` function (line 15), which will order the *Cppless* compiler to turn the user C++ lambda function (line 12) into an AWS Lambda function in the cloud. Users can optionally configure resources assigned to the cloud function, such as memory and temporary storage (line 8). The compiler will convert the configuration to metadata and attach it to the generated function code. As a final compilation step, *Cppless* will upload the generated function code to the cloud and create a function using the cloud provider APIs.

At runtime, the call to `cppless::dispatch` triggers an asynchronous function invocation by sending an HTTP request. The values of `n` and `np` captured in the C++ lambda function are serialized (line 12), and *Cppless* uses internally the C++ library `cereal` [17] for that task. The dispatcher selects the AWS Lambda function to be invoked through the unique type identification generated by the *Cppless* compiler. The third parameter of this function specifies where the result should be stored. Then, `cppless::wait` blocks until all invocations are finished (line 17). The *Cppless* runtime deserializes the output of function execution, which can be read directly by the user and merged to the final result (line 19). In addition to the return value, each invocation returns the cloud-assigned unique invocation identifier and a flag set to true if the invocation is cold.

3.2 C++ Language Extensions

Alternative entry points are the most crucial part in the *Cppless* compilation flow: they enable a single compilation unit to expose multiple entry points, letting us generate code for many serverless functions from a single C++ translation unit. Complemented with the second extension of lambda reflection, the compiler can now dispatch C++ function calls to the cloud.

Alternative Entry Points. *Cppless* exports the code of serverless functions to a separate compilation path by defining *alternative entry points*. Many programming languages define the concept of an entry point, which is a function executed when the program is started. From the entry point, the control flow can diverge and is governed by the language's semantics. In C-family languages, this function is usually called `main`, which is automatically called at the begin of execution.[3]

[3] The actual start is typically done through the `_start` function which is the entry point.

```cpp
template<typename Func>
void serialize_lambda(Func && lambda) {
  double arg1 = 42; std::string arg2 = "capture";
  auto lambda = [arg1, arg2]() { std::cout << arg1 << " " << arg2 << std::endl; };

  // The number of captured values is 2
  constexpr int capture_count = decltype(lambda)::capture_count();
  // Returns 42, type of variable: double
  auto captured_arg1 = lambda.capture<0>();

  lambda.capture<0>() = 43;
  // Will print "43 capture"
  lambda();
}
```

Fig. 5. Example demonstrating our language extension: lambda reflection. *Cppless* allows users and libraries to inspect the capture, their types, and values.

From a user perspective, alternative entry points are annotations added to a function declaration. Adding this annotation affects the compilation process by creating a separate executable or library where the the main function is replaced with the body of the alternative entry point function. However, users are not expected to directly use the alternative entry points; nor do they need to be aware of their existence, as this compiler feature is hidden behind the `cppless` interface. Function objects are used in composition with entry points to implement *bridge classes*, which use template programming to provide an interface to an alternative entry point that they define (Figure 4). This process can be used to model and deploy serverless functions: The client-side representation of a serverless function is an instance of the bridge class, the instantiation of this bridge class also automatically registers an alternative entry point that the runtime can use to create an invocation.

Function Serialization. In *Cppless*, serverless functions are defined as function objects, including C++ lambda functions, which define their own custom type. To seamlessly dispatch function invocations, *Cppless* needs to serialize function arguments. In the case of lambda functions, serialization also includes the state variables captured in the closure. To that end, we introduce a second extension: a compile-time, constexpr-compatible reflection mechanism for lambda functions (Figure 5). The reflection exposes direct accessors to the hidden unnamed capture members. The template member function `capture` returns an l-value reference that can be used both to read and to write the individual unnamed capture members. Thus, the host serialization can read the type and value of a state variable, while the serverless function can update the state based on the deserialized input data. Thus, *Cppless* can serialize functions if all variables captured in the function are serializable.

Why Language Extensions? Similarly to single-source GPU frameworks, *Cppless* compiles application code into multiple objects, which cannot be implemented with standard C++ features. Separating entry points allows us to optimize functions by targeting different architectures and adapting compilation options. Furthermore, a solution using a single executable with runtime dispatch would require additional code changes to adapt `main` function and assist discovery of serverless functions, increasing the burden on the user and making the tool less convenient.

Serializing lambda functions is not possible in C++ due to lack of reflection. Future versions of C++ might include reflection support [18]; *Cppless* supports offloading lambda functions today and can adopt standard-compliant solutions once they become available. Furthermore, the internal storage of lambda capture is not fully specified, and it might not include variables captured by

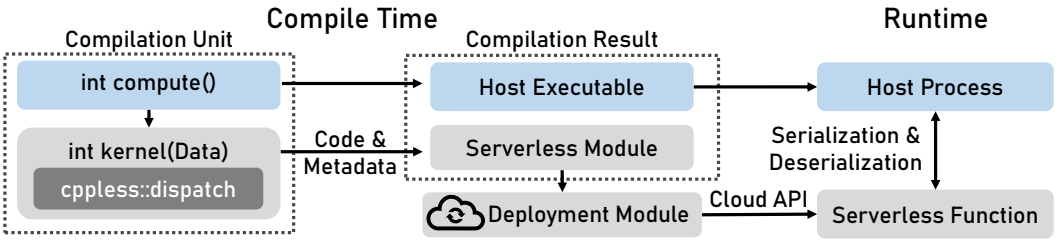

Fig. 6. *Cppless* exports selected functions as additional compilation targets, deploys them to the cloud, and provides invocation and serialization mechanisms at runtime.

reference. In practice, compilers implement reference captures by adding pointers or references to lambda's storage [19], and *Cppless* makes them explicitly available to users and libraries.

3.3 Sending Code to the Cloud

In *Cppless*, the user C++ code is separated into the single host process and potentially multiple serverless functions, which are deployed to the cloud and invoked remotely (Figure 6). First, each serializable function object marked for *FaaS-ification* is wrapped with a bridge class to define an alternative entry point. Free functions, which are not bound to a particular function object, can be serialized by wrapping them with lambda functions. The serialization of the input and output of the function is implemented in a runtime library and relies on the `cereal` library, including support for serialization of many types from the C++ Standard Template Library, such as `std::string` and `std::vector`. The user only has to manually add serialization for custom types, which is necessary as C++ objects cannot be serialized in a unique, cross-platform way.

Once the user application attempts to invoke a serverless function, the *Cppless* library will call the serialization methods for each input argument and identify the external cloud function through the type name. In the function code, the bridge class code is responsible for deserializing the input arguments and calling the original function object. The bridge class entry point interacts with the cloud provider environment, e.g., reading the arguments using the provided function API and writing function results. The return value is serialized and deserialized in the same fashion.

4 *Cppless* Implementation

The *Cppless* runtime library builds on top of the new language extensions and hides all cloud vendor-specific cloud interfaces (Section 4.1). In addition to alternative entry points and lambda reflection, we add a unique name for all lambda functions with adjusted mangling rules (Section 4.2). Internally, all language extensions are implemented directly in the LLVM and Clang compiler [20] (Section 4.3), The interactions with multiple compiler passes and cloud deployment is hidden from the user through extensions to the build system (Section 4.4). Finally, we discuss the limitations of the current *Cppless* implementation and propose solutions to overcome them (Section 4.5).

4.1 User Library with Dispatcher

Cppless implements a *fork-join* style API built on top of a low-level dispatcher interface, which is based on sending tasks in the form of serializable and identifiable function objects. In addition to *joining* on a single function result, dispatchers also offer a *wait-any* operation. Dispatchers encapsulate an interface of a single cloud provider, allowing to easily switch between different systems without requiring users to rewrite their applications; user-code can be implemented as a C++ template with dispatcher type as a template parameter. Dispatchers interact with the

```
"entry_points": [{
    "original_function_name": "mangled_C++_function_name", "filename": "dispatcher_aws_alt_0",
    "user_meta": {
      "ephemeral_storage": 512, "memory": 1024, "timeout": 10,
      "identifier": "./examples/aws/dispatcher.cpp@..."
    }
}]
```

Fig. 7. Alternative entry point metadata, as generated by the *Cppless* compiler.

```
template<typename Func>
void dispatch_function(Func && lambda) {
  // Cppless Language Extension: Lambda Reflection
  constexpr int capture_count = decltype(lambda)::capture_count();
  auto first_capture = serialize(lambda.capture<0>());

  // Unique lambda names, backed by sycl-unique-stable-name but with inline namespaces disabled
  auto func_id = __builtin_unique_stable_name(decltype(lambda));
  invoke(func_id, first_capture);
}
```

Fig. 8. Pseudocode of serverless dispatch: combining reflection, unique identification of lambda functions, and serialization library with user-supplied customizations.

compilation pipeline through the metadata system (Figure 7). The metadata list makes it possible to define configuration options on a per-function level directly in the C++ application.

We implement two methods of generating HTTP requests to trigger serverless functions, an HTTP/2-based implementation with `nghttp2` [21], and an HTTP/1.1-based implementation that uses the `Boost.Beast` library [22]. We implement two solutions as they have different trade-offs: HTTP/2 can achieve better performance when sending multiple requests simultaneously, while the Boost-based implementation increases portability by complementing the HTTP/2-only `nghttp2`.

With nghttp2, we distribute invocation requests in a round-robin fashion to a group of connections to the AWS service. Instead of invoking functions through their own and custom HTTPS addresses, connections are initialized to the AWS Lambda REST API, and we specify the function name in each REST request. With this solution, we can reuse warm connections for consecutive invocations, support many concurrent requests, and reduce the chance of head-of-line blocking issues. On the other hand, the Boost-based implementation issues a TCP-backed HTTP request for each invocation. All requests share the same instance of the Boost.Asio IO context to manage asynchronous requests. However, separate requests mean that the number of concurrent invocations is limited by the space of file descriptors available to the user process.

4.2 Lambda Functions

To differentiate between many deployed serverless functions, the compiler must create a connection between the function object code and the remote entry point. However, in contrast to regular free functions, lambda functions in C++ are unnamed. To solve this problem, we generate unique identification of all types and use these compile-time values to name alternative entry points (Figure 8). This functionality is backed by the Clang implementation of `builtin-sycl-unique-stable-name`, a feature added for to support the SYCL framework [23], which has a similar use-case. The function generates a mangled type name, but uses a slightly modified Itanium C++ ABI mangling [24] where we remove inlined namespaces from the mangling prefix. This feature is controlled by an additional

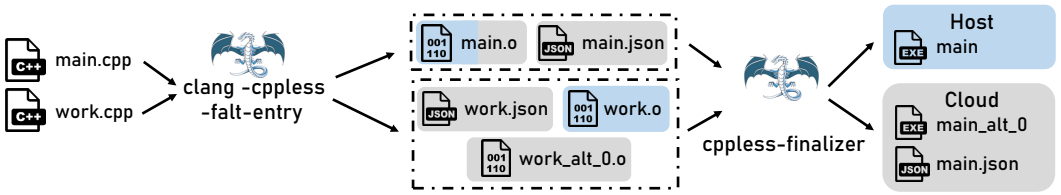

Fig. 9. Compilation code flow of a project with an alternative entry point in `work.cpp`, compiled to `work_alt_0.o`. The CMake extension hides the multiple compilation steps and deployment to the cloud.

flag and is disabled by default to not affect other mangled names. This increases compatibility when client and function code are compiled with different standard library implementations (Section 2.2).

4.3 A Serverless Compiler in Clang

The changes to the Clang compiler are limited to 963 lines of code added and 192 lines removed.

Frontend. The compiler frontend is altered to parse and validate the new annotation for alternative entry points and metadata. As alternative entry points must not be directly called, issues arise where they are not emitted into the LLVM module, especially with top-level declarations. We ensure that methods annotated as alternative entry points are treated as if they are used to prevent dead-code optimizations, and we force the template instantiation when the entry point is present. Clang's `used` annotation has a similar effect. In templated contexts, this change ensures the function is emitted once the parent context is fully instantiated.

Code Generation. We amend the code generation process in LLVM to work with alternative entry points. We propagate the names of functions exported to the cloud to the subsequent compilation steps. The metadata describing the function configuration is evaluated as a constant expression. For each alternative entry point, we convert the metadata to a `std::string` instance available to the compiled program and attach it to the corresponding LLVM function as an attribute.

LLVM Backend. During the backend code generation, we propagate information about alternative entry points through the pipeline. The corresponding LLVM function is annotated as an alternative entry point to ensure a separate treatment in the backend. Once the CodeGen module generates the main LLVM module, it is cloned for each entry point found in the translation unit. For each such module, we generate the code which results in a separate object file for each alternative entry. Additionally, a binary is created for the host application. At this point, we also create the manifest file, which stores configuration data of all entry points, including the user-supplied metadata, such as function resource configuration.

Linking. The main Clang driver handles all linker invocations for specified object files and targets. Build tools often utilize this driver due to its uniform interface, which motivates building a modified linker driver that exposes the same interface and can be used by build systems. We introduce a new tool `cppless-finalizer`, a cross-platform driver for Clang that can handle the linking step for multiple output files when alternative entry points are present. `cppless-finalizer` accepts the same command line interface as Clang, using the same Clang toolchains to support linking for different platforms. Our tool reads manifest files from the compilation which describe alternative entry points, and links them using the original Clang driver. The new driver produces one regular output file and additional files for each alternative entry point, while merging the manifest files into a single result (Figure 9).

```
# Create standard C++ compilation target
add_executable(parallel_pi parallel_pi.cpp)
# C++ target depends on our runtime
target_link_libraries(parallel_pi PRIVATE cppless::cppless)
# Enable processing of alternative entry points, and adds AWS C++ runtime to function
aws_lambda_target(parallel_pi)
# Deploys functions to the AWS cloud
aws_lambda_serverless_target(parallel_pi)
```

Fig. 10. Integrating *Cppless* into the build system requires only minor adjustments to build targets.

4.4 Compilation Pipeline

When using alternative entry points, *Cppless* produces an additional executable for each serverless function (Figure 9). Since the bridge class templates are instantiated lazily, the alternative entry points are only generated for serializable function objects if they are indirectly instantiated from non-templated contexts. We implement a custom deployment tool in addition to the C++ compiler, which is responsible for deploying the compiled function code using metadata stored in a compiler-emitted manifest file. We propose a compilation flow consisting of (1) integration with the CMake build system, (2) a deployment tool that uploads alternative entry point executables to the cloud as serverless functions. The former provides users with automatic invocation of all compilation steps, while the latter encapsulates cloud-specific interfaces.

CMake Build Integration. To generate and deploy serverless functions, users add *Cppless* to their project build system (Figure 10). We implement CMake extensions that define specific build targets with support for compilation with *Cppless*. The target adds a second compilation pass, adjusts compiler flags, and invokes cloud deployment tools when code change is detected.

Each target with alternative entry points is compiled twice. First, we conduct the compilation for the host system when alternative entry points are not emitted, and we resolve all dependencies and configuration details for the known host target. Since this compilation process can use different compilation flags and linker targets than needed for the serverless functions, we create a different CMake configuration for the remote target. Internally, separate build configurations are managed by using the `ExternalProject` functionality of CMake: we instantiate the project again, enable the *serverless* setting, and build only the selected target in the newly created project. In this new project, we change compilation settings to invoke Clang in the mode that emits alternative entry points. Furthermore, the project's nested instance can use different flags and link targets for the *serverless* mode. The user can provide a different CMake toolchain for the alternative mode to adapt code generation for the target environment of serverless functions.

These details are hidden from the user by exposing a function `aws_lambda_serverless_target` in CMake (Figure 10). The function effectively creates an additional cross-compiled target. The user must decide the cloud environment at the configuration time by selecting a specific function in CMake. The specialization to a specific FaaS platform happens at the compilation time because of differences between clouds in target architecture, environments, and deployment steps.

Cloud Deployment Script. The deployment script encapsulates the complexity of managing cloud resources and vendor-specific interfaces. For each alternative entry point, a new serverless function is created, configured, and deployed with the compiled code. Function names are provided through the unique type identifiers generated by the compiler.

Cross-Compilation. Cppless generates alternative entry points within a separate CMake project instance, enabling compilation with completely different toolchains. We demonstrate this with

a simple example of offloading two functions to the cloud, which are cross-compiled from an x86 host to the ARM instance of AWS Lambda running on Graviton CPUs. The implementation process involves extracting the ARM sysroot from official Amazon Linux AWS containers and configuring a custom CMake toolchain file specifying the ARM target triple and sysroot location. All *Cppless* dependencies are built using this cross-compilation toolchain, while the packaging process is adapted to identify dynamic dependencies without relying on the native x86 `ldd` utility.

4.5 Limitations

Serializable Function Objects. Requiring tasks to be serializable functions permits using some constant function pointers, i.e., a pointer where the exact name of the called function is known at compile time. Supplying function pointers does not fit the user-space design where the type of function objects is used as a basis for creating serverless functions. Instead, a potential solution would be to add an implicit conversion from constant function pointers to captureless lambda function objects. This would allow *Cppless* to treat function pointers as regular lambdas.

Entry Point Interface. Since each alternative entry point is treated as a clone of the *main* function, it must use the same interface and be compiled into an executable. In the example of AWS Lambda, this aligns with the design of the platform, where users deploying C++ functions are expected to deploy an executable that starts a server waiting for new requests and manually process them with the selected function. However, other platforms might require different forms of deploying code, such as shared libraries. There, *Cppless* could generate shared libraries instead.

Selective Compilation. Cppless currently treats all alternative entry points in the same way and generates full code for each one of them. To decrease the size of alternative entry points, *Cppless* could incorporate techniques from established offloading frameworks that reduce the number of functions compiled, e.g., explicit annotations of offloaded functions like `__device__` in CUDA or `#pragma omp declare target` in OpenMP. Additionally, functions can also be determined automatically based on use in the explicitly offloaded code, as is the case for SYCL and OpenMP.

Compilation Time. As multiple LLVM modules are produced and then lowered to the target language independently, the compilation time of a target increases: each target with alternative points is passed to the compiler twice, and the code generation time of the target increases linearly with the number of alternative entry points. This limitation could be fixed by integrating the cloning process deeper with LLVM, which currently does not support such integration. Thus, we limit ourselves to using the public LLVM APIs to improve the maintainability of *Cppless*.

Captured Variables. Cppless supports passing arguments only by value, with modified data returned explicitly as the function result. This design addresses potential complications in serverless environments: functions typically execute asynchronously and treating reference-captured variables as value-captured prevents issues with dangling references and memory synchronization.

In *Cppless*, the values of variables captured within C++ lambda expressions are transmitted as the function invocation payload. This means we exclude non-local (global) variables since they are not included in the lambda's capture list. If access to global variables becomes necessary, this limitation could potentially be addressed through a dedicated compiler analysis phase that examines the Abstract Syntax Tree to identify uncaptured variables referenced within the lambda body.

Zero-Copy and Serialization-Free Data Transfer. Cppless requires all data to be serialized for network transfer due to the inherent constraints of serverless function cloud interfaces. Consequently, the framework makes no assumptions about data layout compatibility between the client's system and the function environment. This problem is delegated to the serialization library to ensure

portability. Binary compatibility issues can arise in solutions that implement remote direct memory access and serialization-free data transfer [6, 25]. In such scenarios, the user bears responsibility for ensuring identical memory layouts between the client and function environments, which is a standard requirement for RDMA-accelerated and serialization-free data movement [26].

5 Evaluation

We demonstrate the ease of programming and parallel offloading in *Cppless* with micro-benchmarks, thorough performance analysis using the Monte-Carlo approximation of a PI as a basis of discussion, and case studies using two applications: N-Queens from the Barcelona OpenMP Task Suite (BOTS) [27] and a CPU Ray-Tracing application [28]. We used a `m5.4xlarge` virtual machine instance in the AWS *eu-central-1* region, with 64 GiB RAM, network bandwidth of up to 10 Gb/s, and costing $0.92 per hour. The machine runs Ubuntu 22.04 and Intel Xeon Platinum 8175M CPU, with a frequency of 2.50GHz and 16 vCPUs, where each vCPU unit corresponds to a logical CPU core. The virtual machine costs $0.92 per hour. We compile all benchmarks with the *Cppless* compiler based on the development branch of Clang 15. When compiling, we use the CMake's *Release* mode, which enables the `O3` optimization level in Clang. For all benchmarks except serialization, we repeat measurements 21 times and reject the first warmup run to exclude the effects of initialization, cold startups, and connection setup.

In the evaluation, we answer the following questions:

- How do the two underlying components of *Cppless* perform - serialization and invocation dispatch (Section 5.1)?
- How does our performance compare against multi-threaded computations on the virtual machine (Section 5.2-5.4)?
- How do single-source functions in *Cppless* perform against a manual implementation using cloud provider SDK (Section 5.2.2)?
- Can C++ benefit from fine-grained functions with independent configuration (Section 5.2.3)?
- Are cold startups a major performance issue in C++ functions (Section 5.2.4)?
- How large are the generated function binaries and deployment packages (Section 5.5)?

5.1 Microbenchmarks

We use microbenchmarks to analyze the performance of two critical parts of *Cppless* runtime: data serialization with the `cereal` library and invoking functions with the internal HTTP client. The choice of serialization format directly impacts the performance of serverless offloading, and the function dispatch must scale up to thousands of invocations to allow for parallel processing.

Serialization. Serverless systems put constraints on the type of data that can be transmitted to a function. In the case of AWS Lambda, the integration with API gateway restricts the types. In practice, functions can be invoked with a valid JSON object or Base64-encoded binary data. We evaluate two scenarios of serializing C++ objects with the `cereal` library: the default one using a direct JSON serialization, and an alternative one that performs a binary serialization followed by the Base64 encoding. As a baseline, we use the plain binary serialization that can be used in systems using RPC and RDMA requests [6]. On all benchmarks, we exclude the cost of memory allocation for the resulting payload (serialization) and data objects (deserialization). To create a realistic experiment environment, we exclude the influence of CPU caches and measure the performance of these operations when reading data from main memory. To that end, we use the Intel intrinsic `_mm_clflush` to flush and invalidate cache lines storing serialization objects. We repeat measurements 1,001 times and reject the first warmup execution.

		Time [ms]	Throughput [GiB/s]
Binary	Encode	0.75	4.961
	Decode	0.79	4.743
Binary Base64	Encode	3.4	1.096
	Decode	13	0.287
JSON	Encode	74.61	0.05
	Decode	75.88	0.049

(a) Serialization of an array of 32-bit integers.

		Time [ms]	Throughput [GiB/s]
Binary	Encode	21.65	1.076
	Decode	18.69	1.246
Binary Base64	Encode	34.89	0.667
	Decode	46.51	0.501
JSON	Encode	278.45	0.084
	Decode	289.07	0.081

(b) Serialization of an array of structures.

Fig. 11. Benchmarking the performance of data serialization in *Cppless* with `Cereal` library. REST-based serverless platforms can only accept payload as a JSON or base64-encoded binary.

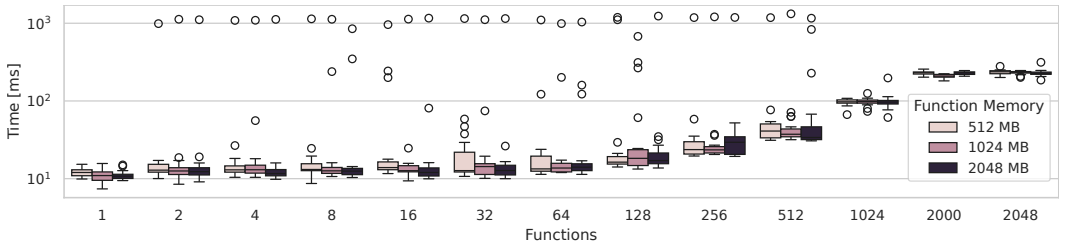

Fig. 12. The total latency of concurrent invocations of warm AWS Lambda functions with the nghttp2 dispatcher, as observed by the client. Boxes represent the data between the first and third quartiles, with whiskers adding 1.5 of the interquartile range.

First, we benchmark the serialization of an array of one million 32-bit unsigned integers (Figure 11a), with a total binary size of 4 MB (3.81 MiB) which is close to the maximum input payload of 6 MB on AWS Lambda. Adding the Base64 encoding decreases the serialization throughput by 4.52x. However, the binary option still significantly improves performance against a plain JSON serialization that has to convert numbers into their string representation. Then, we evaluate the serialization of a structure consisting of two integers and a single string with 42 characters, with a custom serialization method provided to the library (Figure 11b). This serialization is more expensive due to pointer jumping and more complex encoding of a `std::string`. Nevertheless, our custom binary serialization format is up to 8x faster than a vanilla JSON serialization.

AWS Lambda Client. We examined the latency of parallel invocations to AWS Lambda functions when using our HTTP2 dispatcher in *Cppless*. Our dispatcher implementation uses Boost.Asio abstractions to dispatch dozens of parallel invocations without allocating a separate thread for each task. We use the default configuration with 16 HTTP/2 connections and invoke a no-op C++ function deployed with our compiler. We use an AWS cloud account with a maximum number of concurrent function invocations of 2,000.

Figure 12 shows that the median latency of a single invocation is 12 and 11 milliseconds, depending on the function memory size. All invocations are warm, and the median function execution time varies between 700 and 925 microseconds. We do not observe a significant influence of the function memory size on parallel invocations. Since the dispatcher can reuse the TCP connection, the initial latency of establishing an HTTPS connection to the AWS REST API is paid only in the first cold

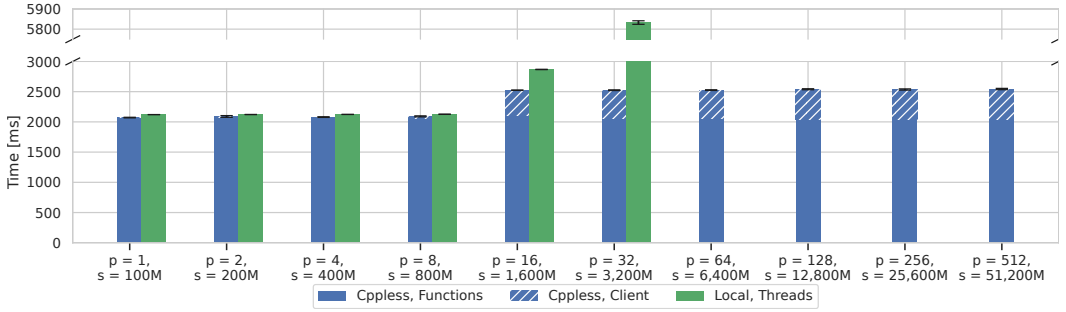

Fig. 13. Weak scaling of PI approximation on AWS Lambda and the virtual machine. Bars represent the mean with 95% confidence intervals.

invocation. After the connection initialization, the client dispatches invocations until it reaches the AWS Lambda concurrency limit or exhausts client dispatcher resources.

Slowdown. The total client runtime increases at larger scales, from ca. 14 milliseconds at 64 invocations up to 230 milliseconds at 2000 invocations. To verify if this slowdown is caused by the serverless system or resource limitations of the client, we replicated the virtual machine 8 times to run one-eighth of the original workload from each machine, obtaining in total the same number of concurrent invocations. We implemented an MPI-based dispatcher with a barrier to synchronize distributed clients, and measure the longest runtime across all ranks. At 512 invocations (64 per MPI rank), the median runtime decreased from 34-41 ms to 28.5-38 ms, indicating that runtime is slightly influenced by the limitations of the dispatcher or virtual machine resources. However, at 1024 and 2000 invocations, the runtime was still up to 6.4x and 7.7x slower when executed by eight VMs concurrently instead of one, thus indicating scaling limitations on the AWS Lambda system.

Boost.Beast. We also compared against the second dispatch implementation using Boost.Beast and independent HTTP requests. We increased the system limit on opened file descriptors to support 2,048 concurrent invocations. There, the lowest observed median latency was 13.45 milliseconds for a single function invocation. However, while the client latency stays relatively constant for the `nghttp2` dispatcher until 256 and 512 invocations, it quickly increases for the `Beast` dispatcher, reaching 135 milliseconds on 64 invocations and over four seconds on 2,000 concurrent invocations. Thus, the dispatcher provided with *Cppless* can offer an optimized implementation of remote functions without relying on the user to select the best-performing library and configuration.

5.2 Case Study: Monte Carlo Pi Benchmark

To evaluate the scalability and elasticity of serverless computing with *Cppless*, we conducted four experiments with a perfectly parallelizable application: approximating the Pi number with Monte-Carlo simulation, as introduced in Section 3. Unless specified otherwise, benchmarks are compiled with the addition of the `fast-math` flag and functions use 2048 megabytes of memory.

5.2.1 Weak Scaling. First, we examine the weak scalability by increasing the number of parallel functions up to 512 (Figure 13). We compare the performance against computation on the virtual machine, using manual dispatch of results to threads. Since the virtual machine is limited to the local 16 virtual CPUs, it cannot handle increasing workloads. Instead, users can extend computing resources by dynamically dispatching tasks with *Cppless*, incurring only a small, constant overhead on the HTTP client. The computing cost stays relatively constant: it changes by less than 3% between problem sizes. At 512 functions, the total cost is 500.4x larger than using only one function.

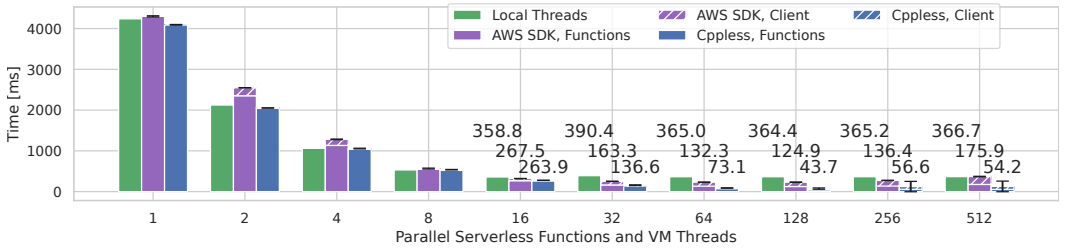

Fig. 14. Strong scaling of PI approximation on AWS Lambda and the virtual machine, with 200 million samples. Bars represent the mean with 95% confidence intervals.

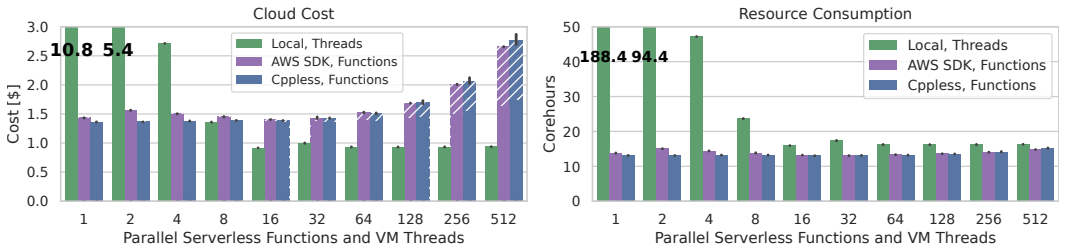

Fig. 15. Cost of 10,000 repetitions of PI benchmark (Figure 14); computing cost (solid) and invocation fee (dashed), mean with 95% confidence interval.

5.2.2 Strong Scaling. We examine the efficiency by scaling the Pi computation with 200 million samples. We compare against two baselines: multithreaded scaling on the virtual machine, and a manual serverless implementation using the official AWS SDK. By comparing against the latter, we can verify that *Cppless* can extract and deploy C++ functions without introducing overheads.

Performance. Dispatching computations with *Cppless* scales efficiently until 128 functions, where the median runtime of a function decreases to 42.7 milliseconds. There, the overheads of function scheduling and network protocols impact the further scalability (Figure 14). The manual baseline scales worse, which can be explained by the underlying implementation that uses a fixed number of HTTP connections and creates a new thread for each asynchronous invocation.[4] We experimented by increasing the fixed number of TCP connections used by the SDK, but it did not improve the performance and significantly decreased stability.

Cost. To compare the costs, we make the optimistic assumption that the virtual machine is charged only for the time spent on the Monte-Carlo computation. In serverless, the cost is split into two components: function cost for memory and CPU time, and a flat fee for each invocation. The results show that offloading with *Cppless* enables scaling beyond VM limits with a slow increase in cost (Figure 15). The increase is caused by the invocation fee charged for each parallel function, which impacts computations with very short and fine-grained functions; at 512 workers, the median duration of a single function is just 28.4 milliseconds. At full utilization, local computations are cheaper because the bare price of a vCPU on a virtual machine is lower than that of Lambda.[5]

However, functions consume fewer core hours than a VM computation. Serverless can perform better when the cloud provider supplies a better CPU. Furthermore, we notice that the variance of serverless cost increases with the number of workers, while resource consumption is stable. The

[4] The issue has been confirmed by the AWS team: https://github.com/aws/aws-sdk-cpp/issues/2991; accessed on 24.06.2024.

[5] In the `m5.4xlarge` virtual machine, a single vCPU costs $0.0575 per hour. AWS Lambda has an equivalent of 1 vCPU at 1,769 MB of memory, which would cost $0.104 per hour - 1.8x more.

(a) Function runtime speedup.

(b) Function cost increase.

Fig. 16. Performance of the PI benchmark with a varying resource allocation for the serverless function. Speedup and costs are normalized to results of 128 MB function.

Function Memory	C++	Python 3.8		Python 3.9		Python 3.10		Python 3.11		Python 3.12	
		Bare	NumPy	Bare	NumPy	Bare	NumPy	Bare	NumPy	Bare	NumPy
128	10.4	123.12	607.6	74.06	498.08	73.31	311.9	73.52	490.79	81.1	526.84
512	10.36	122.5	621.08	74.4	489.97	71.43	311.76	74.29	498.9	78.73	527.48
1024	10.82	124.61	614.76	73.21	496.04	73.31	310.33	72.53	495.11	79	533.87
2048	11.78	124.34	594.1	72.58	471.84	72	297.59	73.94	473.15	79.24	513.24

Fig. 17. The sandbox initialization time on cold startup (milliseconds) at different memory configurations (megabytes). We compare the Cppless Pi function against Python functions that return the input payload.

larger variance is explained by an increased frequency of cold starts (up to 2.5% of all invocations across repetitions). At 512 functions, a cold invocation takes 9.8 ms, which is 7% longer than a warm one, but adds ca. 13 ms of initialization time. Since AWS Lambda charges for the initialization time, a cold invocation at this scale costs up to 2.4x more than a warm one. When spurious cold starts occur, the total cost is up to 18% higher, but these only happen in 3.5% of all repetitions.

5.2.3 Memory Scalability. Serverless frameworks must choose between deploying many distinct Lambda functions or fusing them together. The latter simplifies dynamic code dispatch in systems like Lithops and Crucial and decreases the frequency of cold starts, as each generic function instance can handle all tasks. However, having only a single cloud function type decreases flexibility, as resource allocation for the function has to be decided upfront.

Cppless deploys each offloaded C++ function as a separate Lambda function, allowing to configure memory and CPU resources independently for each task. To verify the gains of this policy, we scaled the memory allocation of the Pi function (Figure 16). In this compute-intensive task, scaling up to a full logical core provides faster computation for the same price. On the other hand, prior works show that other serverless functions achieve the best efficiency at much lower memory allocations [8]. With a single cloud function type, users would waste resources on memory-bound functions or lose the opportunity to accelerate compute-intensive workloads for free. Thus, deploying many function types in *Cppless* allows efficiently mixing different operations types in a single application.

5.2.4 Cold Startups. Serverless is designed for irregular and elastic workloads, and launching many parallel functions leads to cold startups, as each function instance needs its own container. Since *Cppless* deploys functions separately for higher resource efficiency (Section 5.2.3), it has to tolerate more frequent cold invocations. Thus, we examine the performance effects of a cold startup on C++ functions. We compare against functions implemented in Python, the most popular language in serverless computing (Section 2.1). We implemented a simple Python function that returns the payload and its cold status. However, pure Python cannot provide performance that would make it competitive in this Monte-Carlo simulation, requiring libraries such as NumPy. Thus, we use a second variant of this function that only adds a single NumPy import.

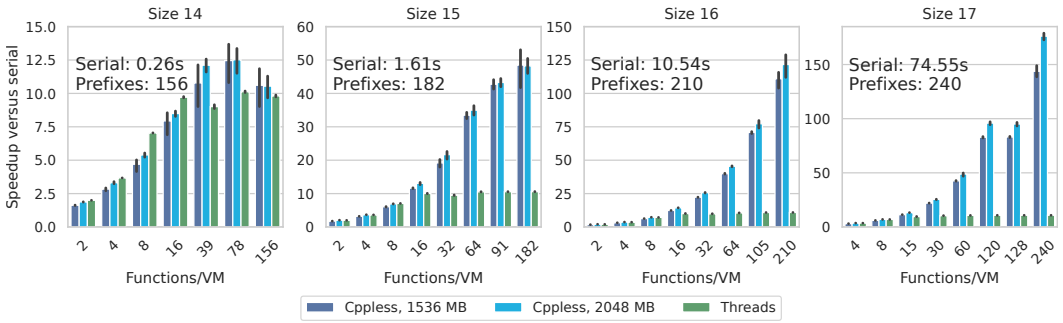

Fig. 18. N-Queens benchmark with varying board size. We measure the entire processing time, including prefix computation. Mean speedup against the serial runtime, with 95% confidence intervals.

We triggered cold initialization by changing the function's environment configuration, and gathered the cost of initializing the function sandbox as reported by the cloud provider (Figure 17). Thanks to the compilation and a lack of interpreter sessions, C++ functions can be initialized very quickly, with only 10-12 milliseconds of added overhead. While the cost of initializing a bare Python function can be up to 11.8x higher than in C++, adding a single popular library needed for numerical computations increases the overhead to 25.26 - 58.4x. Thus, *Cppless* helps users to easily integrate serverless functions into C++ applications, where the runtime overhead of using many fine-grained and specialized functions is minimal.

5.2.5 Code Changes. In *Cppless*, users need to add two lines of configuration and modify 7 lines of code to dispatch tasks and wait for the results. When implementing the same benchmark with the official cloud SDK, users need to manually serialize data and implement callbacks for asynchronous invocations. There, developers need at least 4 lines of code for configuration, 16 lines for the dispatch loop, and 11 lines for the callback handling results. On top of that, users need to separately build, package, and deploy the function and later hardcode the function name in the main application.

5.3 Case Study: OpenMP Benchmark

Next, we evaluate *Cppless* with the N-Queens problem from the BOTS suite. In this benchmark, we find a placement of N queen figures on a chessboard of size NxN, such that no two queens threaten each other. The problem is known to be NP-hard and has numerous solutions dependent on the size of the board N. We changed the array-based implementation from the BOTS benchmark suite with one that uses bit patterns to represent board states [29], improving the performance of determining queen placement. We parallelize the solution by pre-computing *prefix* tasks of length p where the location of the first p queens is fixed, allowing us to decompose the primary problem into smaller tasks [30]. The local parallel implementation requires 35 lines of code, while a solution with the serverless *Cppless* dispatcher is implemented with 47 lines of code. No additional serialization is involved, as prefixes are sent using the `std::vector` class.

Performance. We compare the local computation with the dispatch to serverless when using a prefix of length 2 (Figure 18). At the largest scale, each function or thread receives a single prefix to resolve. Offloading computations to serverless functions provides speedups of up to 111x and 122x for $N = 16$, and up to 144x and 176x for $N = 17$. However, the results indicate that serverless offloading does not achieve linear scaling. This can be explained by the variance in the workload assigned to different tasks, with the total execution time limited by the longest-running task. On a large scale, the slowest task takes 2.38-5.3x longer than the fastest one ($N = 16, 17$) and even up

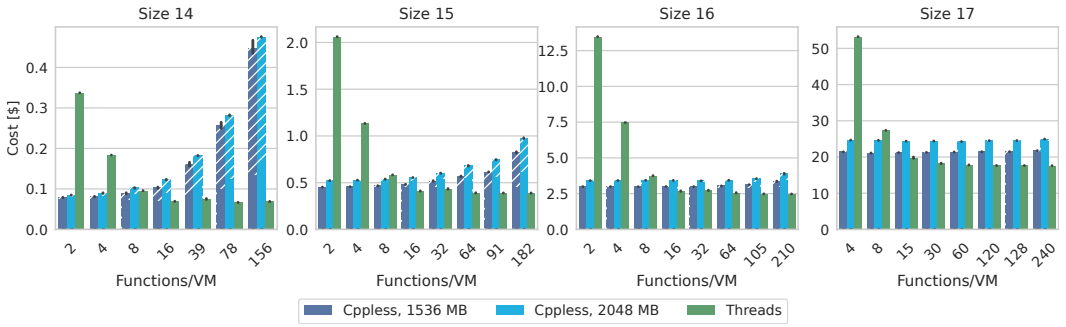

Fig. 19. Cost of 10,000 repetitions of N-Queens benchmark (Figure 18); mean with 95% confidence interval. *Cppless* includes computing cost (solid), invocation fee (dashed), and dispatch cost of host vCPU (black top bar), which does not exceed 0.2% of total cost.

to 151x longer on smaller problem sizes. Furthermore, at smaller problem sizes and larger scales, dispatching tasks can take up to 14% of the total time. Nonetheless, the users benefit from the pay-as-you-go billing mode in such a heterogeneous workload: parallel functions do not wait for the results of longer-running functions and thus do not accrue charges.

Cost. We compare the total computation cost of local and serverless parallelization. As before, the cost of serverless functions is split into two components: the function cost obtained by querying the cloud billing data and the invocation fee. We add a third cost component to obtain a fair comparison against local multithreading: host time, which includes the additional time needed for data preparation and task serialization. We define it as a time from the beginning until all functions are dispatched, and we multiply it by the cost of using a single vCPU of the virtual machine.

As seen previously, a fully utilized virtual machine delivers lower cost due to an increased compute price at AWS Lambda (Figure 19). More importantly, we notice that the total cost at the two largest scales increases only by up to 14% and 2% for $N = 16, 17$, respectively. Thus, as in the previous benchmark, users can obtain significant speedup at a minor cost increase.

Knapsack and Floorplan We also tested *Cppless* with two other benchmarks from the BOTS suite. However, these rely on recursive task creation and use shared memory to synchronize results between tasks to prune early branches. Since available serverless systems support neither distributed memory nor inter-function communication, these benchmarks cannot scale efficiently there. Thus, we focus instead on workloads that fit better this programming model.

5.4 Case Study: Ray-Tracing

As a second application, we consider a Monte-Carlo implementation of ray tracing [28] with a bounding volume hierarchy mechanism [31]. The implementation consists of nine translation units with 1,616 lines of C/C++ code, and the offloading includes dynamic lists of objects, custom and polymorphic types, and shared pointers.

The benchmark scenario renders a random scene divided into smaller tiles to create parallel tasks. In local multi-threading, worker threads can access image tiles directly in the memory. For *Cppless*, we compute the bounding volume hierarchy locally and the serverless function is invoked with the image tile and volume hierarchy. For each benchmark, we measure the entire ray-tracing process, including splitting images into tiles and serialization, and exclude the time needed for the random generation of the initial image.

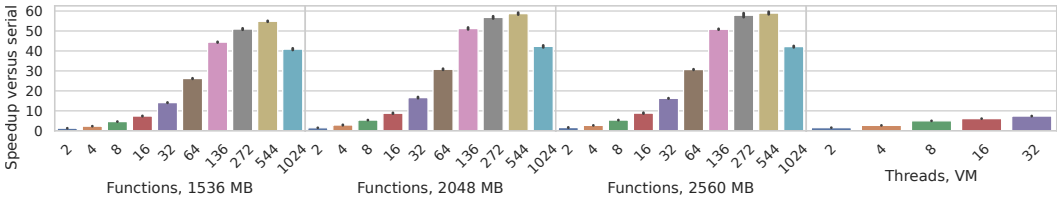

Fig. 20. CPU-Raytracer rendering of a 600x600 image. We measure the entire runtime, including image partitioning. Mean speedup of 20 repetitions against the serial runtime with one thread of 59.159 seconds.

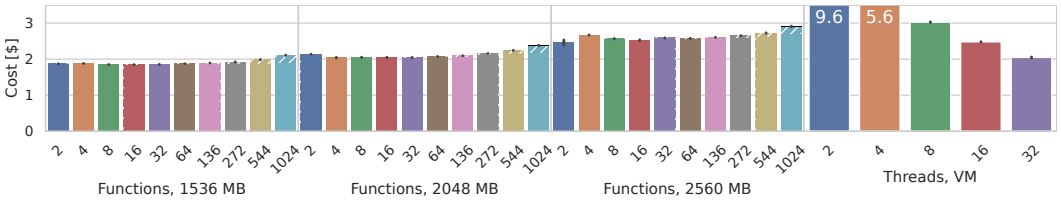

Fig. 21. The total cost of 1,000 invocations of the CPU-Raytracer benchmark from Figure 20. For serverless, we include the computation cost (solid), invocation fee (dashed), and cost of host preparation (black top bar) which does not exceed 0.8% of total cost.

Performance. We evaluate the benchmark on an image of size 600x600, varying the tile dimensions to control the number of dispatched functions. The image tile was changed from 600x300 for two functions to 19x19 for 1024 functions. *Cppless* provides a speedup of up to 59x, while the limited resources of a non-elastic virtual machine saturate quickly (Figure 20). This result is already attractive for the end user, as their compute-intensive task taking almost 60 seconds can now be completed in slightly more than one second without allocating a dedicated and more powerful virtual machine with multiple cores.

However, the obtained speedup is lower than the number of invoked functions. First, we must create the bounding volume hierarchy, a serial process on the host that adds, on average, 4.2 milliseconds to the runtime and increases payload size. However, the main culprit of lower scalability is the unequal work distribution. This uneven workload distribution is caused by the unequal division of the image to selected tile sizes and the varying per-pixel workloads, as the computation time of each task varies and depends on the objects present in the assigned tile. For example, on 136 tiles, the longest-running task can take 27.8x more time than the fastest. There, the contribution of function duration to the total computation time can vary between 3.7% and 94.8%. Finally, the speedup decreases when the number of functions increases from 544 to 1024. This can be explained by the time needed to serialize and dispatch functions, which grows from 0.52 to 0.99 seconds, while the maximum function duration decreases from 402 to 228 ms (2560 MB memory).

Cost. With *Cppless* functions, we significantly decrease processing time with only a minor increase in the total cost (Figure 21). Particularly in the case of functions configured with 1536 MB of memory, 544 functions decrease computing time by up to 54.8x. However, the total cost increases from 2 to 544 functions by only 6%. Thanks to our embedding of elastic cloud functions into C++ applications, users effectively achieve faster processing almost for free.

Code Modifications. To understand the difference between *Cppless* and OpenMP, we compared our interface with a ray tracing application based on OpenMP offloading [32, 33]. We replace parallel directives with an explicit dispatch interface and move the loop body into a C++ lambda

Benchmark	Function			Dynamic Dependencies	Function Data	Zip Package	
	Object Files	Static Links	Final Entrypoint			Compressed	Uncompressed
NQueens	920 kB	15.08 MB	5.89 MB	0	0	8.33 MB	19.8 MB
Ray Tracing	4.3 MB	″	6.45 MB	0	0	8.44 MB	20.19 MB
SeBS Thumbnailer	594 kB	″	5.77 MB	49.37 MB	0	25.5 MB	69.18 MB
SeBS Image Recognition	642 kB	17.33 MB	5.8 MB	479.16 MB	97.96 MB	215.1 MB	595.38 MB
SeBS Graph Pagerank	588 kB	24.5 MB	6.68 MB	0	0	20.32 MB	52.73 MB

Fig. 22. Binary size of serverless functions: object files are linked with static dependencies to produce the final function. Functions are deployed as a zip package, together with dynamically linked libraries and additional function data. The image recognition cannot be deployed as a package on AWS Lambda due to its size.

function. However, the main difference lies in data transmission that uses serialization instead of `target map` directives. Rather than specifying OpenMP `declare mapper` or using plain arrays, we implement twelve serialization routines to support custom datatypes. Of these, eleven simply enumerate serialized class fields. As a benefit, we can transmit C++ vectors and shared pointers without additional modifications, since all variables captured by the lambda are implicitly serialized.

5.5 Code Size

Since *Cppless* does not apply dedicated solutions for selective compilation (Section 4.5), we verified if the generated binary size adds significant overhead to deployment. For each application, we analyze the sizes of generated alternative entry points and final deployment packages uploaded to the cloud. In addition to the NQueens benchmark and Ray Tracing application, we used C++ ports of three serverless functions from the SeBS benchmark suite [8]: image thumbnailer with OpenCV 4.5; image recognition with ResNet-50, OpenCV, Torch 1.1, and Torchvision 0.1; and graph processing with igraph 0.10. We adapt each function for offloading with *Cppless* and generated deployment packages. Figure 22 shows that dynamic libraries dominate the size of serverless deployment. These include libc and its dependencies, which the AWS Lambda packaging script includes by default for compatibility. Alternative entry points are statically linked with Lambda runtime, libcurl, and *Cppless* dependencies like nghttp2. In this case, pruning unnecessary symbols used only on the host side can help reduce the size of the final entry point. However, deploying practical serverless applications is likely to be dominated by complex dependencies like OpenCV and Torch. In particular, the image recognition benchmark can only be deployed as a Docker container due to the large size of the Torch library.

6 Related Work

Prior approaches focus on solutions that exploit dynamic and interpreted languages (Section 6.1), solutions that extend the OpenMP programming model to support HPC clusters and clouds (Section 6.2), and offload C++ functions to accelerator devices (Section 6.3). *Cppless* takes a different approach to support distributed deployment in the language and does not rely on additional runtimes like MPI or Spark, and general-purpose cloud offloading (Section 6.4).

6.1 Interpreted Languages

Many prior systems focused on interpreted and high-level languages, which implicitly support serializing data objects, allow for dynamic code extraction, and provide reflection support for runtime dispatch. Kovachev et al. [34] proposed offloading compute-intensive Android services to a Java server to speed up computations and reduce energy usage. There, Java code is shipped at the runtime to a persistent server in the cloud and cached for reuse in future executions. Lithops [35] is a multi-cloud framework for offloading Python functions to serverless functions. It implements

a replacement of the standard `multiprocessing` library, which provides the same interface but invokes remote cloud functions instead of spawning a new local OS process. The function code is not deployed to the cloud ahead of time; Lithops analyzes it at runtime to detect all dependent modules, serializes them, and sends them to the cloud storage. The code is later fetched and executed by a generic serverless Python worker. Python functions can use modules and libraries embedded in the default container image [36]. To exploit additional libraries, users must manually build and deploy extended container images before using the software.

Crucial implements stateful serverless applications in Java [37]. Functions invocations are represented as remote threads that implement the standard `Runnable` interface, providing an abstraction layer similar to the standard interface of Java threads. Functions are invoked with the payload specifying the name of the class implementing the `Runnable` interface and its parameters. The serverless executor can create an instance of the class thanks to the availability of reflection in Java, which is missing in C++. Crucial deploys a single Lambda function per application [38], preventing fine-grained memory and CPU resource configuration across different Java functions. Crucial implements a custom storage system presented to users as a distributed shared object layer, which allows them to manage function state and receive data from offloaded tasks since the `Runnable` interface does not support the direct return of results. Thus, it cannot be deployed to vanilla serverless platforms. Serverless Shell [39] builds on top of Crucial's distributed shared object layers, executing shell scripts remotely on serverless functions. However, the code intended for remote execution must be explicitly invoked through a dedicated executor.

Kappa [40] targets long-running serverless applications and implements automatic checkpointing for Python functions. A dedicated compiler transforms user code by inserting continuations, but its support is limited to native Python code and does not handle Python C extensions. Containerless [41] compiles a subset of JavaScript into Rust, providing speculative and opportunistic acceleration of functions implemented in high-level languages.

6.2 OpenMP

OpenMP was initially designed for multithreaded computations in shared-memory systems. Multiple attempts have been made to extend OpenMP with offloading computations to a distributed and remote system, and these can be classified as *cluster* and *cloud* OpenMP. Later iterations of the standard added support for offloading computations to a *target* device, with a particular focus on accelerators (Section 6.3).

Cluster OpenMP. In this mode, data and computations are distributed among nodes of an HPC cluster. These attempts were often limited by the overhead of sharing memory across computing nodes. Sato et al. proposed Omni OpenMP [42], where the compiler inserts communication routines into the code to guarantee the correctness and consistency of shared data. In 2006, Intel offered Cluster OpenMP that used memory page protection to detect cross-node memory accesses and enforce synchronization. However, initial evaluation by Terboven et al. demonstrated scalability issues, with basic OpenMP primitives slower by up to four orders of magnitude [43]. Cluster OpenMP has been discontinued in 2010 [44]. XcalableMP (XMP) is a programming model for clusters that uses compiler directives similar to OpenMP, with data represented as distributed arrays in the Partitioned Global Address Space (PGAS) model [45]. XcalableACC [46] extended later XMP with OpenACC directives to support accelerator programming, focusing on clusters with HPC interconnects like InfiniBand and Tightly Coupled Accelerators (TCA). This solution relies on a custom Omni compiler that translates XMP directives into functions to a runtime library.

Several works attempted to combine OpenMP and distributed computations with MPI by exploiting OpenMP 4.0 offload. Jacob et al. [47] was the first to use OpenMP offload to target remote

CPUs. Instead of defining a global address space, cluster nodes are treated as a collection of disjoint shared-memory address spaces. The implementation is based on LLVM, with kernel code included in a fat binary with host code. The actual runtime is built on top of MPI, with the host communicating with worker nodes to distribute computation tasks. Recently, Yviquel et al. proposed OpenMP Cluster (OMPC) [48], a tasking model for HPC clusters. They implement a new device plugin for LLVM's OpenMP, with a dedicated management module for automatic data movement and handling task dependencies across nodes. The host and target code are also compiled into a fat binary, and communication is done with MPI. Keftakis et al. [49] implemented offloading where each cluster node becomes a separate OpenMP device. Their approach is based on the OMPi compiler and generates a single executable that contains the host code and all target kernels. Remote communication is delegated to MPI by spawning new MPI processes as copies of the host process, which avoids the problem of dispatching code at runtime.

Cluster OpenMP solutions tend to assume a homogenous system where a single copy of an application executes on different nodes. In contrast, *Cppless* has to generate multiple targets that are separately uploaded to a cloud platform. Instead of modeling external resources as devices that often need to be predefined ahead of time through a configuration file [49], we dynamically determine the number of cloud functions used at the runtime. Furthermore, many of these systems delegate communication and actual offloading to MPI since it provides a portable solution for HPC systems and can benefit from fast networks. *Cppless* provides its own dispatcher implementation without relying on MPI, which is often unavailable in the cloud and not elastic enough for serverless offloading.

Cloud OpenMP. In OpenMR, Wottrich et al. proposed to offload parallel loops to Apache Hadoop MapReduce runtime in the cloud [50]. New OpenMP directives are designed to execute MapReduce jobs, with a restricted execution model since OpenMP synchronization directives cannot be supported there. However, this work does not implement an actual compiler for the extensions and instead conducts manual source code modifications for selected benchmarks.

OmpCloud [51, 52] offloads parallel loop into Spark by representing cloud resources as a new device type in OpenMP. Input data is sent to object storage in the cloud or an HDFS server, and parallel regions are mapped to Spark's map-reduce execution model. The target code is embedded in the fat binary with host code, and the Java Native Interface (JNI) is needed for compatibility with Scala used in Spark. Since Spark jobs operate in a distributed setting with data mapped to Resilient Distributed Datasets (RDDs), OpenMP synchronization directives like atomic operations and barriers are not supported. OmpCloud has been used to offload ray tracing [33], a case study similar to ours (Section 5.4). The operation is perfectly parallelizable since each pixel is rendered independently. However, the evaluation revealed scalability issues in Spark offloading: non-negligible overheads explained by a bottleneck in the Spark system, and multithreading within a single Spark node that is less efficient than using OpenMP.

These solutions use distributed cloud systems fundamentally different from serverless functions in *Cppless*. Spark is deployed on a dedicated cluster that cannot be quickly rescaled since allocating a new virtual machine can take up to several minutes. Even though OmpCloud can start and stop virtual machines automatically, decreasing startup overheads requires users to pay to keep the cluster ready and available. Meanwhile, serverless functions do not generate any costs when unused and can be scaled up rapidly, with commercial systems achieving up to 1,000 new execution units every 10 seconds [1], and open-source systems scaling up to up to 2,500 new functions in a second [53]. OmpCloud results show significant overhead of the Spark system that is comparable to or even larger than the cost of moving data from host to target [51], which is the main bottleneck of serverless offloading in *Cppless*. Furthermore, the low efficiency of parallelization in small

benchmarks is explained by the cost of initializing Spark context, and authors propose to employ in the future advanced Spark systems that retain the context alive between jobs [33]. This problem is similar to cold starts, and *Cppless* helps to decrease the negative performance effects of cold starts by facilitating C++ programming in serverless (Section 5.2.4).

6.3 GPU

Offload was an early approach for programming heterogeneous multicore systems, demonstrated with the Cell Broadband Engine (BE) [54]. An offload block could contain a list of variables copied to the accelerator's local context, similar to a C++ lambda function with capture by value. However, data movement of arrays was handled by explicitly marking pointers as *outer* to trigger access to a software cache or DMA. Since offloaded functions are compiled to the device only, a source-to-source translator was used to duplicate the call graph and create copies of functions used on the device. This reduces the developer's effort but creates issues in type inference, function pointers, and handling projects with multiple translation units.

In the field of GPU programming, single-source programming models have been developed to compile C++ code into accelerators [23, 55, 56]. Early attempts include C++AMP [57], which was supported initially only by the Microsoft compiler and later added to AMD's hcc compiler [58]; both implementations have since been deprecated [59]. Offload was an early approach for programming heterogeneous multicore systems [54], which extended variable declarations with new to handle data movement and employed call graph duplication to create copies of functions offloaded to the device. Single-source programming of GPUs requires dedicated compilers and compilation and code generation are simplified at the cost of forcing users to change their source code. While CUDA requires users to annotate functions compiled into device code and handle data copies, OpenMP offloading to GPUs automatically determines functions compiled to the device and data handling with implicit and explicit data mapping [60]. Later, the support for OpenMP offloading to GPUs has been added to LLVM [60]. Different device codes are embedded in a single fat binary to avoid recompilation when switching devices.

SYCL [61] is a single-source programming model for GPUs, where programmers use dedicated data accessors to their code and provide explicit kernel naming [62]. However, functions compiled to the device are determined automatically without explicit annotations. SYCL has been adopted by many solutions, including Codeplay's ComputeCpp [63], Intel oneAPI Data Parallel C++ (DPC++) [64], and the AdaptiveCpp [65]. The latter is the only single-pass compiler, which stores the kernel code in an intermediate representation and lowers it later to the target ISA in a JIT fashion. This improves portability at the cost of a slight decrease in performance on some devices.

Cppless takes a different approach as we avoid embedding offloaded code in a single binary and instead create multiple deployments at compilation time. We do not require portability because the target architecture of serverless functions must be known at the deployment time. We apply code duplication at the LLVM module level. Even though *Cppless* does not target GPU devices, it could incorporate similar methods to determine functions used in the serverless target to reduce the size of generated binaries.

Remote Devices Kasmeridis et al. [66] extended the OpenMP cluster solution [49] with remote offloading. A new module virtualizes the remote device by forwarding all data and kernels at runtime. This solution is similar to API remoting that has been previously implemented through hijacking API calls [67] or compiler-assisted translation of API calls [68].

Patel et. [69] proposed offloading OpenMP to remote CPU and GPU targets. The code is compiled into a single binary, and they adapted the LLVM/OpenMP plugin API to explicitly copy the device's binary code to the remote server. Users are expected to preallocate execution servers and pass their configuration through an environment variable to the host application. The implementation can

serialize data with protobuf and custom serialization, and it targets clouds with gRPC and HPC clusters with UCX support. The UCX transport layer was a focus of optimizations [70] and was later replaced with an MPI communication layer [71].

In *Cppless*, the deployment is more complicated as we have to preallocate multiple functions before execution, and functions can come with many external dependencies that are linked dynamically. Furthermore, runtimes like AWS Lambda require shipping the *libc* with all dependencies, and the deployment package of even simple functions can weigh dozens of megabytes. In the future, *Cppless* can help OpenMP with offloading to serverless clouds by handling alternative entry points and implementing a virtual OpenMP device that delegates invocation to our runtime.

6.4 Others

gg [72] and llama [73] offload UNIX processes tasks to the serverless cloud, with a particular focus on offloading compilation and linking steps through integration into a build system. These solutions operate on a different granularity of the problem since an entire process is executed on an AWS Lambda function instead of a single C++ function like in *Cppless*.

Finally, the concept of an *alternate* entry point can be found in COBOL. There, it is used to start the program execution from different places in the code [74]. Thus, COBOL's alternate entry points effectively implement logically separated subprograms [75]. In *Cppless*, we use alternative entry points to expose multiple remote targets while maintaining the single source implementation and similar compilation workflow.

7 Discussion

Why a C++ compiler extension? Our primary motivation for supporting automatic offloading in C++ is to leverage the many existing codebases, particularly in scientific computing. These can be supported without requiring users to rewrite their applications in a different language. While a compiler patch requires additional effort, this burden will likely decrease once compile-time reflection is adopted in C++26. This advancement could potentially simplify our solution to a source-to-source translator for implementing alternative entry points.

Languages with built-in metaprogramming capabilities, such as Rust, can support alternative entry points without compiler modifications. We validate a proof-of-concept implementation that uses Rust's procedural macros to adapt functions for dispatching and offloading, utilizing the `syn` and `quote` crates to convert between Rust tokens and syntax tree. The implementation duplicates the offloaded function: one version (enabled in default compilation mode) replaces the function body with a simple dispatcher that loads a shared library and locates the corresponding symbol; the second version preserves the original function body and is exported as a shared library during a separate compilation. A Rust-based approach could increase the adoption of compiled languages in serverless environments while still benefiting from *Cppless* infrastructure for deployment, cloud resource management, and efficient runtime dispatch.

Does serverless need explicit data management? *Cppless* transmits all function data during dispatch and does not provide an interface for explicit memory movement, unlike the `map` and `data` directives in OpenMP or memory copy functions in CUDA. In the current serverless function model, users cannot benefit from asynchronous data movement or maintain warm data on the remote side. The payload can only be sent once with each invocation request, and consecutive invocations cannot be guaranteed to target the same remote environment. These operations would be feasible in a stateful serverless model [76, 77]. There, *Cppless*'s dispatcher module could be extended with operations to support state management and data sharing between functions.

***Cppless* versus OpenMP** The problem of offloading computations has been extensively studied in OpenMP (Section 6.2 and 6.3), and *Cppless* addresses several similar challenges. OpenMP's

offloading solution handles data passing with implicit capture and explicit mapping directives, implements user-defined serialization via mapper directives, uses target directive to annotate code for device offloading, generates unique kernel names, and compiles offloaded kernel entry points by embedding device code in fat binary. Cppless differs in its deployment approach, requiring separate binaries for each offloaded function and mandatory data serialization. However, we chose to implement a new model primarily because serverless OpenMP would face significant challenges in data locality, communication, and scalability.

First, applying the OpenMP model to serverless environments could create a false sense of compatibility and lead to poor performance since functions operate in a distributed model. OpenMP applications are implemented with the assumption of low-latency access to shared memory, which is not the case in a serverless environment. While OpenMP's offloading capabilities could restrict function address spaces to explicitly mapped data and serverless can implement disjoint address spaces [47], performance would still be compromised. A single OpenMP device can share address space among dozens of CPU cores or hundreds of GPU cores, whereas AWS Lambda scales to only six virtual CPUs per function. Second, the OpenMP standard includes features incompatible with serverless computing, such as synchronization directives. Implementing these would require suitable cloud storage solutions [9, 37], increasing both cost and complexity. Finally, *Cppless* can target applications beyond the scientific workloads, such as microservices [78], where OpenMP is not commonly used.

7.1 Future Work

We demonstrate *Cppless* by offloading to AWS Lambda. Other platforms can be supported even if they lack C++ support, for example, by cross-compiling into WebAssembly deployed with a Node.js shim. We validated this approach with Clang for Google Cloud Functions, and leave the integration of the toolchain in *Cppless* for future work. *Cppless* could generate leaner deployments by implementing selective compilation. There, we can benefit from approaches taken by OpenMP. Another potential enhancement would be supporting variables captured by reference, using explicit annotation to prevent accidental modifications and issues with dangling references (Section 4.5).

Finally, fault tolerance is currently the full responsibility of the user. *Cppless* could be extended to catch exceptions thrown inside functions, return failures with appropriate error information, and throw a custom exception on the host side. When available, the runtime could return the exception's type and the explanation provided by the `what` function of `std::exception`.

8 Conclusions

We present *Cppless*, a single-source approach to program serverless C++ functions. *Cppless* overcomes the inherent constraints of C++, and offers developers a streamlined way to transparently offload tasks to FaaS platforms. We introduce compiler extensions and meta-programming techniques to transition the runtime code deployment to compilation time. With a selection of parallel benchmarks, we demonstrate that serverless functions can be effortlessly and efficiently integrated into C++ applications. In *Cppless*, users employ parallel functions to elastically extend computing resources and accelerate their applications with negligible cost overheads.

Acknowledgments

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 program (grant agreement PSAP, No. 101002047). This work was partially supported by the ETH Future Computing Laboratory (EFCL), financed by a donation from Huawei Technologies. We thank Amazon Web Services for support through AWS Cloud Credit for Research program, and Swiss National Supercomputing Centre (CSCS) for access to compute resources.

References

[1] 2022. AWS Lambda Scaling Behavior. https://docs.aws.amazon.com/lambda/latest/dg/scaling-behavior.html. Accessed: 2024-06-30.

[2] Jiawei Jiang, Shaoduo Gan, Yue Liu, Fanlin Wang, Gustavo Alonso, Ana Klimovic, Ankit Singla, Wentao Wu, and Ce Zhang. 2021. Towards Demystifying Serverless Machine Learning Training. In *ACM SIGMOD International Conference on Management of Data (SIGMOD 2021)*.

[3] Lixiang Ao, Liz Izhikevich, Geoffrey M. Voelker, and George Porter. 2018. Sprocket: A Serverless Video Processing Framework. In *Proceedings of the ACM Symposium on Cloud Computing* (Carlsbad, CA, USA) *(SoCC '18)*. Association for Computing Machinery, New York, NY, USA, 263–274. https://doi.org/10.1145/3267809.3267815

[4] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. 2017. Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 363–376. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/fouladi

[5] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. 2019. Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 193–206. https://www.usenix.org/conference/nsdi19/presentation/pu

[6] Marcin Copik, Konstantin Taranov, Alexandru Calotoiu, and Torsten Hoefler. 2023. rFaaS: Enabling High Performance Serverless with RDMA and Leases. In *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS) (IPDPS '23)*. 897–907. https://doi.org/10.1109/IPDPS54959.2023.00094

[7] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. 2018. SAND: Towards High-Performance Serverless Computing. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference* (Boston, MA, USA) *(USENIX ATC '18)*. USENIX Association, USA, 923–935.

[8] Marcin Copik, Grzegorz Kwasniewski, Maciej Besta, Michal Podstawski, and Torsten Hoefler. 2021. SeBS: A Serverless Benchmark Suite for Function-as-a-Service Computing. In *Proceedings of the 22nd International Middleware Conference (Middleware '21)*. Association for Computing Machinery. https://doi.org/10.1145/3464298.3476133

[9] Marcin Copik, Alexandru Calotoiu, Pengyu Zhou, Konstantin Taranov, and Torsten Hoefler. 2024. FaaSKeeper: Learning from Building Serverless Services with ZooKeeper as an Example. In *Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing (HPDC '24)*.

[10] 2022. The State of Serverless. https://www.datadoghq.com/state-of-serverless/. Accessed: 2023-08-16.

[11] Simon Eismann, Joel Scheuner, Erwin van Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina L. Abad, and Alexandru Iosup. 2022. The State of Serverless Applications: Collection, Characterization, and Community Consensus. *IEEE Transactions on Software Engineering* 48, 10 (2022), 4152–4166. https://doi.org/10.1109/TSE.2021.3113940

[12] TIOBE. 2024. TIOBE Index. https://www.tiobe.com/tiobe-index/. Accessed: 2024-05-26.

[13] Marcin Copik, Marcin Chrapek, Larissa Schmid, Alexandru Calotoiu, and Torsten Hoefler. 2024. Software Resource Disaggregation for HPC with Serverless Computing. In *Proceedings of the 38th IEEE Interational Parallel and Distributed Processing Symposium (IPDPS '24)*.

[14] Jacek Kuśnierz, Vincenzo E. Padulano, Maciej Malawski, Kamil Burkiewicz, Enric Tejedor Saavedra, Pedro Alonso-Jordá, Michael Pitt, and Valentina Avati. 2022. A Serverless Engine for High Energy Physics Distributed Analysis. In *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. 575–584. https://doi.org/10.1109/CCGrid54584.2022.00067

[15] Jinfeng Wen, Zhenpeng Chen, Yi Liu, Yiling Lou, Yun Ma, Gang Huang, Xin Jin, and Xuanzhe Liu. 2021. An empirical study on challenges of application development in serverless computing. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 416–428. https://doi.org/10.1145/3468264.3468558

[16] Lukas Möller. 2022. Cppless: A single-source programming model for high-performance serverless. https://spcl.inf.ethz.ch/Publications/index.php?pub=542. Bachelor Thesis.

[17] W. Shane Grant and Randolph Voorhies. 2017. cereal - A C++11 library for serialization. http://uscilab.github.io/cereal/.

[18] Wyatt Childers, Peter Dimov, Dan Katz, Barry Revzin, Andrew Sutton, Faisal Vali, and Daveed Vandevoorde. 2024. Reflection for C++26 (P2996R4). https://isocpp.org/files/papers/P2996R4.html. Accessed: 2024-06-30.

[19] Barry Revzin, Andrei Alexandrescu, David Olsen, Daveed Vandevoorde, Barry Revzin, and Michael Garland. 2024. Introspection of Closure Types (P3273R0). https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2024/p3273r0.html. Accessed: 2024-06-30.

[20] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *CGO*. San Jose, CA, USA, 75–88.

[21] Tatsuhiro Tsujikawa. 2013. nghttp2 - HTTP/2 C Library. https://github.com/nghttp2/nghttp2.
[22] Vinnie Falco. 2016. Beast: C++ HTTP and WebSocket built on Boost.Asio. https://github.com/boostorg/beast.
[23] Ronan Keryell, Ruyman Reyes, and Lee Howes. 2015. Khronos SYCL for OpenCL: A Tutorial. In *Proceedings of the 3rd International Workshop on OpenCL* (Palo Alto, California) *(IWOCL '15)*. Association for Computing Machinery, New York, NY, USA, Article 24, 1 pages. https://doi.org/10.1145/2791321.2791345
[24] 2024. Itanium C++ ABI. https://itanium-cxx-abi.github.io/cxx-abi/abi.html. Accessed: 2024-06-22.
[25] Fangming Lu, Xingda Wei, Zhuobin Huang, Rong Chen, Minyu Wu, and Haibo Chen. 2024. Serialization/Deserialization-free State Transfer in Serverless Workflows. In *Proceedings of the Nineteenth European Conference on Computer Systems* (Athens, Greece) *(EuroSys '24)*. Association for Computing Machinery, New York, NY, USA, 132–147. https://doi.org/10.1145/3627703.3629568
[26] Konstantin Taranov, Rodrigo Bruno, Gustavo Alonso, and Torsten Hoefler. 2021. Naos: Serialization-free RDMA networking in Java. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 1–14. https://www.usenix.org/conference/atc21/presentation/taranov
[27] Alejandro Duran, Xavier Teruel, Roger Ferrer, Xavier Martorell, and Eduard Ayguade. 2009. Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp. In *2009 international conference on parallel processing*. IEEE, 124–131.
[28] Peter Shirley. 2020. *Ray Tracing in One Weekend*. https://raytracing.github.io/books/RayTracingInOneWeekend.html
[29] Martin Richards. 1997. *Backtracking algorithms in MCPL using bit patterns and recursion*. Technical Report. Citeseer.
[30] Kenji Kise, Takahiro Katagiri, Hiroki Honda, and Toshitsugu Yuba. 2004. Solving the 24-queens Problem using MPI on a PC Cluster. *Graduate School of Information Systems, The University of Electro-Communications, Tech. Rep* (2004).
[31] Peter Shirley. 2020. *Ray Tracing: The Next Week*. https://raytracing.github.io/books/RayTracingTheNextWeek.html
[32] Mortatti Matheus and Yviquel Hervé. 2023. Ray-Tracer. https://github.com/matheusmortatti/Ray-Tracer. Accessed: 2025-03-24.
[33] Matheus Mortatti, Herve Yviquel, and Guido Araujo. 2018. Automatic Ray-Tracer Cloud Offloading in OPENMP. In *2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE, Lyon, France, 428–435. https://doi.org/10.1109/CAHPC.2018.8645871
[34] Dejan Kovachev and Ralf Klamma. 2012. Framework for Computation Offloading in Mobile Cloud Computing. *International Journal of Interactive Multimedia and Artificial Intelligence* 1, 7 (2012), 6. https://doi.org/10.9781/ijimai.2012.171
[35] Josep Sampe, Pedro Garcia-Lopez, Marc Sanchez-Artigas, Gil Vernik, Pol Roca-Llaberia, and Aitor Arjona. 2021. Toward Multicloud Access Transparency in Serverless Computing. *IEEE Software* 38, 1 (2021), 68–74. https://doi.org/10.1109/MS.2020.3029994
[36] Lithops. 2024. Lithops runtime for AWS Lambda. https://github.com/lithops-cloud/lithops/tree/8c5f6aba063fbad850ecc0fb6e9f19882be9b63d/runtime/aws_lambda. Accessed: 2024-05-26.
[37] Daniel Barcelona-Pons, Marc Sánchez-Artigas, Gerard París, Pierre Sutra, and Pedro García-López. 2019. On the FaaS Track: Building Stateful Distributed Applications with Serverless Architectures. In *Proceedings of the 20th International Middleware Conference* (Davis, CA, USA) *(Middleware '19)*. Association for Computing Machinery, New York, NY, USA, 41–54. https://doi.org/10.1145/3361525.3361535
[38] 2022. Crucial Project: Crucial Examples. https://github.com/crucial-project/examples/. Accessed: 2024-06-11.
[39] Aurèle Mahéo, Pierre Sutra, and Tristan Tarrant. 2021. The serverless shell. In *Proceedings of the 22nd International Middleware Conference: Industrial Track* (Québec city, Canada) *(Middleware '21)*. Association for Computing Machinery, New York, NY, USA, 9–15. https://doi.org/10.1145/3491084.3491426
[40] Wen Zhang, Vivian Fang, Aurojit Panda, and Scott Shenker. 2020. Kappa: A Programming Framework for Serverless Computing. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (Virtual Event, USA) *(SoCC '20)*. Association for Computing Machinery, New York, NY, USA, 328–343. https://doi.org/10.1145/3419111.3421277
[41] Emily Herbert and Arjun Guha. 2020. A Language-based Serverless Function Accelerator. arXiv:1911.02178 [cs.DC]
[42] Mitsuhisa Sato, Shigehisa Satoh, Kazuhiro Kusano, and Yoshio Tanaka. 1999. Design of OpenMP Compiler for an SMP Cluster. In *Proc. of the 1st European Workshop on OpenMP*. 32–39.
[43] Christian Terboven, Dieter an Mey, Dirk Schmidl, and Marcus Wagner. 2008. First Experiences with Intel Cluster OpenMP. In *OpenMP in a New Era of Parallelism*, Rudolf Eigenmann and Bronis R. de Supinski (Eds.). Springer, Berlin, Heidelberg, 48–59. https://doi.org/10.1007/978-3-540-79561-2_5
[44] Jay Hoeflinger (Intel). 2010. Cluster OpenMP for Intel Compilers Discontinued. https://web.archive.org/web/20181116055322/http://software.intel.com/en-us/articles/cluster-openmp-for-intel-compilers/. Accessed: 2025-03-22.
[45] Masahiro Nakao, Jinpil Lee, Taisuke Boku, and Mitsuhisa Sato. 2012. Productivity and Performance of Global-View Programming with XcalableMP PGAS Language. In *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (Ccgrid 2012)*. 402–409. https://doi.org/10.1109/CCGrid.2012.118

[46] Masahiro Nakao, Hitoshi Murai, Takenori Shimosaka, Akihiro Tabuchi, Toshihiro Hanawa, Yuetsu Kodama, Taisuke Boku, and Mitsuhisa Sato. 2014. XcalableACC: Extension of XcalableMP PGAS Language Using OpenACC for Accelerator Clusters. In *2014 First Workshop on Accelerator Programming Using Directives*. IEEE, New Orleans, LA, USA, 27–36. https://doi.org/10.1109/WACCPD.2014.6

[47] Arpith C. Jacob, Ravi Nair, Alexandre E. Eichenberger, Samuel F. Antao, Carlo Bertolli, Tong Chen, Zehra Sura, Kevin O'Brien, and Michael Wong. 2015. Exploiting Fine- and Coarse-Grained Parallelism Using a Directive Based Approach. In *OpenMP: Heterogenous Execution and Data Movements*, Christian Terboven, Bronis R. De Supinski, Pablo Reble, Barbara M. Chapman, and Matthias S. Müller (Eds.). Vol. 9342. Springer International Publishing, Cham, 30–41. https://doi.org/10.1007/978-3-319-24595-9_3

[48] Hervé Yviquel, Marcio Pereira, Emílio Francesquini, Guilherme Valarini, Gustavo Leite, Pedro Rosso, Rodrigo Ceccato, Carla Cusihualpa, Vitoria Dias, Sandro Rigo, Alan Souza, and Guido Araujo. 2022. The OpenMP Cluster Programming Model. In *Workshop Proceedings of the 51st International Conference on Parallel Processing*. ACM, Bordeaux France, 1–11. https://doi.org/10.1145/3547276.3548444

[49] Ilias Keftakis and Vassilios V. Dimakopoulos. 2022. Experiences with Task-Based Programming Using Cluster Nodes as OpenMP Devices. https://doi.org/10.48550/arXiv.2205.10656 arXiv:2205.10656 [cs]

[50] Rodolfo Wottrich, Rodolfo Azevedo, and Guido Araújo. 2014. Cloud-Based OpenMP Parallelization Using a MapReduce Runtime. In *2014 IEEE 26th International Symposium on Computer Architecture and High Performance Computing*. 334–341. https://doi.org/10.1109/SBAC-PAD.2014.46

[51] Herve Yviquel and Guido Araujo. 2017. The Cloud as an OpenMP Offloading Device. In *2017 46th International Conference on Parallel Processing (ICPP)*. IEEE, Bristol, United Kingdom, 352–361. https://doi.org/10.1109/ICPP.2017.44

[52] Hervé Yviquel, Lauro Cruz, and Guido Araujo. 2018. Cluster Programming Using the OpenMP Accelerator Model. *ACM Transactions on Architecture and Code Optimization* 15, 3 (Sept. 2018), 1–23. https://doi.org/10.1145/3226112

[53] Lazar Cvetković, François Costa, Mihajlo Djokic, Michal Friedman, and Ana Klimovic. 2024. Dirigent: Lightweight Serverless Orchestration. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles* (Austin, TX, USA) *(SOSP '24)*. Association for Computing Machinery, New York, NY, USA, 369–384. https://doi.org/10.1145/3694715.3695966

[54] Pete Cooper, Uwe Dolinsky, Alastair F. Donaldson, Andrew Richards, Colin Riley, and George Russell. 2010. Offload – Automating Code Migration to Heterogeneous Multicore Systems. In *High Performance Embedded Architectures and Compilers*, David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Yale N. Patt, Pierfrancesco Foglia, Evelyn Duesterwald, Paolo Faraboschi, and Xavier Martorell (Eds.). Vol. 5952. Springer Berlin Heidelberg, Berlin, Heidelberg, 337–352. https://doi.org/10.1007/978-3-642-11515-8_25

[55] Thomas Heller, Hartmut Kaiser, Patrick Diehl, Dietmar Fey, and Marc Alexander Schweitzer. 2016. Closing the Performance Gap with Modern C++. In *Lecture Notes in Computer Science*. Springer International Publishing, 18–31. https://doi.org/10.1007/978-3-319-46079-6_2

[56] Jingyue Wu, Artem Belevich, Eli Bendersky, Mark Heffernan, Chris Leary, Jacques Pienaar, Bjarke Roune, Rob Springer, Xuetian Weng, and Robert Hundt. 2016. Gpucc: An Open-Source GPGPU Compiler. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization* (Barcelona, Spain) *(CGO '16)*. Association for Computing Machinery, New York, NY, USA, 105–116. https://doi.org/10.1145/2854038.2854041

[57] Kate Gregory and Ade Miller. 2012. *C++ AMP*. Vol. 10. Microsoft Press.

[58] AMD. 2020. HCC: An open source C++ compiler for heterogeneous devices. https://github.com/ROCm/hcc. Accessed: 2025-03-22.

[59] Microsoft. 2023. C++ AMP (C++ Accelerated Massive Parallelism). https://learn.microsoft.com/en-us/cpp/parallel/amp/cpp-amp-cpp-accelerated-massive-parallelism?view=msvc-170. Accessed: 2025-03-22.

[60] Samuel F. Antao, Alexey Bataev, Arpith C. Jacob, Gheorghe-Teodor Bercea, Alexandre E. Eichenberger, Georgios Rokos, Matt Martineau, Tian Jin, Guray Ozen, Zehra Sura, Tong Chen, Hyojin Sung, Carlo Bertolli, and Kevin O'Brien. 2016. Offloading Support for OpenMP in Clang and LLVM. In *2016 Third Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*. IEEE, Salt Lake City, UT, USA, 1–11. https://doi.org/10.1109/LLVM-HPC.2016.006

[61] Ruyman Reyes and Victor Lomüller. 2016. SYCL: Single-source C++ accelerator programming. In *Parallel Computing: On the Road to Exascale*. IOS Press, 673–682.

[62] Marcin Copik and Hartmut Kaiser. 2017. Using SYCL as an Implementation Framework for HPX.Compute. In *Proceedings of the 5th International Workshop on OpenCL* (Toronto, Canada) *(IWOCL 2017)*. Association for Computing Machinery, New York, NY, USA, Article 30, 7 pages. https://doi.org/10.1145/3078155.3078187

[63] Codeplay. 2023. ComputeCpp SDK. https://github.com/codeplaysoftware/computecpp-sdk. Accessed: 2025-03-22.

[64] Intel. 2024. Data Parallel C++: the oneAPI Implementation of SYCL. https://www.intel.com/content/www/us/en/developer/tools/oneapi/data-parallel-c-plus-plus.html. Accessed: 2025-03-22.

[65] Aksel Alpay and Vincent Heuveline. 2023. One Pass to Bind Them: The First Single-Pass SYCL Compiler with Unified Code Representation Across Backends. In *International Workshop on OpenCL*. ACM, Cambridge United Kingdom, 1–12. https://doi.org/10.1145/3585341.3585351

[66] Ilias K. Kasmeridis, Spyros Mantelos, Apostolos Piperis, and Vassilios V. Dimakopoulos. 2024. Transparent Remote OpenMP Offloading Based on MPI. In *Euro-Par 2023: Parallel Processing Workshops*, Demetris Zeinalipour, Dora Blanco Heras, George Pallis, Herodotos Herodotou, Demetris Trihinas, Daniel Balouek, Patrick Diehl, Terry Cojean, Karl Fürlinger, Maja Hanne Kirkeby, Matteo Nardelli, and Pierangelo Di Sanzo (Eds.). Vol. 14352. Springer Nature Switzerland, Cham, 237–241. https://doi.org/10.1007/978-3-031-48803-0_24

[67] José Duato, Antonio J. Peña, Federico Silla, Rafael Mayo, and Enrique S. Quintana-Ortí. 2010. rCUDA: Reducing the number of GPU-based accelerators in high performance clusters. In *2010 International Conference on High Performance Computing & Simulation*. 224–231. https://doi.org/10.1109/HPCS.2010.5547126

[68] Minoru Oikawa, Atsushi Kawai, Kentaro Nomura, Kenji Yasuoka, Kazuyuki Yoshikawa, and Tetsu Narumi. 2012. DS-CUDA: A Middleware to Use Many GPUs in the Cloud Environment. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*. 1207–1214. https://doi.org/10.1109/SC.Companion.2012.146

[69] Atmn Patel and Johannes Doerfert. 2022. Remote OpenMP Offloading. In *High Performance Computing*, Ana-Lucia Varbanescu, Abhinav Bhatele, Piotr Luszczek, and Baboulin Marc (Eds.). Vol. 13289. Springer International Publishing, Cham, 315–333. https://doi.org/10.1007/978-3-031-07312-0_16

[70] Wenbin Lu, Baodi Shan, Eric Raut, Jie Meng, Mauricio Araya-Polo, Johannes Doerfert, Abid M. Malik, and Barbara Chapman. 2022. Towards Efficient Remote OpenMP Offloading. In *OpenMP in a Modern World: From Multi-Device Support to Meta Programming: 18th International Workshop on OpenMP, IWOMP 2022, Chattanooga, TN, USA, September 27–30, 2022, Proceedings* (Chattanooga, TN, USA). Springer-Verlag, Berlin, Heidelberg, 17–31. https://doi.org/10.1007/978-3-031-15922-0_2

[71] Baodi Shan, Mauricio Araya-Polo, Abid M. Malik, and Barbara Chapman. 2023. MPI-based Remote OpenMP Offloading: A More Efficient and Easy-to-use Implementation. In *Proceedings of the 14th International Workshop on Programming Models and Applications for Multicores and Manycores*. ACM, Montreal QC Canada, 50–59. https://doi.org/10.1145/3582514.3582519

[72] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. 2019. From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 475–488. https://www.usenix.org/conference/atc19/presentation/fouladi

[73] 2021. llama – A CLI for outsourcing computation to AWS Lambda. https://github.com/nelhage/llama. Accessed: 2025-03-22.

[74] 2024. Enterprise COBOL for z/OS 6.4. Programming Guide. https://www.ibm.com/docs/en/SS6SG3_6.4.0/pdf/pgmvs.pdf. Accessed: 2024-06-11.

[75] 2024. Enterprise COBOL for z/OS 6.4. Language Reference. https://www.ibm.com/support/pages/enterprise-cobol-zos-documentation-library. Accessed: 2024-06-11.

[76] Simon Shillaker and Peter Pietzuch. 2020. Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 419–433. https://www.usenix.org/conference/atc20/presentation/shillaker

[77] Marcin Copik, Alexandru Calotoiu, Gyorgy Rethy, Roman Bö hringer, Rodrigo Bruno, and Torsten Hoefler. 2024. Process-as-a-Service: Unifying Elastic and Stateful Clouds with Serverless Processes. In *Proceedings of the 2024 ACM Symposium on Cloud Computing* (Redmond, WA, USA) *(SoCC '24)*. Association for Computing Machinery, New York, NY, USA, 223–242. https://doi.org/10.1145/3698038.3698567

[78] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, and et al. 2019. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 3–18. https://doi.org/10.1145/3297858.3304013