Cppless: Productive and Performant Serverless Programming in C++

Marcin Copik
ETH Zurich
Switzerland

Marcin Chrapek
ETH Zurich
Switzerland

Larissa Schmid
ETH Zurich
Switzerland

Alexandru Calotoiu
ETH Zurich
Switzerland

Torsten Hoefler
ETH Zurich
Switzerland

ABSTRACT

The rise of serverless introduced a new class of scalable, elastic and highly 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 found 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 from a C++ application to serverless workers can provide up to 30x speedup, requiring only minor code modifications and costing less than one cent per computation.

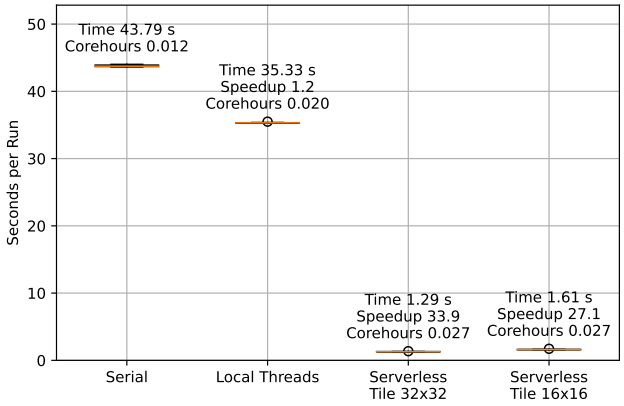

Figure 1: CPU-Raytracer used on rendering a 500x500 image (Sec. 5.3). Parallelization with *Cppless* provides speedups unattainable on a local machine, with minor code modifications.

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 3000 new invocations in a minute on commercial cloud platforms [1]. Thus, even though 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 [3, 4, 10, 13, 17, 22–24].

In the pay-as-you-go system of FaaS, users are charged for each millisecond of active computation in a function. Optimizing an application deployed to virtual machines improves the responsiveness and throughput of service. Still, these improvements might not immediately lead to decreased costs when rescaling the deployment is not feasible. In serverless, each optimization provides immediate benefits as every millisecond saved decreases the cost of running an application in the cloud, and increases the workload size where serverless is more efficient than a persistent deployment [7, 9]. Thus, it is important that such services are implemented on efficient backends and enjoy the performance advantages of fast and compiled languages.

However, even though serverless has been shifting to larger and more compute-intensive workloads, it is still dominated by languages popular in web services, such as Node.js and Python [2]. Achieving high performance in these high-level and dynamic languages is difficult, and requires programmers to use native extensions. 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.

The low adoption of C++ in serverless can be explained by the complex deployment model, where functions are split from the main application and compiled separately, deployed to the cloud using interfaces that are not standardized and differ for each cloud platform, and executed through a vendor-specific API (Sec. 2.1). While managed languages can use the bytecode and runtime introspection mechanisms to automatically extract function code from an application (Sec. 2.2), 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 into and from loosely typed JSON format (Sec. 2.3). This results in an unnecessarily 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++. *Cppless* accelerates parallel C++ applications by shifting compute-intensive tasks to serverless functions, which are automatically created, deployed, and invoked by the framework. In the example of a Ray-Tracer application (Fig. 1), using serverless with *Cppless* requires only 13 lines of code more than for an implementation with local threads and provides 33.9x speedup against serial computation. Each serverless processing costs less than 0.003$ and is delivered in less than two seconds, while a virtual machine could take several minutes to boot, and keeping online even a small virtual machine with two virtual CPUs costs 0.048$ per hour.

Cppless achieves these performance, cost, and productivity results by combining the serverless and non-serverless program parts in a single source code. The compiler detects serverless function code and creates **alternative entry points** to allow for a separate compilation path for select functions (Sec. 3). Input data is packed into a binary format with the help of a serialization library, and the framework encapsulates the vendor-specific process of deploying and invoking functions. The compiler is built on top of the LLVM framework and requires only a few modifications to the `clang` codebase (Sec. 4). The new language extensions added to support single-source programming are hidden from the user behind a high-level library interface. Thus, 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 in C++ (Sec. 5), we demonstrate that *Cppless* enables efficient offloading parallel computations 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 parallel computations in C++ to the cloud.
- A C++ toolchain that allows users for straightforward embedding of serverless functions into their applications.
- A C++ standard compliant framework providing high-level abstractions that hide the complexity of cloud provider APIs.

2 BACKGROUND

Serverless functions provide high scalability but introduce a division between functions shipped to the cloud and the main application code (Sec. 2.1). While several frameworks have been developed to integrate functions and applications using them, they targeted high-level and dynamic languages (Sec. 2.2). The unique challenges of the statically typed and compiled C++ require a different approach (Sec. 2.3).

2.1 Serverless Functions

Serverless functions are expanded and shrank automatically by the cloud provider, according to the number of invocations arriving at the REST API of a FaaS system (Fig. 2) [9, 18]. New instances of functions are deployed in isolated sandboxes such as containers and micro virtual machines, and the user is not concerned with adjusting the scale of an application. All function invocations must arrive at the REST API of the serverless cloud, and input data must be serialized and sent in the loosely typed JSON format.

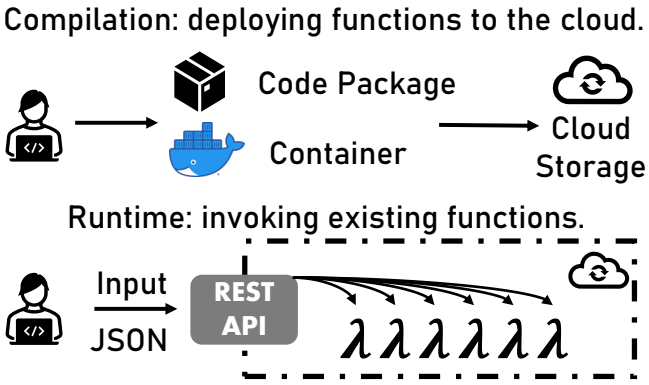

Figure 2: Development workflow in serverless: functions are shipped to the cloud during the build time.

Such scaling is only possible if users define their functions earlier, and provide the cloud operator with code in a compatible format. The standard deployment method is a compressed code package with the function code, which requires locating its dependencies and gives users very little control over the runtime environment. An alternative technique ships to the cloud entire Docker images, leaving users with more freedom to control the software stack of a function [6]. In practice, cloud operators distribute the container images used for workers handling packaged code, as the choice of the image impacts the build of native libraries. While the process of deploying functions is quite straightforward for languages with ubiquitous package managers and managed runtimes, it is more complex for applications that compile to native code and require building many dependencies.

2.2 Related Work

Lithops [27] 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 instead of spawning a new process, a function is invoked. Instead of deploying function code ahead of time, Lithops analyzes the function code at runtime to detect all dependent modules, serializes them, and sends it to the cloud storage. The code is later fetched and executed by a generic serverless Python worker. Crucial implements stateful serverless applications in Java [5]. Function invocations are abstracted as threads that execute the `Runnable` interface, similarly to the standard interface for built-in Java Threads. Similarly to Lithops, a generic serverless function is used, which receives a marshalled Java object with code attached and parameters attached. Crucial comes with a distributed shared object layer to manage state and return data from the offloaded task. Thus, it cannot be deployed to vanilla serverless platforms.

Kappa [32] 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 it does not handle Python C extensions. Containerless [16] compiles a subset of JavaScript into Rust, providing speculative and opportunistic acceleration of functions implemented in high-level languages. In the field of GPU programming, single-source programming models have been developed to compile C++ code into GPUs [15, 19, 25, 31]. These require using dedicated compilers, which simplified the code generation with techniques such as annotating functions compiled to device code, using dedicated data accessors, and explicit kernel naming [8].

2.3 Why *Cppless*?

Cppless attempts to achieve in C++ the same goal as Lithops and Crucial did for Python and Java, respectively; namely, to provide a standard-compliant interface for single-source serverless programming. However, it needs new techniques and approaches due to the lack of platform-independent byte code that can be inspected and serialized at runtime. Integrating serverless functions into C++ applications is unnecessarily 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 Bash allow to extract function code and send it with the input data, 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 program must be restructured to allow for separate compilation of the serverless functions, extended with serialization and invocation interface, and the compilation artifacts must be uploaded to the cloud before the compilation is finished.

Challenge #2: Server Environment Serverless functions still execute on a *server*, which can be easily hidden in the high-level and interpreted languages with their own runtime. 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 sandbox on an x86 server, with `libstdc++` and `musl` available as implementations of the C++ and C standard libraries, respectively. Thus, it is not sufficient to decompose the function code into a separate shared library but still compile the entire application in a single environment with the same configuration. Statically typed and compiled 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 and presenting a uniform interface.

Challenge #3: Static Typing Cloud platforms accept function input through a RESTful interface, in a JSON format, which requires a dedicated conversion from strongly typed C++ data structures. A regular function or an OpenMP task benefit from the compile-time verification of type conversion and compatibility. On the other type, separated implementations of the invoking application and *serverless* function are joined only by an intermediate proxy in the form of a loosely typed JSON, which prevents the compiler from analyzing type compatibility. Incorrect data serialization and typing are caught only when it corrupts the execution of a function unless users manually write a JSON schema or add convoluted error-handling code.

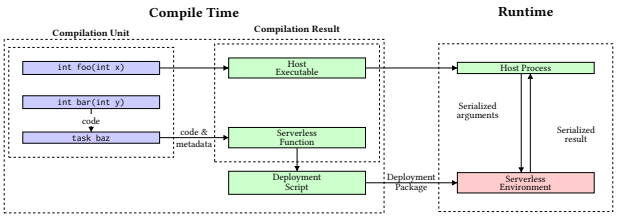

Figure 3: *Cppless* exports selected functions as remote serverless functions, and deploys them to the cloud.

3 COMPILER FOR SERVERLESS

Cppless implements a separation of user C++ code into the host process and serverless functions, which are deployed to the cloud and invoked remotely (Fig. 3). The design comprises three main parts: a set of language extensions implemented at the compiler level, a user library that encapsulates language extensions to expose a single-source programming model to users, and runtime tools responsible for deploying and invoking cloud functions.

3.1 Alternative Entry Points

Cppless exports 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 initially started. From the entry point, the control flow can diverge and is governed by the programming language's semantics. In languages from the C-family, this function is usually called `main`, which is automatically called when the program initially starts[1]. We extend this concept with *alternative entry points* by allowing multiple entry points to co-exist in a single program. Each new entry point can be seen as a separate program, as the alternative entry functions are compiled into separate executables or libraries. Entry points allow programmers to define multiple separate and distinct programs in the same source file, while transparently forming borders between them with functions that define the start of execution flow for each entry point.

Alternative entry points are combined with template metaprogramming techniques to implement some of the new additions in a library, rather than modify the compiler.

3.2 Serverless Function Definition

In *Cppless*, serverless functions are defined as function objects, including lambda functions which define their own type as well.

[1]Usually through a `_start` function which is the actual entry point

```
template <class Func> struct ProcessBridge {
  void operator()() {
    // Spawn process executing the alternative
    // entry point.
    auto alt_entry_name = gen_id<Func>();
    spawn(alt_entry_name);
  }
  __attribute((entry)) int main(int, char**) {
    Func func;
    func();
  }
};
```

Figure 4: `ProcessBridge` connects a user-defined function object with an invocation of a remote serverless function.

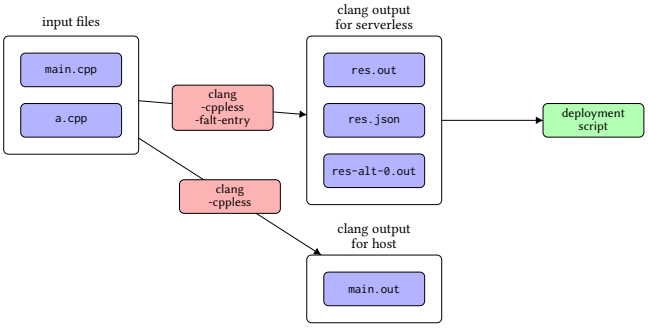

Figure 5: Compilation code flow of a project with an alternative entry point

The limitation of a pure library approach is that the alternative entry point only gains access to the type of the function object, as the value it is not known at the compile time. The value of the function object, which contains the actual code, thus has to be transferred to the entry point at the runtime through serialization, which prevents deployment ahead of time. To that end, *Cppless* employs a compiler extension that allows for generic serialization of lambda functions, which allows end users to implement serverless functions this way, as long as all variables captured in the function are serializable.

Functions 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 (Listing 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. To differentiate between many deployed serverless functions, the compiler must create a connection between the function object code and the remote entry point. To solve this problem, we implement unique identification of all types and use these compile-time values to name alternative entry points.

3.3 Compilation Pipeline

Using proposed language extensions, we can split the user code into separate programs that can be deployed to the cloud. First, each serializable function object 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 input and output of the function is implemented in a runtime 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.

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. The bridge class entry point is also responsible for interacting with the cloud provider environment, e.g., reading the arguments using the provided function API and writing function results. When using alternative entry points, *Cppless* produces an additional executable for each serverless function (Figure 5). 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. 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 return value is serialized and deserialized in the same fashion.

The compilation workflow can be extended to support cross-compilation. To that end, two separate compilation passes are configured: one host compilation pass which ignores alternative entry points, and one focused on the main entry points for the serverless function.

Compilation Pipeline. 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.

CMake Build Integration. 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, creates *a sysroot*, and invokes cloud deployment tools. Serverless functions are redeployed only if a code change is detected.

Cloud Deployment Script. The deployment script encapsulated 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.

4 IMPLEMENTATION

Cppless implements a high-level interface for dispatching serverless invocations in C++ applications (Sec. 4.2). Internally, the library depends on a set of language extensions that are implemented directly in the LLVM [21] (Sec. 4.2), including reflection suport for C++ lambda functions (Sec. 4.3). Finally, we discuss the limitations of the current *Cppless* implementation and proposed solutions to overcome them 4.4.

```cpp
double pi_estimate(int n);

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

  cppless::aws_dispatcher dispatcher;
  auto aws = dispatcher.create_instance();

  using config = lambda::config<
    cppless::lambda::with_memory<512>,
    cppless::lambda::with_ephemeral_storage<64>
  >;

  std::vector<double> r(np);
  auto fn = [=] { return pi_estimate(n / np); };
  for (auto& result : r)
    cppless::dispatch<config>(aws, fn, result);
  cppless::wait(aws, np);

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

Figure 6: Offloading parallel PI computation to AWS Lambda with *Cppless*.

```cmake
add_executable(parallel_pi parallel_pi.cpp)
target_link_libraries(
    parallel_pi
    PRIVATE cppless::cppless
)
aws_lambda_target(parallel_pi)
aws_lambda_serverless_target(parallel_pi)
```

Figure 7: Integrating *Cppless* into the build system requires minor adjustments to serverless build targets.

4.1 User Library

The *Cppless* runtime library hides both new language extensions and all vendor-specific cloud interfaces. We use the classic example of parallelizing PI estimation (Listing 6) to demonstrate how users can implement serverless functions with *Cppless*, while not making their code dependent on additional third-party SDKs and keeping the function code united with the main application. In order to execute the computation serverless, the user creates an instance of a dispatcher configured for the selected cloud system (9). Each instance of that dispatcher acts as a namespace for invocations. To invoke the function concurrently across 128 instances of an AWS Lambda function, the user calls the `cppless::dispatch` function (19), which will order the *Cppless* compiler to turn the user C++ lambda function (17) into an AWS Lambda function in the cloud. Users can optionally configure resources assigned to the cloud function, such as memory and temporary storage (11-14). The configuration will be converted to metadata by the compiler and attached to the generated function code. Then, `cppless::wait` waits for all invocations to finish (20). The *Cppless* runtime deserializes the results and can be read directly by the user and merged (22). At the runtime, the call to `cppless::dispatch` triggers a function invocation. The values of `n` and `np` captured in the C++ lambda function are serialized (17), and *Cppless* uses internally the C++ library `cereal` [14] 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.

CMake Integration. To generate and deploy serverless functions, users add *Cppless* to their project build system (Listing 7). The deployment script is integrated into the build system using a set of CMake functions. The application has to be built effectively twice: once for the host architecture and once for the architecture of remote functions. The compiler flags and configuration values used in these two targets might not be the same, requiring using different CMake configurations. Internally, separate build configurations are managed by using the `ExternalProject` functionality of CMake. These details are hidden from the user by exposing a CMake function `aws_lambda_serverless_target`. The function creates an additional cross-compiled target that is deployed as a serverless function to the cloud at compilation time.

Dispatcher. The interface demonstrated above is a *fork-join* style API based on a low-level dispatcher interface, which is based on sending tasks in the form of serializable and identifiable function objects. Dispatchers encapsulated an interface of a single cloud provider, allowing to easily switch between different systems without requiring users to rewrite their applications. Dispatchers interact with the compilation pipeline through the metadata system. This allows to correctly identify functions and define configuration options on a per-function level directly in the C++ application.

Internally, we implement two methods of generating HTTP requests to trigger serverless functions in AWS, an HTTP/2-based implementation with `nghttp2` [30], and an HTTP/1.1-based implementation that uses the `Boost.Beast` library [12]. Both solutions have different trade-offs: while HTTP/2 is faster and more efficient when many different requests are to be sent at the same time, Boost-based solution is more flexible and portable. The nghttp2-based dispatcher uses round-robin scheduling to assign requests to a pool of connections to the AWS Lambda API, which allows us to support many concurrent requests. Furthermore, using a pool of connections decreases the probability of head-of-line blocking problems. On the other hand, the Boost-based implementation issues a TCP-backed HTTP request for each invocation, which means that the number of concurrent requests is limited by the space of file descriptors available to the user process.

Alternative Entry Points. 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 to generate code for many serverless functions from a single C++ translation unit. From a user perspective, alternative entry points are an annotation 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 expected to neither use the alternative entry points directly nor be aware

of their existence, as this compiler feature is hidden behind the `cppless` interface.

4.2 Serverless Compiler in LLVM

The changes to the Clang compiler are limited to 800 lines of code added and less than 200 modified.

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, ad 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. Additionally, alternative entry points are type-checked similarly to main function definitions, leveraging existing semantic analysis.

Code Generation. The LLVM code generation process is modified to support alternative entry points and output function names to the subsequent compilation steps. Furthermore, the expression associated with the metadata attribute of alternative entry points can be evaluated as a constant expression. The result is converted to a `std::string` available to the compiler program, and the binary string is attached to the corresponding LLVM function.

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 and renamed for each entry point. We then generate code for all modules, creating separate object files for each alternative entry and the original binary for the host application. At this point, we also output 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-ld`, a cross-platform linker that can handle multiple output files when alternative entry points are present. `cppless-ld` accepts the same command line interface as Clang, using the same Clang toolchains to support linking for different platforms. Our linker reads manifest files from the compilation which describes alternative entry points, and uses the original Clang driver to link them. The linker produces one regular output file and additional configuration files for each alternative entry point, while merging the manifest files into the output.

4.3 Lambda Functions

We implemented in *Cppless* two new language features to support dispatching C++ lambda functions (Listing 8). First, to support the serialization of function arguments, we implemented a compile-time, constexpr-compatible reflection mechanism for lambda functions. 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, including passing the captured value to serialization.

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

  auto func_id =
    __builtin_unique_stable_name(
      decltype(lambda)
    );

  invoke(func_id, first_capture);
}
```

Figure 8: Pseudocode of serverless invocation demonstrating the new lambda reflection and identification features.

Then, we created a unique identification system to connect entry points with the invoked function. We implemented an internal function that takes a type as an argument, and returns a literal string used to identify the type that was added. This function is backed by the Clang implementation of `builtin-sycl-unique-stable-name`, a feature added for to support the SYCL framework [19], which has a similar use-case. The function generates a mangled type name but uses a slightly modified Itanium mangling scheme. The main change we implemented was removing inlined namespaces from the mangling prefix. This increases compatibility between different standard library implementations (Sec. 2.3) and improves the stability of these identifiers.

4.4 Limitations

Serializable Function Objects. Requiring that tasks are serializable functions permits using some constant function pointers, i.e., a pointer where the exact name of the function that will be called is known at the 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 the user-space design to treat function pointers as regular lambdas if possible.

Dependencies. The linker used in *Cppless* treats all alternative entry points in the same way, which requires using the same set of linking dependencies for each function. However, the alternative entry points can be unbloated after linking is completed by examining which symbols of a library are used, reducing the size of the

		Time [ms]	Throughput [GiB/s]
`binary`	Encode	5.90	1.32
	Decode	3.18	2.46
`binary_json`	Encode	13.03	0.60
	Decode	28.63	0.27
`structured_json`	Encode	462.40	0.02
	Decode	144.15	0.05

Figure 9: Benchmarking serialization of an array of unsigned integers.

		Time [ms]	Throughput [GiB/s]
`binary`	Encode	97.35	0.3061
	Decode	81.48	0.3658
`binary_json`	Encode	131.62	0.60
	Decode	158.62	0.041
`structured_json`	Encode	726.51	0.041
	Decode	650.27	0.0458

Figure 10: Benchmarking serialization of an array of structures.

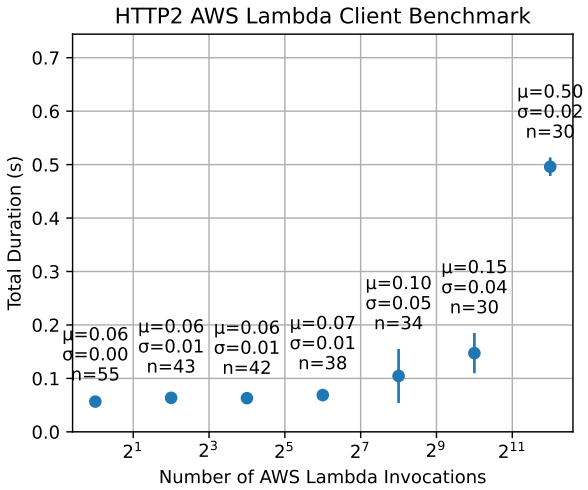

Figure 11: The latency of concurrent invocations of warm AWS Lambda functions with the HTTP dispatch, as observed by the client.

executable. An alternative would be an integration into the cppless metadata system which could enable control over the linkage process through compile-time directives.

Compilation Time. As multiple LLVM modules are produced and then lowered to the target language independently, the compilation time increases linearly with the number of alternative entry points. This limitation could be fixed by integrating the cloning process deeper with LLVM. Currently, we limited ourselves to using the public LLVM APIs to improve the maintainability of *Cppless*, which do not support such integration.

5 EVALUATION

We demonstrate the ease of programming and parallel offloading in *Cppless* with micro-benchmarks, application from the Barcelona OpenMP Task Suite (BOTS) [11], and a CPU RayTracing application [28]. Benchmarks were executed on a `t3.medium` virtual machine instance in the AWS *eu-central-1* region, with 4 GiB RAM, 2 vCPUs, and a 5 Gb/s network connection. AWS Lambda functions are configured by default with a memory limit of 1 GiB, except for the N-Queens benchmark, where we use Lambda instances with 2 GiB of memory. We compile all benchmarks with the `O3` optimization level, and measure only the runtime of applications, excluding the compilation and cloud deployment times, excluding the first three warmup iterations.

5.1 Microbenchmarks

We use microbenchmarks to analyze the performance of two critical parts of *Cppless* runtime: serialization and the AWS Lambda client that the dispatcher uses to invoke tasks. The choice of serialization format directly impacts the performance of serverless offloading. On the other hand, the function dispatch must scale up to thousands of invocations to allow for massively parallel processing.

Serialization. Serverless systems put constraints on the type of data that can be transmitted to a function. For example, AWS Lambda requires that function input and output are valid JSON objects. We evaluate two JSON-based scenarios of serializing C++ objects with the `cereal` library: one using direct JSON serialization, and the other one using binary serialization and `base64` encoding of the resulting data to create a valid JSON object. We compare both scenarios against the plain binary serialization.

First, we benchmark the serialization of an array of 1000000 64-bit unsigned integers (Table 9). The overhead of the *binary_json* format can be attributed to the additional base64 encoding that has to be executed. On the other hand, the JSON archive performs much worse due to number parsing and additional memory allocations. Then, we evaluate the serialization of a structure consisting of two integers and a single string, with a custom serialization method (Table 10). This serialization is more expensive due to pointer jumping and more complex encoding of a `std::string`. Nevertheless, our custom `binary_json` format is up to 5.52x faster than a vanilla JSON serialization.

AWS Lambda Client. We examined the latency of parallel invocations with the custom AWS Lambda client used in *Cppless*. The client uses Boost ASIO abstractions to dispatch dozens of parallel invocations without allocating a separate thread for each task. In our test setup, 16 HTTP/2 connections are employed, each one handling up to 100 concurrent requests. For the purpose of this benchmark, the client invokes an AWS Lambda function implemented in Node.js, hosted on an account with a maximal function concurrency of 1000.

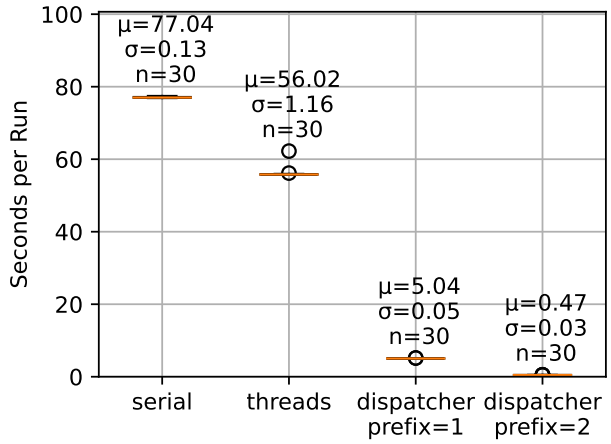

Figure 12: N-Queens benchmark for $N = 17$. Serverless results are obtained with prefix values of 1 and 2.

As shown in Fig. 11, the latency for a single invocation is equal to around 50 ms, increasing linearly to approximately 150 ms until the client resources are exhausted. At this point, invocations are not dispatched directly anymore but have to wait until the response of pending invocations arrives. The client latency includes a constant initial latency for connecting to the AWS API, and afterward, HTTP/2 requests reuse the underlying TCP connection and can pipeline multiple requests at once. After the connection initialization, the client dispatches invocations at a rate of around ten invocations per millisecond, as long as either the AWS Lambda concurrency limit or dispatcher resources are not exhausted. This allows using thousands of parallel invocations for small tasks, even in cases where the offloaded tasks take a few dozen of milliseconds to complete.

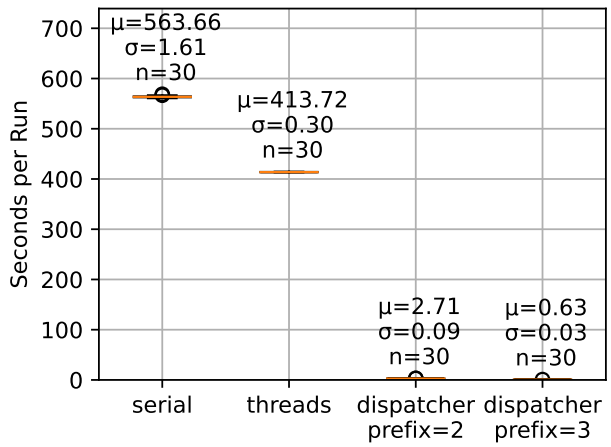

Figure 13: N-Queens benchmark for $N = 18$. Serverless results are obtained with prefix values of 2 and 3.

5.2 N-Queens

The N-Queens problem is defined as finding 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. A commonly employed solution is backtracking, where queens are placed row by row incrementally, and the algorithm backtracks when placement becomes impossible.

While typical board representation involves arrays, this is inefficient as placing a queen requires a search over the entire array. Thus, we replace the array-based implementation from the BOTS benchmark suite with one that uses bit patterns to represent board states [26], improving the performance of determining queen placement. To parallelize finding the solution, we use *prefix* tasks of length p where the location of the first p queens is fixed, allowing us to break down the primary problem [20]. The length of the prefix determines the number of tasks that are generated and can be offloaded, with a longer prefix creating more subtasks and increasing the available parallelism. The created tasks are offloaded to serverless workers, and results are accumulated to create the final solution.

We compare the local serial and parallel computation with the dispatch to serverless. The local parallel implementation requires 25 lines of code, while a solution with the serverless *Cppless* dispatcher is implemented with 36 lines of code. Performance results for the N-Queens problem with N equal to 17 and 18 are presented in Figures 12 and 13, respectively. Offloading computations to serverless functions provides speedups of up to 164x and 894x for $N = 17, p = 2$ and $N = 18, p = 3$, respectively. When compared to the number of generated parallel tasks, which is equal to 240 for the first scenario and 3420 for the second one, these results indicate that serverless offloading does not achieve linear scaling. This result is explained by the variance in the workload assigned to different tasks, with the total execution time limited by the longest-running task and the overhead of assembling results on the host machine. The heterogeneous task workload highlights the benefits of computing with serverless functions: the pay-as-you-go pricing model ensures costs are proportional to the actual work done by all tasks, eliminating the need to consolidate smaller tasks. Thus, waiting for the longest-running task to finish does not incur any additional cost as all other parallel workers are not generating any charges.

5.3 Raytracer

As a second application, we consider a Monte-Carlo implementation of ray tracing [28]. The implementation incorporates a bounding volume hierarchy mechanism [29], and it has been implemented to use the AVX2 vector instructions. The benchmark scenario renders a random scene, which is divided into smaller tiles to create parallel tasks. As for the previous benchmark, we compare the performance of local serial and parallel computation on a small virtual machine against offloading these tasks to serverless functions. In this example, using serverless workers requires adding 13 more lines of code.

Performance. We evaluate the benchmark on an image of size 500x500 and show that serverless computing with *Cppless* provides

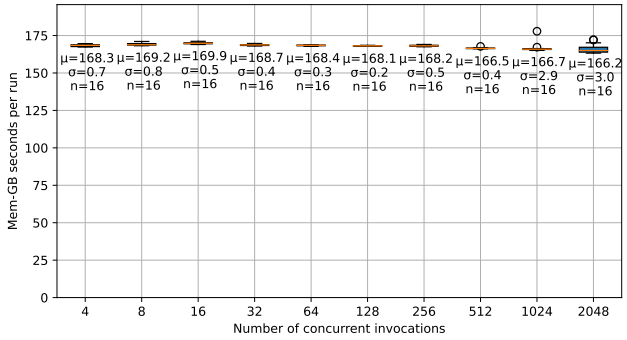

Figure 14: The total cost of parallel computations in the Ray-Tracer benchmark, expressed in the gigabyte-seconds metric of AWS Lambda, i.e., a total computation time of all functions multiplied by the function memory size.

a speedup of up to 34x (Fig 1, p.1). This result is already attractive for the end user, as their compute-intensive task taking over 40 seconds can now be completed in less than two seconds, without having to allocate a dedicated and more powerful virtual machine with multiple cores.

However, the obtained speedup is lower than the number of parallel workers, which is equal to 256 (tile 32x32) and 1024 (tile 16x16), and the culprit of lower scalability is unequal work distribution. By analyzing the latency distribution of serverless invocations, we found out that the overall latency is dominated by a few long-running functions. While the median task latency scales almost perfectly with the tile size, the maximum workload only diminishes by approximately 40% when the tile size is halved. This uneven workload distribution is due to varying per-pixel workloads. The computation time of each task varies and depends on the objects present in the assigned tile.

Furthermore, we analyzed the time required to spawn the invocations on the client side to verify that the serialization overhead and accumulation of results are not the scalability bottleneck. The complicated and structured scene graph creates approximately 88 KiB of data for each invocation, and the serialization takes around 40% of the invocation overhead latency. Nevertheless, *Cppless* runtime can spawn more than 2000 such tasks per second, indicating that our runtime can support massively parallel computations, even when operating from a virtual machine with constrained resources.

Cost. To estimate the overheads that larger scales of parallelism with serverless, we executed the same workload with a varying number of parallel tasks offloaded to functions. Then, we queried the billing data provided by the cloud operator AWS to summarize the total computation cost. The cost metric expresses the total computation time across all tasks. As shown in Fig. 14, the cost of the computation is barely affected by the parallelism scale. Although the result is available more quickly, the cost stays almost constant, emphasizing the benefit of the pay-as-you-go billing model. Specifically, for this benchmark, the task duration for the smallest tile size varies between 8 and 150 milliseconds. Even with invocations in the single-digit millisecond range, the cost of the computation is dominated by the productive work, as a cost increase with a larger number of functions would indicate inefficiencies introduced by *Cppless* workers.

6 CONCLUSIONS

We present *Cppless*, an innovative approach to define serverless functions within C++ applications. Despite the inherent constraints of C++, *Cppless* offers a streamlined way for programmers to transparently offload tasks to serverless platforms. *Cppless* defines compiler extensions and employs meta-programming techniques to transition to compile time the work typically done at the runtime, helping to diminish the serialization overheads and minimizing the required changes to the core language. With a selection of parallel application benchmarks, we demonstrate that serverless invocations can be effortlessly integrated with *Cppless*, achieving high scaling with negligible overhead.

ACKNOWLEDGMENT

This project has received funding from EuroHPC-JU under grant agreements DEEP-SEA, No 95560, and RED-SEA, No 055776. 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 supporting this research with credits through the AWS Cloud Credit for Research program.

REFERENCES

[1] 2014. AWS Lambda. https://aws.amazon.com/lambda/. Accessed: 2020-01-20.
[2] 2022. The State of Serverless. https://www.datadoghq.com/state-of-serverless/. Accessed: 2023-08-16.
[3] 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.
[4] 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
[5] 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
[6] Marc Brooker, Mike Danilov, Chris Greenwood, and Phil Piwonka. 2023. On-demand Container Loading in {AWS} Lambda. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. 315–328.
[7] Marcin Copik, Alexandru Calotoiu, Pengyu Zhou, Konstantin Taranov, and Torsten Hoefler. 2023. FaaSKeeper: Learning from Building Serverless Services with ZooKeeper as an Example. arXiv:2203.14859 [cs.DC]
[8] 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
[9] 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
[10] 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
[11] 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.

[12] Vinnie Falco. 2016. Beast: C++ HTTP and WebSocket built on Boost.Asio. https://github.com/boostorg/beast.

[13] 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

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

[15] 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

[16] Emily Herbert and Arjun Guha. 2020. A Language-based Serverless Function Accelerator. arXiv:1911.02178 [cs.DC]

[17] 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)*.

[18] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Jayant Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. 2019. Cloud Programming Simplified: A Berkeley View on Serverless Computing. *CoRR* abs/1902.03383 (2019). arXiv:1902.03383 http://arxiv.org/abs/1902.03383

[19] 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

[20] 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).

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

[22] Ingo Müller, Renato Marroquin, and Gustavo Alonso. 2019. Lambada: Interactive Data Analytics on Cold Data using Serverless Cloud Infrastructure. *ArXiv* abs/1912.00937 (2019).

[23] Matthew Perron, Raul Castro Fernandez, David DeWitt, and Samuel Madden. 2020. Starling: A Scalable Query Engine on Cloud Functions. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 131–141. https://doi.org/10.1145/3318464.3380609

[24] 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

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

[26] Martin Richards. 1997. *Backtracking algorithms in MCPL using bit patterns and recursion*. Technical Report. Citeseer.

[27] 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

[28] Peter Shirley. 2020. *Ray Tracing in One Weekend*. https://raytracing.github.io/books/RayTracingInOneWeekend.html

[29] Peter Shirley. 2020. *Ray Tracing: The Next Week*. https://raytracing.github.io/books/RayTracingTheNextWeek.html

[30] Tatsuhiro Tsujikawa. 2013. nghttp2 - HTTP/2 C Library. https://github.com/nghttp2/nghttp2.

[31] 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

[32] 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