KERYX: A CUDA/HIP Framework for Adaptive, Runtime Compilation on Heterogeneous Systems

Marc Gonzalez Tallada
Universitat Politècnica de Catalunya (UPC)
Barcelona, Spain
marc.gonzalez@upc.edu

Pedro Valero Lara
Oak Ridge National Laboratory (ORNL)
Oak Ridge, TN, USA
valerolarap@ornl.gov

Keita Teranishi
Oak Ridge National Laboratory (ORNL)
Oak Ridge, TN, USA
teranishik@ornl.gov

Jeff Vetter
Oak Ridge National Laboratory (ORNL)
Oak Ridge, TN, USA
vetter@ornl.gov

ABSTRACT

This paper describes *KERYX*, a framework for runtime compilation (RTC) in heterogeneous computing systems. *KERYX* is embedded in the *C++* language and implemented as a set of *C++* libraries. *KERYX* enables the device code to be described in *C++* instead of strings. *KERYX* also solves template code expansion by avoiding complex interactions with the device driver for name resolution of template code. *KERYX* defines a high-level intermediate representation (IR) that is close to the source representation and suitable for high-level code transformations. With this, *KERYX* makes it possible to introspect and transform the kernel code at runtime and invoke the existing device driver RTC support for binary generation. The authors used *KERYX* to code nine different benchmark examples from different domains in the CUDA samples and three applications from the NPB-MZ suite in the scientific computing domain. On two different heterogeneous systems, one with two NVIDIA A100 GPUs and the other composed of an AMD Instinct MI300A, *KERYX* enables the execution of the same binary application, with observed RTC deviations over 76 kernels and geometric deviations of 0.95× and 1.07× of the total execution kernel time, resulting in similar performance levels as those observed for native CUDA/HIP code with static compilation. As a case study for adaptive compilation, *KERYX* enables kernel fusion at the application level, with speedups between 1.5× and 2.98×, exposing the IR introspection and transformation capabilities to implement dynamic optimizations. *KERYX* drastically improves programming for applications that need CUDA/HIP RTC support.

CCS CONCEPTS

• **Computing methodologies** → **Parallel computing methodologies**; **Parallel programming languages**;

KEYWORDS

Heterogeneous Systems, GPU programming, Runtime Compilation

ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of a national government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only. Request permissions from owner/author(s).
PACT '26, October 19–22, 2026, Chicago, IL, USA
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2915-7/2026/10
https://doi.org/10.1145/3838684.3846867

ACM Reference Format:
Marc Gonzalez Tallada, Pedro Valero Lara, Keita Teranishi, and Jeff Vetter. 2026. KERYX: A CUDA/HIP Framework for Adaptive, Runtime Compilation on Heterogeneous Systems. In *International Conference on Parallel Architectures and Compilation Techniques (PACT '26), October 19–22, 2026, Chicago, IL, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3838684.3846867

1 INTRODUCTION

The landscape of high-performance computing (HPC) has undergone a fundamental shift toward heterogeneous computing architectures. Unlike traditional homogeneous systems that rely solely on CPUs, heterogeneous systems can integrate a variety accelerators, most notably GPUs, to handle data-parallel workloads. This paradigm is now an essential stream in HPC because it offers orders of magnitude improvements in throughput and energy efficiency, which are critical for modern scientific simulations, large-scale data analytics, and AI.

To harness the power of these accelerators, developers primarily rely on specialized programming models, with NVIDIA's CUDA [30] and AMD's HIP (Heterogeneous-interface for Portability) [2] emerging as the dominant frameworks. CUDA has long served as the industry standard for proprietary NVIDIA hardware, providing a robust ecosystem for low-level GPU optimization. Conversely, HIP has gained significant traction as an open-source alternative designed for portability, allowing developers to target both AMD and NVIDIA hardware from a single codebase. Together, these models provide the necessary abstractions to manage memory hierarchies and execution grids across the host-device boundary.

Despite their dominance, these programming models face significant limitations regarding runtime compilation (RTC) techniques and dynamic kernel generation. Currently, developers seeking to compile kernels during program execution are largely restricted to rigid workflows. Most frameworks rely on string-based kernel descriptions, in which the device source code must be passed as a raw text string to a Just-In-Time (JIT) compiler [1, 13, 36], and also require expensive I/O operations to load source code from external files [8, 9, 31, 33, 34]. These approaches introduce substantial overhead, complicate the integration of high-level domain-specific languages, and lack the flexibility required for highly adaptive, data-dependent execution environments.

The main contribution of this paper is KERYX, a framework for adaptive and RTC designed to bridge the gap between high-level application logic and low-level device drivers. KERYX facilitates the direct *delivery* of compute kernels to the hardware, bypassing the high-latency I/O operations and manual string manipulations that characterize current CUDA/HIP RTC workflows. KERYX is embedded in the *C++* language and implemented as a *C++* library. Because of KERYX's *C++* embedding, the device code can also be described in *C++*. KERYX improves overall device programming: code is not described by using strings, *C++* templates are available at compile time, and instantiating versions of the code to be compiled at runtime solves this main limitation in current CUDA/HIP RTC support. KERYX defines a high-level intermediate representation (IR) that is close to the source representation and suitable for high-level code transformations. KERYX also includes an API to manipulate and transform the kernel code at runtime. With KERYX, it has been possible to implement runtime kernel fusion [4, 11, 15, 27, 38–40] over a set of tensor operations. Because KERYX facilitates the RTC process with primitives that hide the complexities in current CUDA/HIP device driver specifications, KERYX also improves portability as a side effect, owing to its dynamic compilation nature.

In this work, the authors used KERYX to code nine benchmark examples from different domains in the CUDA samples [21] and three applications from the NPB-MZ suite [12, 23] in the scientific computing domain. Additionally, runtime kernel fusion for a tensor library [11] was evaluated as a case of study for adaptive compilation with KERYX, achieving speedups between 1.5× and 2.75× compared with non-fused kernels. The present work also analyzes the overheads introduced by the KERYX RTC support as well as the performance of the generated code for the KERYX kernels. The authors used two different heterogeneous systems, one equipped with two NVIDIA A100 GPUs [32] and another equipped with an AMD Instinct MI300A APU [3]. Using either system, KERYX enables the execution of the same binary application, with observed geometric mean values of 0.95× (for CUDA) and 1.07× (for HIP) of average RTC deviations from total execution kernel time, resulting in similar performance levels as those observed for native CUDA/HIP code with static compilation.

2 KERYX FRAMEWORK

The KERYX framework is composed of a *C++* embedded programming language and support for RTC. Both KERYX components are implemented as *C++* libraries with classes and operators that enable the description of a computation targeting a heterogeneous architecture.

2.1 Overview

Listing 1 shows the full KERYX SAXPY benchmark code. In (1), a KERYX expression *keryx::expr* includes the description of the SAXPY code using all the *C++* classes that implement the language (the *#include* directive includes all KERYX declarations). The *C++* lambda expression contains the KERYX code. The lambda is captured by the KERYX implementation, and its later execution will trigger all the IR building for the KERYX code within it. The *keryx::kernel* (2) corresponds to a *C++* class that, when its constructor is invoked,

```
#include "keryx-lang.h"
keryx::expr Code = [] { // (1) Expression containing the SAXPY code
  keryx::kernel // (2) Kernel definition
  saxpy("saxpy",
        [&] (keryx::in<keryx::_double> a,
             keryx::in<keryx::_double*> x,
             keryx::in<keryx::_double*> y,
             keryx::out<keryx::_double*> out,
             keryx::in<keryx::_int> n) {

    keryx::_int tid; // (3) Integer variable definition
    // (4) Statement computing the thread id
    tid = keryx::blockIdx.m<"x"> * keryx::blockDim.m<"x"> +
          keryx::threadIdx.m<"x"> ;
    // (5) The kernel computation
    keryx::_if (tid < n, out[tid] = a * x[tid] + y[tid]);
  });
};
int main () {
  keryx::program P(Code, "Saxpy Benchmark"); // (6) Program build
  keryx::compile(P); // (7) Code compilation
  keryx::kernel SAXPY(P, "saxpy"); // (8) Kernel extraction

  void *dX, *dY, *dOut;
  keryx::alloc(&dX, bufferSize); // Alloc device memory for SAXPY.
  keryx::alloc(&dY, bufferSize);
  keryx::alloc(&dOut, bufferSize);
  // Move data to device memory for SAXPY.
  keryx::copyHtoD(dX, hX, bufferSize);
  keryx::copyHtoD(dY, hY, bufferSize);
  // (9) Execute SAXPY.
  void *args[] = { &a, &dX, &dY, &dOut, &n };
  keryx::run(SAXPY, keryx::grid(NUM_BLOCKS,1,1),
                    keryx::block(NUM_THREADS,1,1), 0, args);
  keryx::copyDtoH(hOut, dOut, bufferSize);
}
```

Listing 1: SAXPY benchmark in KERYX.

defines a kernel builds the IR. The arguments of the *kernel* construct correspond to an identifier and the kernel body that appears as a *C++* lambda expression. The implementation captures its argument types and forwards this information to the implementation. In (3), an integer variable is defined using the construct *keryx::_int*. In (4), the common GPU thread organization primitives are available in the form of constructs *keryx::blockIdx, keryx::blockDim, and keryx::threadIdx*. In (5), the conditional execution of the actual SAXPY operation is shown. The *keryx::_if* construct corresponds to the common *if-then-else* control flow construct. The *condition* corresponds to the first argument, and the *then* corresponds to the second argument.

In general, KERYX code is expressed using the constructors of the *C++* classes that implement the language. The execution of these constructors triggers the IR building. The actual embedding of the KERYX language is done through *C++* lambda expressions that are used to contain the code and define the KERYX contexts.

Listing 1 also shows how the KERYX code can be compiled and generate a kernel for its execution. A program is built with the KERYX code (6), and then the program is compiled (7). After this step, the KERYX code has a binary representation ready for execution. In (8), the SAXPY kernel is located, and class *keryx::kernel* acts as a bridge to the actual device driver interface for the kernel execution. Finally, in (9), the kernel is launched with its arguments. The KERYX API also includes common primitives to allocate memory and move data to/from the host and device.

In summary, KERYX includes both a front end to describe the device kernels and an API to enable RTC and the actual kernel execution. Beyond these features, KERYX also includes an API to

	Primitive	Types	Control Flow	Variable/Functions	Device	
Keryx Constructs	op sema def attr context expr literal	_char _int _bool _float _double _struct _volatile _const	_if _for _while _dowhile _break _continue _return	var decl fn in out inout kernel _operator _namespace	blockIdx threadIdx gridDim blockDim	shared constant syncthreads memfence
	Operators					
	a+b, a-b, a*b, a/b, a%b, a^b, a&b, a|b, ~a, !a, a=b, a<b a>b, a+=b, a-=b a*=b, a/=b, a%=b, a^=b, a&=b, a|=b, a<<b, a>>b, a>>=b, a<<=b, a==b, a!=b, a<=b,a>=b, a<=>b, a&&b, a||b a++, b--, a->*b, a(), a[b]					

Figure 1: Core of the KERYX programming language.

manipulate the IR. This makes it possible to modify the IR and apply code optimizations and transformations before the actual binary code generation. This feature is essential. For current RTC technology in CUDA or HIP, the code is described by using strings. In *KERYX*, this changes completely, and the code can be transformed by using compiler techniques that manipulate an IR. The following sections describe the *KERYX* definition with its main components: front end, type system, constructs, and contexts/namespaces. Figure 1 shows the core of the *KERYX* language addressed in the following sections. In general, *KERYX* builds upon the C/*C++* core language features, dropping aspects that are not considered by the CUDA/HIP specifications (e.g., *KERYX* does not include an object model with class derivation and polymorphism).

2.2 Front End

The fundamental building blocks of the *KERYX* language are literals/identifiers, attributes, sema, definitions, operators, and semantic contexts.

Identifiers/Literals: An identifier/literal is a valid *C++* literal used within *KERYX* constructs. Because *KERYX* is embedded directly in *C++*, any valid *C++* identifier/literal is also considered a valid *KERYX* identifier.

Attributes and Sema: An attribute identifies a specific abstraction that assigns semantics to a definition or expression. Common examples of *KERYX* attributes include *Type* (matching the standard concept of an object's type), *Computation* (describing the executable code associated with a function), and *Name* (specifying the designation of a definition or expression). A *sema* represents a concrete binding of these abstractions and consists of a pairing of an attribute and a *KERYX* expression that defines the value for that attribute. Therefore, *sema* is the mechanism used to attach semantics to definitions and expressions.

Definitions: A *KERYX* definition is composed of a list of *sema*. All definitions must include a value for the *Name* attribute.

Operators: An operator corresponds to a functional language construct. For example, standard iterative control flows (e.g., *while*, *do-while*, and *for*) are implemented through the definition of operators. An invocation of an *op* represents a specific instance of the construct, and the operands of the invocation constitute the structural components of that construct.

Expressions: *KERYX* expressions are the intermediate objects formed when the *KERYX* program generates identifiers, definitions, or operator invocations.

Semantic Contexts: A *KERYX* semantic context is defined as a sequence of expressions and definitions. The primary role of a context is to group these elements under specific scoping rules, which determine their visibility and resolution.

The implementation of *KERYX* provides a specific *Abstract Data Type* (ADT), which is a *C++* class, for each of the elements described above. The constructors of these classes serve as the front-end interface for the language. Consequently, every time a class constructor is invoked, a corresponding *KERYX* construct is generated, and the underlying IR is immediately built. *KERYX* establishes a direct correspondence between a language construct and the invocation of a class constructor. For convenience, we refer to any such class constructor as a *KERYX operator*. In summary, every *KERYX* expression resolves to one of the following: an identifier (e.g., a captured *C++* string literal), an attribute, a sema, a definition, an operator invocation (via a class constructor call), or a context expression (via a captured *C++* lambda).

2.3 Type System

KERYX is a statically and strongly typed language. In *KERYX*, types are represented as expressions and classified as primitive (e.g., natural, integer, real, boolean, and string) and composite (e.g., struct).

2.3.1 Primitive Types. *KERYX* includes the definition of primitive types such as character, natural, integer, floating-point, and boolean data types. These types match those that already exist in *C++*.

2.3.2 Composite Types. *KERYX* supports the definition of composite types through the *keryx::_struct* ADT. This type matches that of the C language's *struct*. *KERYX* does not deploy an object-oriented programming model (class derivation and polymorphic types are not supported). Listing 2 (1) shows an example for the definition of type *pixel* as a composite type with 3 data members of an integer type. The description of the composite type is given by the definition of a *C++* lambda expression that contains all the data member definitions. In (2), the *pixel* type is defined, and the expression *pixel_struct* is associated with the *pixel* type. On the next line, the *C++ using* declaration creates a shortcut to use the *keryx::type<"pixel">* as a *C++* type to define variables of type *pixel*. In (4), a variable of type *pixel* is defined, and the members of the variable are initialized. In *KERYX*, the access to a data member of a composite type is expressed through the *"m"* data member of the *C++* class that represents the composite type (e.g., the *keryx::type<"pixel">* class). The *KERYX* implementation solves the association between the strings used to identify the *struct* field access and the definition of the *struct* itself (notice the string argument in each member of the *struct* in (1)).

2.4 Contexts and Namespaces

In *KERYX*, contexts are defined through *C++* lambda expressions that are captured by the *KERYX* implementation. Depending on where the lambda appears, the semantic of the lambda is different and the code inside the lambda is treated accordingly. For instance, in Listing 1, the lambda associated with the *for* construct is treated as a *C++* compound statement. The lambdas in Listing 2 are treated as a *namespace*. A named *namespace* can be defined with a *C++* lambda as an argument of the *keryx::expr* construct and by setting

```
keryx::_struct<"pixel"> pixel_struct([&] { // (1) Define the struct
  keryx::_int r("r"_i);
  keryx::_int g("g"_i);
  keryx::_int b("b"_i);
  keryx::_operator<"a+b">(keryx::_struct<"pixel">);
});
keryx::type<"pixel">(pixel_struct); // (2) Define a type
using pixel = keryx::type<"pixel">;

keryx::_namespace example("Example", [&] { // Define the namespace

// Code and definitions in namespace
pixel p; // (3) Define a variable using the new type

keryx::fn<void> init([&](pixel p) {
  p.m("r") = 123; // (4) Update a data member of a composite type
  p.m("g") = 14;
  p.m("b") = 2;
}); // function

}); // namespace
```

Listing 2: Examples of KERYX struct and namespace.

the *Name* attribute for the expression to the actual name of the *namespace*.

2.5 Constructs

2.5.1 C++ Operators. As stated previously, within the KERYX implementation, a language construct matches the invocation of an operator (e.g., a call to a class constructor). Within KERYX, there are specific *primitive* operators that require special treatment by the implementation. These correspond to the building blocks of language described in Section 2.2. Notably, some C++ operators (e.g., +, -, *, /, ...) are also considered primitive and require special treatment. All other operators are considered non-primitive. KERYX defines specific operators for all the C++ native operators that support overloading. For example, the binary C++ *operator +* corresponds to *op<"a+b">*. In general, for all these C++ operators, the KERYX implementation includes a specific overload that captures its usage and activates the corresponding IR building for the expression in which the operator appears.

2.5.2 Control Flow. KERYX includes common control flow constructs such as *while, do-while, for,* and *if-then-else*. For each one, KERYX defines an operator (e.g., an ADT). For example, the *if* statement (5) in the SAXPY code (Listing 1) corresponds to the invocation of operator *keryx::_if = keryx::op<"if-then-else">*, that is, the execution of the constructor of class *keryx::op<"if-then-else">*, which takes the *condition-then-else* parts of the *if* construct as arguments. This constructor triggers the IR building for the construct. The rest of control flow constructs (e.g., *keryx::op<"for">*, *keryx::op<"while">*, *keryx::op<"do-while">*) operate in the same manner in terms of IR building.

2.5.3 Variables and Functions. KERYX variables are defined with the *keryx::var* construct (Listing 4). It is possible to use C++ types (e.g., float, double, unsigned int, int) and type modifiers (e.g., **, &, const, volatile*) for the type specification. The KERYX implementation captures these modifiers and generates a KERYX expression to represent the type. For example, Listing 4 shows several variable definitions that combine the native C++ type modifiers and the KERYX native types. If no C++ type modifiers are used, then it is possible to skip

```
keryx::kernel
matrix_mul([&] (keryx::in<int *> a, keryx::in<int *> b, keryx::in<int *> c,
                keryx::in<int> m, keryx::in<int> n, keryx::in<int> k) {
    keryx::var<int> row, col;
    row = keryx::blockIdx.m<"y"> * keryx::blockDim.m<"y"> + keryx::threadIdx.m<"y">;
    col = keryx::blockIdx.m<"x"> * keryx::blockDim.m<"x"> + keryx::threadIdx.m<"x">;
    keryx::var<int> sum = 0;
    keryx::_if( col < k && row < m, [&] {
        keryx::var<int> i;
        keryx::_for( i = 0, i < n, i++, [&] {
            sum += a[row * n + i] * b[i * k + col];
        });
        c[row * k + col] = sum;
    });
});
```

Listing 3: KERYX kernel for matrix multiply.

the usage of the *keryx::var* construct and define a variable using the KERYX type (this corresponds to (6) in Listing 4). KERYX functions can be specified with the construct *keryx::fn = keryx::op<"function">*. The function is defined through the capture of a C++ lambda expression that supplies both the signature and the body of the function. The KERYX implementation captures the lambda and extracts the parameter types and return type and then converts the lambda into a callable object from the internal implementation. This object remains stored in the KERYX IR. The *keryx::kernel* corresponds to a code device entry point. Listing 1 shows the kernel example for a SAXPY operation. The argument types and return type are specified with the *in*, *out*, and *inout* constructs to indicate whether the function reads/modifies each argument. Variable and function declarations are supported with the *keryx::decl* construct ((4) and (5) in Listing 4).

Listing 3 shows an example of KERYX code for a matrix multiply device kernel. The code has not been optimized at all and is used simply to facilitate the description of KERYX. The kernel is defined with the *keryx::kernel* class, and the KERYX implementation captures the signature and body of the function definition (C++ lambda expression). The body contains several variable definitions and the actual computation description with a *keryx::_for* construct. Variables *row* and *col* are initialized by using the GPU thread organization primitives in KERYX. Notably, the code leverages C++ operators (e.g., +, +=, <, ++, []), which are captured by the KERYX implementation for IR building. The same happens with the *keryx::_for* and *keryx::_if* constructs, which will trigger their corresponding IR building.

2.6 Device, Thread, and Memory Models

KERYX follows the same thread organization and memory model used in the CUDA/HIP programming models. KERYX recognizes the thread organization primitives (e.g., *threadIdx*, *blockIdx*, *gridDim*) from CUDA/HIP. KERYX also includes device memory hierarchy constructs (e.g., *keryx::shared* and *keryx::constant*) and consistency and synchronization primitives (e.g., *keryx::mem_fence*, *keryx::syncthreads*). All of them map to their counterparts in the GPU execution model. For brevity, only some of them are listed here.

2.7 Templates

Because KERYX is embedded in C++, the KERYX code can be integrated into the C++ template engine. This means that it is possible

```
keryx::var< keryx::_int > i; // (1) defines integer
keryx::var< const keryx::_int* > i; // (2) defines pointer to const integer
keryx::var< volatile keryx::_int > i; // (3) defines volatile integer
keryx::decl<"i",  keryx::_int> i; // (4) declares a variable of type
                                  // integer and named "i"
// (5) declare a SAXPY kernel
keryx::decl<keryx::kernel<keryx::_double, keryx::_double*, keryx::_double*,
                          keryx::_double*, keryx::_int> > saxpy;
keryx::_int ii; // (6) defines integer
keryx::constant<int[256]> constant_values; // (7) Constant variables
```

Listing 4: KERYX variable examples.

```
#include "keryx-lang.h"
  // (1) Template kernel definition
  template<class T>
  keryx::kernel
  saxpy("saxpy",
        [&] (keryx::in< T > a, keryx::in< T*> x, keryx::in< T*> y,
             keryx::out<T*> out, keryx::in<keryx::_int> n) {
    keryx::_int tid; // Integer variable definition
    tid = keryx::blockIdx.m<"x"> * keryx::blockDim.m<"x"> +
          keryx::threadIdx.m<"x"> ;
    keryx::_if (tid < n, out[tid] = a * x[tid] + y[tid]);
  });
  // (2) Template instantation
  keryx::expr Code = [] { // (1) Expression containing the SAXPY code
    saxpy<keryx::_int>( ... );
    saxpy<keryx::_double>( ... );
  };
```

Listing 5: KERYX template example with SAXPY kernel.

to parameterize the *KERYX* code with template variables and make the *C++* compiler expand the templates. This very powerful feature mitigates one of RTC's main limitations in terms of template expansion and how to propagate it to the binary code. For example, in CUDA/HIP, specific driver primitives exist to instantiate *templated* code, but this requires the programmer to guide the instantiation. The CUDA/HIP API primitives must be carefully invoked in a prescribed sequence [20, 22]: first, build a template instantiation string (*(nv-hip)rtcGetTypeName*) and register it in the program (*(nv|hip)rtcAddNameExpression*) before the compilation; then, find the decorated name in the program (*(nv-hip)rtcGetLoweredName*) after the compilation. The identifier provided to this function must exactly match the name first used with *(nv-hip)rtcAddNameExpression*. This process must be repeated for each template expression to be expanded. In contrast, *KERYX* ensures that all template code is expanded at compile time by generating all necessary code versions that will be runtime compiled (e.g., Listing 5 (1) shows a template version of the SAXPY kernel). Two versions of the kernel are instantiated: one for *int* and another for *double*.

2.8 Intermediate Representation

The *KERYX* IR follows a functional representation of the program. Every *KERYX* expression is represented as an invocation to an operator. In this regard, the *KERYX* IR is very similar to the Multi-Level Intermediate Representation (MLIR) [26]. For example, the *KERYX* code in Listing 6 shows the expressions that are formed for the code (1) in the same listing. In (2), an invocation to the *keryx::ident* operator builds an identifier expression with the *"fcn"* string. In (3), a context expression is generated with an invocation to the *keryx::context* operator. The *C++* lambda expression is captured

```
keryx::expr Code = []{
  // keryx::fn<void> fcn([&] { // (1) KERYX sample code
  //     keryx::_int i;
  //     keryx::_for(i=0,i<10;i++, [&] {
  //       Loop body
  //     });
  // });
  keryx::attr Name, Type, Computation;
  keryx::ident fcn("fcn"); // (2) Function identifier
  keryx::context FcnBody([] { // (3) Body of the function
    keryx::ident i("i");
    keryx::def(keryx::sema(Name, i),
               keryx::sema(Type, keryx::_int::expr());
    keryx::context loop_body([&] { // (3) Loop body
      // Loop body
    });
    // (4) For loop construct
    keryx::vop<"for">(keryx::vop<"a=b">(i,keryx::literal(0)),
                      keryx::vop<"a<b">(i,keryx::literal(10))
                      keryx::vop<"a++">(i),
                      loop_body);
  });
  keryx::def(keryx::sema(Name, fcn), // (5) Function definition
             keryx::sema(Type, (keryx::_void::expr(), keryx::_void::expr())),
             keryx::sema(Computation, FcnBody);
}
```

Listing 6: KERYX IR example.

by the constructor of this operator. Within this context, a definition occurs when the *keryx::def* operator is invoked with a list of sema as arguments. Each sema corresponds to an invocation to the *keryx::sema* operator. In (4), a *keryx::_for* invocation is generated with invocations to the *KERYX* operators corresponding to the *C++* native operators (e.g., =, <, ++). This would generate the IR represented in (4). Finally, the definition of *fcn* in (5) shows how the IR of the definition is expressed as the nest of operator invocations. The *keryx::_void::expr()* generates the *KERYX* expression that represents the *C++ void* type within *KERYX*. In summary, every *KERYX* expression is represented as a nest of operator invocations in which the leaves of the nest can only be invocations to primitive operators that have captured a *C++* literal (e.g., a numerical or string literal or lambda expression) and formed a primitive *KERYX* expression.

Every captured lambda expression defines a *KERYX* context. Within the IR, a context is represented as a list of expressions and a symbol table in which the definitions that occurred in the context are stored. The *KERYX* IR represents all contexts with a graph-based structure with edges and nodes: nodes correspond to the contexts, and edges correspond to semantic connections that connect the contexts. Clusters of nodes define a single scope that contains the definitions in all clustered nodes. On top of this structure, lookup processes can be implemented to traverse a cluster checking for the existence of a required definition. By default, *KERYX* contexts are connected through two semantic connections: *Nested* and *Parent*. The connections represent the lexicographical nesting that contexts generate. This structure is automatically generated by the *KERYX* front end and the IR implementation.

2.9 Programming APIs

The *KERYX* runtime libraries are organized into three main components: the IR API, the device API, and the RTC API. The main aspects of the RTC API are described in the SAXPY example in Listing 1 (e.g., *keryx::compile* and *keryx::run* primitives). The device API includes the usual application requirements for device management operations: memory allocation and data transfer primitives (e.g.,

```
int main() {
  api:lift(Code);
  api::oprnds O;
  api::operands(Code, O);
  for (auto e : O) api::print(e);
  api::expr E;
  E = api::operand(1,api::arity(Code)-1);
  std::cout << "Arity of expression E = " << api::arity(E) << std::endl;
  if (api::is<"a+b">(E)) std::cout << "ADD operation!" << std::endl;
  for (auto e : O) if (api::is<api::def>(e) {
      api::print(api::get<api::Name>(e));
      api::print(api::get<api::Type>(e));
  }
  std::list<api::def> D; // API for semantic contexts
  api::lookup("fcn", Code, D);
}
```

Listing 7: Code introspection in KERYX.

alloc, *copyHtoD*, *copyDtoH*) as well as host-device synchronization primitives. The IR API is used in several examples in the next section. Generally, the simplicity of the IR translates into a simple but expressive IR API. Within the context of the examples, the computations behind each IR API primitive are implicitly understandable. All of the APIs are fully described in this paper's supplemental material.

3 KERYX DYNAMIC COMPILATION

In the context of programming for heterogeneous computing architectures, *KERYX* enables runtime kernel code transformation and optimization. This feature is thanks to its *C++* embedding and the ability of any *KERYX* program to combine IR API coding with the actual computation's description. Importantly, this type of support is not available in common programming frameworks (e.g., OpenCL/CUDA/HIP) that support RTC for heterogeneous systems. All of them rely on strings to expose the device code for their RTC support. Moreover, these systems do not provide access to their device code IR. This is an important contribution of *KERYX* because it opens up the programming for configuring the kernel code according to dynamic values that are unknown at compile time, thus leveraging dynamic compilation for adapting the code to the heterogeneous computing resources. The examples in this section use the IR API to show how to introspect and transform code.

3.1 Introspection

Listing 7 shows an example of how to use the IR API to introspect the code. For example, we use the *KERYX* code in Listing 6, which involves some definitions, expressions, and context examples. In this example, all of the *KERYX* code is contained within a *keryx::expr* and initialized with a context expression named *Code*. Listing 7 contains examples of the IR API being used to manipulate the *KERYX* expression. First, the *Code* expression is lifted, which means that the lambda that contains the *KERYX* code is executed and its IR has been built. After this step, it is possible to obtain the list of expressions contained in the *KERYX* context *Code* and print them. Also, variable *E* is initialized with one of the expressions in the *Code* context, and then the code checks on its arity. Finally, the IR API also allows for looking up a symbol defined within the *Code* context. In Listing 7, the actual *KERYX* code is not mixed with IR API code. Rather, the IR API is only used to introspect the *KERYX* code. Next, we describe how to simultaneously mix IR API code and *KERYX* code.

```
keryx::expr TileLoop(const keryx::expr& Loop,
                     const keryx::expr& BS) {
  if (not api::is<"for">(Loop)) // (1) Check loop is a for statement
    return (api::expr::undef);
  keryx::expr i = InductionVar(Loop); // (2) Check loop induction variable
  if (api::null(i)) return (api::expr::undef);
  api::oprnds O;  api::operands(Loop,O);
  api::expr initE = O[0]; // (4) Take init expr
  api::expr cmpE = O[1];  // (5) Take cmp expr
  api::expr incE = O[2];  // (6) Take inc expr
  api::expr bodyE = O[3]; // (7) Take body expr
  // (8) Check number of iterations
  keryx::expr N = NumberOfIterations(initE,cmpE,incE);
  if (api::null(N)) return (api::expr::undef);
  // (9) Build inner loop
  keryx::_for Inner(i = ii*BS, i < (ii+1)*BS, i++, api::copy(bodyE));
  keryx::expr NB = N/BS; // (10) Build number of blocks expression
  keryx::_int ii; // (11) Build outer tiled loop
  keryx::_for Outer(ii = 0, ii <NB, ii++, Inner);
  return (Outer);
}
```

Listing 8: Tiling code transformation in KERYX.

3.2 Transformation

3.2.1 Tiling. Listing 8 shows a function that tiles an input loop in the form of a *keryx::_for* statement. Lines (1) and (2) check that the actual input expression corresponds to a *keryx::_for* loop and that it is possible to extract its induction variable (the code for these steps is omitted for brevity). Then, the parts of the *keryx::_for* statement are obtained as the operands of the *keryx::_for* invocation. The code then checks for the number of iterations of the loop. Notably, the outcome of this step is a *keryx::expr* and not an actual integer number. Instead, it corresponds to a symbolic expression determined by the *NumberOfIterations* function. Finally, the tiled *keryx::_for* loop building happens in the front end: the *keryx::_for* constructs are available because they are *C++* and mixed with the actual IR API. The builders are already available within the *KERYX* framework, making it completely unnecessary to use a library with IR builders for each type of expression in the language. The same observation applies for expressions in lines (9), (10), and (11) for the actual parts of the *keryx::_for* constructs. Although simple, this example shows how the *KERYX* code is captured by its front end and can be mixed with actual IR API code to simplify the implementation of code transformations.

3.2.2 Kernel Fusion. Listings 9 and 10 show an example of kernel fusion for a set of 1D tensor operations using the IR API and the capabilities of *KERYX* to combine both IR transformers with actual *KERYX* code. Listing 9 contains three kernels, *scale*, *add*, and *relu*, and each one applies the corresponding operation to a tensor. Listing 10 shows the code that applies kernel fusion. Although simple, this example corresponds to a common and effective optimization [4, 11, 15, 27, 38–40]. In general, kernel fusion can be implemented differently depending on the context and the computation in each kernel. In this example, we have chosen an implementation that allows us to describe the *KERYX* capabilities to enable dynamic optimization: (i) combine *KERYX* code (e.g., actual IR building) with IR API code, (ii) name and identify a particular expression within the *KERYX* code, (iii) use lookup mechanisms to locate and manipulate a named expression, (iv) use IR manipulation (e.g., copy/extract/inject expressions), and (v) switch to reflective

```
keryx::kernel scale([](keryx::out<float *> result,
                       keryx::_float alpha, keryx::in<const float *> a,
                       keryx::_int n) {
    keyrx::_int idx = keryx::blockIdx.m("x") * keryx::blockDim.m("x") +
                      keryx::threadIdx.m("x");
    keryx::expr eScale;
    eScale.mSelf = ( result[idx] = alpha * a[idx] ); // (1) Reflective code
    keryx::_if ifGuard(idx < n, eScale);
    api::set<api::Name>("ComputeExpr",ifGuard); // (1) Name expression
}); // (2) Kernel scale
keryx::kernel add([](keryx::out<float *> result,
                     keryx::in<const float *> a,
                     keryx::in<const float *> b,
                     keryx::_int n) {
    keryx::_int idx = keryx::blockIdx.m("x") * keryx::blockDim.m("x") +
                      keryx::threadIdx.m("x");
    keryx::expr eAdd;
    eAdd.mSelf = ( result[idx] = a[idx] + b[idx] ); // (3) Reflective code
    keryx::_if ifGuard(idx < n, eAdd);
    api::set<api::Name>("ComputeExpr",ifGuard); // (3) Name expression
}); // (4) Kernel add
keryx::kernel relu([](keryx::out<float *> result,
                      keryx::in<const float *> a,
                      keryx::_int n) {
    keyrx::_int idx = keryx::blockIdx.m("x") * keryx::blockDim.m("x") +
                      keryx::threadIdx.m("x");
    keryx::expr eRelu;
    eRelu.mSelf = ( result[idx] = fmaxf(0.0f, a[idx]) ); // (5) Reflective code
    keryx::_if ifGuard(idx < n, eRelu));
    api::set<api::Name>("ComputeExpr",ifGuard); // (5) Name expression
}); // (6) Kernel relu
```

Listing 9: Different kernels to be fused: scale, add, and relu.

```
keryx::kernel K;
K.mSelf = fuse1D(scale, add, relu, [&] {
  keryx::_int n; keryx::_double alpha;
  keryx::var<float*> result, a, b;
  scale(result, alpha, a, n);
  add(result, result, b, n);
  relu(result, result, n);
});
keryx::kernel fuse1D(keryx::kernel A, keryx::kernel B, keryx::kernel C
                     keryx::expr E) {
  // (1) Extract and copy named expression corresponding to per-thread computation
  keryx::expr eA = api::lookup("ComputeExpr", api::get<Computation>(A));
  keryx::expr eB = api::lookup("ComputeExpr", api::get<Computation>(B));
  keryx::expr eC = api::lookup("ComputeExpr", api::get<Computation>(C));
  eA.mSelf = api::copy(eA));
  eB.mSelf = api::copy(eB));
  eC.mSelf = api::copy(eC));
  api::def f_A, f_B, f_C; // (2) Create functions to encapsulate kernel code
  f_A = api::function(eA); api::set<api::_inline>(f_A, api::expr(true));
  f_B = api::function(eB); api::set<api::_inline>(f_B, api::expr(true));
  f_C = api::function(eB); api::set<api::_inline>(f_C, api::expr(true));
  // (3) Copy expression E. Replace original kernel identifiers
  // by function identifiers.
  api::expr fused_code = api::copy(E);
  api::atcallsite<api::substitute>(api::expr(A), api::expr(f_A), fused_code);
  api::atcallsite<api::substitute>(api::expr(B), api::expr(f_B), fused_code);
  api::atcallsite<api::substitute>(api::expr(C), api::expr(f_C), fused_code);
  keryx::_int idx; // (4) Define thread identifier variable
  api::atcallsite<api::augment>(api::expr(f_A), idx, fused_code); // (5) Inject
  api::atcallsite<api::augment>(api::expr(f_B), idx, fused_code); // argument idx
  api::atcallsite<api::augment>(api::expr(f_C), idx, fused_code);
  // fused_code = {
  //    f_A(idx, result, alpha, a, n);
  //    f_B(idx, result, result, b, n);
  //    f_C(idx, result, result, n);
  // }
  keryx::expr body = [&]{ (6) Body for the fusion kernel
    api::link(idx); // (7) Link the variable definition to this context
    idx = keryx::blockIdx.m("x") * keryx::blockDim.m("x") +
          keryx::threadIdx.m("x"); // (8) Initialization of idx var
    api::inject(fused_code); // (9) Insert fused code to this context
  }
  keryx::kernel K; // (10) Kernel building
  K.mSelf = api::function(body);
  return (K);
}
```

Listing 10: Kernel fusion transformation in KERYX.

mode, where the *KERYX* front end is inhibited, and skip the code capturing.

All of these features appear in Listing 10. We start the fusion process with the invocation of *fuse1D* in the upper part of Listing 10. In (1), we use the *KERYX* features for naming expressions and performing lookup operations to locate them. Notably, in Listing 9 (1), (2), and (3), the *if-thread-guard* expression is named with a specific identifier (e.g., *"ComputeExpr"*). Note that the reflective code is associated with *mSelf* use, which makes the variables of type *keryx::expr* skip the front-end behavior and act as reflective code: the variables are assigned with the *lhs* expression instead of building IR for the assignment. In Listing 10 (1), the expressions are copied. In (2), we generate one function per expression and the corresponding expression is a function body. The role of these functions is to encapsulate the work done at thread level for each kernel. We set the *inline* attribute for these functions so that their *C++* code generation will include the corresponding attribute in their definition. In Listing 10 (3), we copy expression *E*, and replace the kernel identifiers in it with function identifiers (e.g., *f_A*,*f_B* and *f_C*). The role of expression *E* is to indicate the execution order of kernels to be fused. In (4), we define a variable to hold the *thread-id* of a kernel execution (e.g., *idx*). In (5), we add this variable to each function call in the expression *fused_code*. In (6), we build the body of the kernel resulting from the fusion, and (7) links the definition of the variable *idx* to the kernel context. In (8), the initialization statement for variable *idx* is built. In (9), we inject the code of the fused operations into the kernel body. Finally, in (10), we build the kernel. Note that in the *fuse1D* process description, we expect three kernels to be fused explicitly. This is a strategy to ease the explanations. In reality, by analyzing the expression *E*, it is possible to detect both the list of kernels and the order of fusion, making the code in procedure *fuse1D* generalized to any number of kernels.

In conclusion, mixing code that describes computation with code that introspects and transforms the code itself—all in the same programming language (*C++*)—is an important advancement for heterogeneous computing. No industrial programming paradigms for heterogeneous systems (e.g., OpenCL, CUDA, HIP, SYCL, OpenACC, OpenMP) support this feature.

4 EVALUATION

We evaluate the *KERYX* framework based on the following criteria: ease of programming, performance, portability, and adaptive compilation. Programming is evaluated in terms of the effort required to port an existing CUDA/HIP code to *KERYX*. The selected code samples [21] and the NPB-MZ benchmark suite [12, 23] belong to different domains to ensure that the evaluation captures the generality and expressiveness of the *KERYX* language. To evaluate performance, we study the overheads introduced by the *KERYX* RTC. We also compare the performance of all tested kernels in terms of total kernel execution time when statically compiled (CUDA/HIP) and dynamically compiled (*KERYX*) because static compilation and dynamic compilation rely on different compilers: one corresponds to nvcc/hipcc, and the other corresponds to the compiler support that exists in the RTC CUDA/HIP dynamic libraries. Finally, we evaluate overall performance for the NPB-MZ applications, including for multi-GPU execution. For dynamic code optimization, we study

the example described in Section 3.2 for dynamic kernel fusion to evaluate KERYX's effectiveness in terms of adaptive compilation. Finally, the evaluation also considers the portability levels enabled by the KERYX framework. Dynamic compilation has as a side benefit of the immediate portability of the device code described using the KERYX front end. The evaluation was conducted on two different heterogeneous systems with two different GPU architectures: one with two NVIDIA A100 GPUs (CUDA 12.4) and the other featuring an AMD Instinct MI300A APU (ROCm 7.0.1). We used g++ 12.2.0 to compile all *C++* sources.

4.1 Programming

This section describes the evaluation of the programming effort required to port an existing CUDA/HIP application to KERYX. We define a methodology that outlines the steps for the porting effort and then assess the complexity of each step in terms of code refactoring. We start by describing how to port a single kernel/device function. This process is performed by changing the CUDA/HIP code by following these steps:

(i) Switch from *__global__* kernel to *keryx::kernel*. From *__device__* function to *keryx::fn*.
(ii) Define a lambda and change the argument types of every parameter to KERYX native types or *keryx::in*, *keryx::out*, and *keryx::inout*.
(iii) Same as (ii) but for global variables referenced by the kernel/-function. For constant memory, replace CUDA *constant* with *keryx::constant*. Same as in (ii) but for local variables within the kernel/function. Replace the CUDA *shared* construct with the *keryx::shared* construct.
(iv) Replace *C++* compound statements (e.g., { ... }) with lambdas with *void* return type and no parameters.
(v) Replace CUDA/HIP constructs (e.g., grid, block, and thread primitives and synchronizations and memory fences) with their corresponding KERYX constructs.
(vi) Replace control flow statements (e.g., if/for/while/do-while) with their KERYX corresponding constructs.

For all tested codes, we applied this methodology to port existing CUDA/HIP code to KERYX. Notably, this porting effort did not increase the total number of source code lines for each code sample or application. The process can also be performed quickly and without the risk of changing the original code semantics.

4.2 Performance

4.2.1 RTC Overheads. The RTC of the device code occurs in different phases, as shown in Figure 2. The entries *Code Lift* and *C++ code generation* correspond to the KERYX runtimes associated with its embedded front end and *C++* back end. The other entries map to calls to the RTC runtime system that are specific to the device driver. In particular, specific primitives exist to create, compile, and obtain the output log of the compilation and generate binary device code. The chart in Figure 2 shows the weight of each phase (execution time in logarithmic scale) for each code sample executed on the NVIDIA A100. The *y*-axis shows execution time in µs.

Notably, one phase accounts for almost 95% of total overhead: *Compile program*, which corresponds to the actual invocation to the device driver compiler. The *Code Lift*, *C++ code generation*, and *Load PTX code* phases represent a very small amount of overhead. Generally, both phases do not exceed 20 ms, except for FFT2D and DXTC, which correspond to the code samples with more source code lines. All the other samples stay below 30 ms. Notice that the code samples are composed of hundreds of lines of source code (Figure 2).

When analyzing this behavior for the NPB-MZ applications (BT-MZ, SP-MZ, and LU-MZ), the introduced overhead is similar in scale. Again, the phase that correspons to the compilation of the device code is the one that accounts for almost 95% of the total overhead. KERYX's *Code Lift* and *C++ code generation* appear at a lesser order of magnitude (but now always below 30 ms because of the number of lines of source code). The NPB-MZ applications show that, even when the device code contains thousands of lines, the RTC overheads remain at the same scale as the code samples that have only a few kernels and far fewer lines of code.

The total time spent in the RTC compilation process is less than 400 ms, even in the worst cases: SP-MZ and LU-MZ executed on an NVIDIA A100 (Figure 2). Given that the compilation of the device code happens only once for the entire execution, KERYX enables RTC techniques without incurring unacceptable overheads, and the total execution time will be dominated by computations—not by RTC overheads. For brevity, we omitted the results for the AMD MI300A system because they showed very similar trends for the weight of each phase and their impact on absolute execution time.

4.2.2 Kernel Execution Time. Figure 3 shows the execution time deviations between the CUDA/HIP and KERYX versions of all kernels in the tested code samples and benchmarks. In total, 76 different kernels were evaluated. The CUDA/HIP versions are statically compiled, and the KERYX versions are runtime compiled. In the NVIDIA A100 case, we observed some kernels that execute faster in their KERYX versions and others that execute slower. In all cases, the deviations range from 0.75× to 1.16×. The average deviation expressed in terms of the geometric mean is 0.95×, as shown in left and upper side of Figure 3. In the AMD MI300A case, the trends are different. Some kernels execute faster and others slower, and the deviations range from 0.75× to 1.34×. The average deviation in terms of the geometric mean is 1.07×, meaning that, generally, the KERYX versions will be 7% faster than the HIP version. In contrast, generally, KERYX will be 5% slower than the CUDA version. We verified that the *C++* code emitted by KERYX is exactly the same as the original kernel samples. These differences are related to the static versus dynamic execution in different compilers: one corresponds to nvcc/hipcc, and the other corresponds to the compiler support in the RTC CUDA/HIP dynamic libraries. In all cases, the compiler flags are set to the same values. Currently, it is not possible to interface with the CUDA/HIP RTC support and compare which compiler passes are actually activated in static versus dynamic compilations. Because the code compiled in both the static and dynamic versions is the same, and the compilers are different, the trends described by the geometric mean in each system are created by the actual CUDA/HIP RTC support and associated deviations.

4.2.3 Overall Performance. With KERYX, the compilation of the device code is executed on demand, as happens with the device memory allocation, data transfers, kernel launch, and host-device synchronizations. The compilation of the device code is executed only once, but the device code is loaded as many times as devices

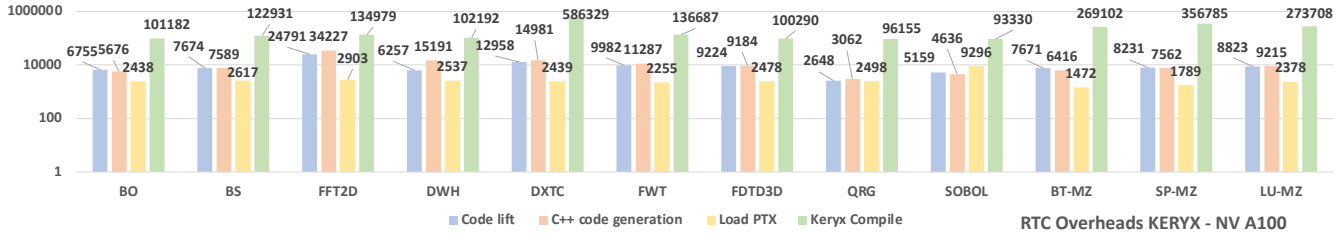


Figure 2: Phases of the KERYX dynamic compilation. Overheads of *KERYX* RTL for dynamic compilation on an NVIDIA A100. Y-axis: Runtime execution time of each phase in µs (logarithmic scale). Benchmarks from CUDA samples, Binomial Options (1 kernel, 197 scl), Black-Scholes (1 kernel, 103 scl), FFT2D (6 kernels, 457 scl), dwHaar (2 kernels, 272 scl), dxtc (1 kernel, 604 scl), Fast Walsh Transform (3 kernels, 223 scl), FDTD3D (1 kernel, 294 scl), Quasi Random Generator (2 kernels, 202 scl), and Sobol Random Generator (1 kernel, 250 scl). From NPB-MZ, BT-MZ (20 kernels, 2,262 scl), SP-MZ (15 kernels, 2,269 scl), and LU-MZ (23 kernels, 2,108 scl). Total number of kernels: 76. Total number of source code lines (scl): 9,242.

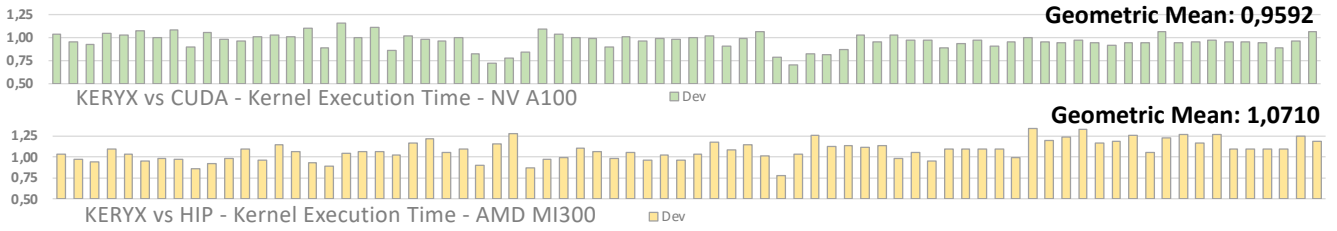


Figure 3: Kernel execution time deviation between dynamic and static compiled versions (CUDA/HIP static and KERYX dynamic compilation on NVIDIA A100 and AMD MI300A devices). Y-axis: CUDA/HIP execution time divided by KERYX execution time. Upper-left side: geometric mean values for both systems.

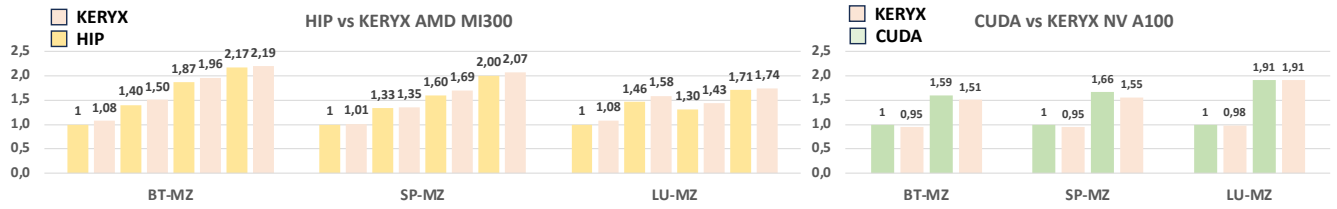


Figure 4: Overall performance (speedup over CUDA/HIP on 1 GPU) for CUDA/HIP static and KERYX dynamic compilation for NVIDIA's A100 and AMD's MI300A. Benchmarks include BT-MZ, SP-MZ and LU-MZ (CLASS C).

are active. With this in mind, the results (Section 4.2.1) show that, in general, the PTX (parallel thread execution) loading time will always be negligible compared to the total execution time of each application, even if more than one device is active. Compilation happens just once, and loading the PTX takes only a few milliseconds (see *Load ptx code* in Figure 2). For example, in the system with two NVIDIA A100 GPUs, the executions times (all running class C [23]) are as follows for one GPU and two GPUs: BT-MZ in 22.61 s and 14.17 s, SP-MZ in 18.54 s and 11.33 s, and LU-MZ in 13.05 s and 6.7 s, which indicates a larger order of magnitude than the actual RTC overheads. Figure 4 shows the speedup for the single GPU CUDA/HIP versions on the NVIDIA A100 and AMD MI300A systems for the three applications in the NPB-MZ benchmark suite. For the AMD MI300A system and the BT-MZ application, all executions with one, two, three, or four GPU(s) show that both HIP statically compiled and *KERYX* dynamically compiled perform at similar levels, but *KERYX*'s executions are faster depending on the number of active devices. Notably, *KERYX* is 1.08× faster with one GPU.

This result determines the differences between two, three, or four GPUs, where again *KERYX* is always faster. This tracks with the performance observed at the kernel level, where we see a geometric mean for kernel execution time deviations of 1.07× (Section 4.2.2). For the SP-MZ and LU-MZ applications, the results expose a similar trend: for one GPU, *KERYX* is slightly faster, which—again—agrees with the observed geometric mean on kernel execution time deviations. The executions with two, three, or four GPUs also show that the *KERYX* versions are faster. For the NVIDIA A100 system and the same applications, executions with one and two active GPU(s) show that both CUDA statically compiled and *KERYX* dynamically compiled perform at similar levels. However, in this case, *KERYX* does

execute slower. This tracks with the geometric mean of execution time deviations being 0.95× (Section 4.2.2). In conclusion, KERYX can operate with multi-GPU systems, enable and simplify RTC techniques without inducing significant overhead, and perform similarly to statically compiled versions of CUDA/HIP applications.

4.3 Dynamic Code Optimization

This section describes the implementation and validation of the kernel fusion optimization described in Section 3.2. We coded a set of tensor operations (e.g., *scale*, *add*, *sub*, *relu*) and synthesized different combinations over tensors of different sizes. Here, we compare original CUDA/HIP non-fused versions with KERYX fused versions. We also study the execution time for the actual kernel fusion process at the KERYX IR level. Figure 5 shows the speedup (y-axis) for the kernel fusion optimization described in Section 3.2 for the NVIDIA A100 system. We also test increasing the tensor sizes for different operations (x-axis) until the speedup starts decreasing. Three different kernels are studied: *Add*, *FMA*, and *Relu*. Each kernel is generated with the fusion of other tensor kernels: *Add* fuses two kernels that perform the corresponding operation on three tensors (e.g., *temp = a+b; result = temp+c;*), *FMA* computes similarly but with *multiply* and *add* operations, and *Relu* chains the *scale*, *add*, and *relu* operations. As expected, with small tensor sizes (e.g., fewer than 1 million elements [*float* type]), the benefit is observable with a speedup factor in the range of 1.75× to 2.98×. With a larger tensor, fusion impact starts decreasing, and speedup factors fall between 1.5× and 2.35×. In terms of overhead related to IR fusion, we measured the execution time of each fusion transformation at the IR API level. The actual kernel fusion can be processed in less than 0.3 ms, including lifting the IR and generating the *C++* code. This tracks with the results shown in Figure 2, where *Code Lift* and *C++ code generation* accounted for less than 15 ms (on average) for samples and benchmarks that included many more lines of source code. For the AMD MI300A system, the experiments show the same trends, but we omit those results and descriptions for brevity. In conclusion, with KERYX it is possible to implement kernel fusion within a tensor library at the IR API level.

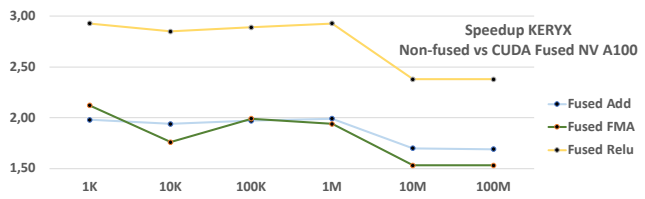


Figure 5: KERYX overall kernel fusion performance (speedup over CUDA's non-fused kernels) on an NVIDIA A100.

4.4 Portability

Portability is evaluated in terms of the programming effort required to execute the benchmarks on both tested heterogeneous systems, each one equipped with different vendor-specific programming environments (NVIDIA's CUDA and AMD's HIP). We manually generated HIP versions for all CUDA samples as well as for the NPB-MZ suite using the *hipify* [19] tool, which is available in the HIP programming framework. This step led to different sources for each version. In contrast, all KERYX versions define a single source code implementation. Moreover, because the benchmarks do not have additional dependencies other than the ones related to kernel code compilation and device management, we can run the same binary on the two tested systems with no need to recompile the benchmarks. Notably, by no means do we claim that KERYX is more portable than other common programming models (e.g., OpenCL, SYCL, OpenMP, OpenACC, Kokkos). Instead, we insist that—thanks to its dynamic compilation—KERYX enables portability on a similar level as other solutions.

5 RELATED WORK

Many frameworks such as *TACO* [24],*Tiramisu* [5], and *Halide* [35] have used operator overloading and symbolic execution to embed their domain-specific languages (DSLs) in host languages such as *C++* and Python. Intel's ArBB [29] enables runtime generation of vector-style code using a combination of operator overloading and macros in *C++*. *Terra* [10] generates and auto-tunes high-performance code for BLAS routines and stencil computations using Lua. In contrast, DSLs such as *GraphIt* [41] and *Simit* [25] take the compiler approach for two-stage execution. A specialized compiler parses all operators and control flow structures from the input program and compiles them down to low-level *C++*/CUDA code. In [28], a DSL for quantum computing is embedded in *C++* using the Clang support for parsing plugins. This approach also needs specialized compiler phases for analyses and transformations. Among the mentioned DSLs, two main aspects set KERYX apart: its embedding in *C++* and the RTC techniques that it enables. None of the previously mentioned approaches combines both.

BuildIt [7] follows a similar approach as KERYX. Classes and operators are deployed to capture the program specification, but the range of language expression that *BuildIt* can capture is much narrower than it is for KERYX. For example, the notion of semantic context is not supported, and *BuildIt* cannot capture *C++* lambda expressions and operate with them. Also, there is no methodology to extend *BuildIt* with new classes, whereas KERYX can be extended with new operators.

Legion [6] is a programming model and runtime system for heterogeneous architectures that provides abstractions for describing the structure of program data in a machine independent way. Compared to KERYX, Legion misses integration and usability with *C++*. Because KERYX is embedded in *C++*, it benefits from the many high-level features that already exist in *C++* (e.g., template coding). Also, KERYX immediately aligns with a widely used language in HPC.

Sequoia [14] provides rich memory abstractions to enable explicit control over movement and placement of data at all levels of a heterogeneous memory hierarchy. Currently, KERYX does not include these features. However, by design, KERYX can be extended to support them.

Delite [37] is a compiler framework implemented in Scala and targets high performance with back ends that emit *C++*, CUDA, or OpenCL. The Delite front end supports the deep embedding of a DSL by using Scala and exposes all the available primitives/patterns to parallelize the computations expressed with the DSL.

The *exo* [17, 18] framework explores the definition of *exo-compilation* to enable the programmer to modify the program through user-level libraries that implement schedules. In this way, *KERYX* follows a similar approach but includes the definition of an abstract IR at the user level and provides a rich IR API to implement what *exo* defines as schedules, which within *KERYX* are directly identified as code transformation and optimization.

The *PACXX* framework [16] offers similar runtime capabilities as *KERYX*. But PACXX does not allow for introspection, nor introduces an abstract IR and API to modify it. PACXX relies on an offline compilation and post JIT phases happening at runtime, implemented using the LLVM JIT support. In order to modify the device code, a programmer would have to use the LLVM representation. With KERYX, the abstract IR is much closer to the source code representation, more suitable for common specialization of device code to runtime known values.

Finally, the GSParlib [36] library enables a programmer to describe device code with string-based patterns under a systematized methodology that emulates high-level programming as a macro-based language. It does not incorporate any actual real semantic analysis nor an IR definition.

6 CONCLUSIONS

In this paper, we have described the *KERYX* framework for dynamic compilation in heterogeneous systems. We have shown that it is possible to embed *KERYX* in *C++* and use it to describe device computations. This embedding solves important limitations in state-of-the-art RTC support for CUDA/HIP. Specifically, *KERYX* solves the template code expansion, avoiding complex interactions with the device driver for name resolution of template code. Additionally, *KERYX* defines a simple IR representation that can be introspected and transformed at runtime to enable adaptive RTC. *KERYX* makes it possible to implement kernel fusion for a tensor library at the application level. The *KERYX* runtime libraries support the device management as well as the dynamic compilation of the device kernels. Altogether, *KERYX* enables both code and performance portability. By benchmarking CUDA samples and the NPB-MZ suite, we have shown how these can be executed in different heterogeneous systems with no need to recompile the source code. In doing so, *KERYX* achieved similar performance as the statically generated versions (e.g., NVIDIA's CUDA, AMD's ROCm) on two heterogeneous computing systems—one featuring two NVIDIA A100 GPUs and the other featuring an AMD MI300A APU.

ACKNOWLEDGE

This work has been supported by FEDER, UE and PID2023-147979NB-C21 funded by MICIU/AEI/10.13039/501100011033. This research used resources of the Experimental Computing Laboratory (ExCL) at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725. This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research Program, through "Competitive Portfolios" DE-FOA-0003264, under award number DE-SC0025645 and FWP ERKJ452.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: a system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, USA, 265–283.

[2] Advanced Micro Devices, Inc. 2024. *HIP Programming Guide, ROCm 6.3.* AMD Inc. https://rocm.docs.amd.com/ Accessed: 2026-01-11.

[3] Advanced Micro Devices, Inc. 2025. *AMD CDNA 3 ARCHITECTURE.* Technical Report. . https://www.amd.com/content/dam/amd/en/documents/instinct-tech-docs/white-papers/amd-cdna-3-white-paper.pdf Accessed: 2026-02-05.

[4] Arash Ashari, Shirish Tatikonda, Matthias Boehm, Berthold Reinwald, Keith Campbell, John Keenleyside, and P. Sadayappan. 2015. On optimizing machine learning workloads via kernel fusion. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (San Francisco, CA, USA) *(PPoPP 2015)*. Association for Computing Machinery, New York, NY, USA, 173–182. https://doi.org/10.1145/2688500.2688521

[5] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. 2019. Tiramisu: A Polyhedral Compiler for Expressing Fast and Portable Code. *Proc. ACM Program. Lang.* -, - (2019), 193–205.

[6] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. 2012. Legion: Expressing locality and independence with logical regions. In *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. ., ., 1–11. https://doi.org/10.1109/SC.2012.71

[7] Ajay Brahmakshatriya and Saman Amarasinghe. 2021. BuildIt: A Type-Based Multi-stage Programming Framework for Code Generation in C++. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. ., ., 39–51. https://doi.org/10.1109/CGO51591.2021.9370333

[8] Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. 2019. HetExchange: encapsulating heterogeneous CPU-GPU parallelism in JIT compiled engines. *Proc. VLDB Endow.* 12, 5 (Jan. 2019), 544–556. https://doi.org/10.14778/3303753.3303760

[9] Periklis Chrysogelos, Panagiotis Sioulas, and Anastasia Ailamaki. 2019. Hardware-conscious Query Processing in GPU-accelerated Analytical Engines. *CIDR* -, - (2019), –. http://cidrdb.org/cidr2019/papers/p127-chrysogelos-cidr19.pdf

[10] Zachary DeVito, James Hegarty, Alex Aiken, Pat Hanrahan, and Jan Vitek. 2013. Terra: A Multi-Stage Language for High-Performance Computing. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 105–116. https://doi.org/10.1145/2491956.2462166

[11] Matija Dodović, Milica Veselinović, and Marko Mišić. 2026. Analyzing the Impact of Kernel Fusion on GPU Tensor Operation Performance: A Systematic Performance Study. *Electronics* 15, 5 (2026), . https://doi.org/10.3390/electronics15051034

[12] Jörg Dümmler and Gudula Rünger. 2013. Execution Schemes for the NPB-MZ Benchmarks on Hybrid Architectures: A Comparative Study. In *Procs. of the Intl. Conf. on Parallel Computing, ParCo 2013*. ., ., 733–742. https://doi.org/10.3233/978-1-61499-381-0-733

[13] Peter Eastman, Raimondas Galvelis, Raúl P. Peláez, Charlles R. A. Abreu, Stephen E. Farr, Emilio Gallicchio, Anton Gorenko, Michael M. Henry, Frank Hu, Jing Huang, Andreas Krämer, Julien Michel, Joshua A. Mitchell, Vijay S. Pande, João PGLM Rodrigues, Jaime Rodriguez-Guerra, Andrew C. Simmonett, Sukrit Singh, Jason Swails, Philip Turner, Yuanqing Wang, Ivy Zhang, John D. Chodera, Gianni De Fabritiis, and Thomas E. Markland. 2024. OpenMM 8: Molecular Dynamics Simulation with Machine Learning Potentials. *The Journal of Physical Chemistry B* 128, 1 (2024), 109–116. https://doi.org/10.1021/acs.jpcb.3c06662 arXiv:https://doi.org/10.1021/acs.jpcb.3c06662 PMID: 38154096.

[14] Kayvon Fatahalian, Timothy J. Knight, Mike Houston, Mattan Erez, Daniel Reiter Horn, Larkhoon Leem, Ji Young Park, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. 2006. Sequoia: Programming the Memory Hierarchy. In *SC '06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*. ., ., 4–4. https://doi.org/10.1109/SC.2006.55

[15] Junji Fukuhara and Munehiro Takimoto. 2022. Automated kernel fusion for GPU based on code motion. In *Proceedings of the 23rd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems* (San Diego, CA, USA) *(LCTES 2022)*. Association for Computing Machinery, New York, NY, USA, 151–161. https://doi.org/10.1145/3519941.3535078

[16] Michael Haidl, Michel Steuwer, Tim Humernbrum, and Sergei Gorlatch. 2016. Multi-stage programming for GPUs in C++ using PACXX. In *Proceedings of the 9th Annual Workshop on General Purpose Processing Using Graphics Processing Unit* (Barcelona, Spain) *(GPGPU '16)*. Association for Computing Machinery, New York, NY, USA, 32–41. https://doi.org/10.1145/2884045.2884049

[17] Yuka Ikarashi, Gilbert Louis Bernstein, Alex Reinking, Hasan Genc, and Jonathan Ragan-Kelley. 2022. Exocompilation for productive programming of hardware accelerators. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 703–718. https://doi.org/10.1145/3519939.3523446

[18] Yuka Ikarashi, Kevin Qian, Samir Droubi, Alex Reinking, Gilbert Louis Bernstein, and Jonathan Ragan-Kelley. 2025. Exo 2: Growing a Scheduling Language. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 426–444. https://doi.org/10.1145/3669940.3707218

[19] AMD Inc. 2024. HIPIFY documentation. https://rocm.docs.amd.com/projects/HIPIFY/en/latest/ [Online; accessed 12-Nov-2024].

[20] AMD Inc. 2026. AMD RTC documentation: Lowered Names. https://rocm.docs.amd.com/projects/HIP/en/develop/how-to/hip_rtc.html#lowered-names-mangled-names [Online; accessed 24-April-2026].

[21] NVIDIA Inc. 2025. Cuda Samples. https://github.com/NVIDIA/cuda-samples/tree/master/Samples/5_Domain_Specific/ [Online; accessed 5-Feb-2025].

[22] NVIDIA Inc. 2026. NVIDIA RTC documentation: Lowered Names. https://docs.nvidia.com/cuda/nvrtc/index.html#code-lowered-name-cpp [Online; accessed 24-April-2026].

[23] Jörg Dümmler. 2023. NPB-CUDA, an implementation of the NAS Parallel Benchmarks (NPB) for Nvidia GPUs in CUDA. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf

[24] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The Tensor Algebra Compiler. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 77 (oct 2017), 29 pages. https://doi.org/10.1145/3133901

[25] Fredrik Kjolstad, Shoaib Kamil, Jonathan Ragan-Kelley, David I. W. Levin, Shinjiro Sueda, Desai Chen, Etienne Vouga, Danny M. Kaufman, Gurtej Kanwar, Wojciech Matusik, and Saman Amarasinghe. 2016. Simit: A Language for Physical Simulation. *ACM Trans. Graph.* 35, 2, Article 20 (mar 2016), 21 pages. https://doi.org/10.1145/2866569

[26] Chris Lattner, Jacques A. Pienaar, Mehdi Amini, Uday Bondhugula, River Riddle, Albert Cohen, Tatiana Shpeisman, Andy Davis, Nicolas Vasilache, and Oleksandr Zinenko. 2020. MLIR: A Compiler Infrastructure for the End of Moore's Law. *CoRR* abs/2002.11054 (2020), - pages. arXiv:2002.11054 https://arxiv.org/abs/2002.11054

[27] Ao Li, Bojian Zheng, Gennady Pekhimenko, and Fan Long. 2022. Automatic horizontal fusion for GPU kernels. In *Proceedings of the 20th IEEE/ACM International Symposium on Code Generation and Optimization* (Virtual Event, Republic of Korea) *(CGO '22)*. IEEE Press, New York, NY, USA, 14–27. https://doi.org/10.1109/CGO53902.2022.9741270

[28] Alexander J McCaskey, Dmitry I Lyakh, Eugene F Dumitrescu, Sarah S Powers, and Travis S Humble. 2020. XACC: a system-level software infrastructure for heterogeneous quantum–classical computing. *Quantum Science and Technology* 5, 2 (feb 2020), 024002. https://doi.org/10.1088/2058-9565/ab6bf6

[29] Chris J. Newburn, Byoungro So, Zhenying Liu, Michael McCool, Anwar Ghuloum, Stefanus Du Toit, Zhi Gang Wang, Zhao Hui Du, Yongjian Chen, Gansha Wu, Peng Guo, Zhanglin Liu, and Dan Zhang. 2011. Intel's Array Building Blocks: A retargetable, dynamic compiler and embedded language. In *International Symposium on Code Generation and Optimization (CGO 2011)*. ., ., 224–235. https://doi.org/10.1109/CGO.2011.5764690

[30] NVIDIA Corporation. 2025. *CUDA C++ Programming Guide, Release 13.0.* NVIDIA Inc. https://docs.nvidia.com/cuda/ Accessed: 2026-01-11.

[31] NVIDIA Corporation. 2025. *NVRTC: CUDA Runtime Compilation Library Technical Report.* Technical Report. NVIDIA Research.

[32] NVIDIA, Inc. 2020. *NVIDIA A100 Tensor Core GPU Architecture.* Technical Report. . https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf Accessed: 2026-02-05.

[33] NVIDIA, Inc. 2026. *NVIDIA jitify.* NVIDIA Inc. https://github.com/NVIDIA/jitify Accessed: 2026-02-05.

[34] S. Paul, J. Giceva, and T. Neumann. 2021. Improving Execution Efficiency of Just-in-time Compilation based Query Processing on GPUs. *Proceedings of the VLDB Endowment* 14, 9 (2021), 1469–1481. https://doi.org/10.14778/3461535.3461537

[35] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. *SIGPLAN Not.* 48, 6 (jun 2013), 519–530. https://doi.org/10.1145/2499370.2462176

[36] Dinei A. Rockenbach, Gabriell Araujo, Dalvan Griebler, and Luiz Gustavo Fernandes. 2025. GSParLib: A multi-level programming interface unifying OpenCL and CUDA for expressing stream and data parallelism. *Comput. Stand. Interfaces* 92, C (March 2025), 15 pages. https://doi.org/10.1016/j.csi.2024.103922

[37] Arvind K. Sujeeth, Kevin J. Brown, Hyoukjoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. 2014. Delite: A Compiler Architecture for Performance-Oriented Embedded Domain-Specific Languages. *ACM Trans. Embed. Comput. Syst.* 13, 4s, Article 134 (apr 2014), 25 pages. https://doi.org/10.1145/2584665

[38] Wei Sun, Ang Li, Sander Stuijk, and Henk Corporaal. 2024. How Much Can We Gain From Tensor Kernel Fusion on GPUs? *IEEE Access* 12 (2024), 126135–126144. https://doi.org/10.1109/ACCESS.2024.3411473

[39] Rohan Yadav, Shiv Sundram, Wonchan Lee, Michael Garland, Michael Bauer, Alex Aiken, and Fredrik Kjolstad. 2025. Composing Distributed Computations Through Task and Kernel Fusion. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 182–197. https://doi.org/10.1145/3669940.3707216

[40] Chenchen Zhang, Hao Luo, and Chao Yang. 2025. Leonid: Exploring Automated Kernel Fusion in Performance-Portable Programming Models for Scientific Computation. In *Proceedings of the 39th ACM International Conference on Supercomputing (ICS '25)*. Association for Computing Machinery, New York, NY, USA, 928–942. https://doi.org/10.1145/3721145.3728489

[41] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. 2018. GraphIt: A High-Performance Graph DSL. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 121 (oct 2018), 30 pages. https://doi.org/10.1145/3276491