SafeGPU: Contract- and library-based GPGPU for object-oriented languages

Alexey KOLESNICHENKO

Christopher M. POSKITT
Singapore Management University, cposkitt@smu.edu.sg

Sebastian NANZ

SafeGPU: Contract- and Library-Based GPGPU for Object-Oriented Languages

Alexey Kolesnichenko[a,*], Christopher M. Poskitt[a,b,**], Sebastian Nanz[a]

[a]*Department of Computer Science, ETH Zürich, Switzerland*
[b]*Singapore University of Technology and Design, Singapore*

Abstract

Using GPUs as general-purpose processors has revolutionized parallel computing by providing, for a large and growing set of algorithms, massive data-parallelization on desktop machines. An obstacle to their widespread adoption, however, is the difficulty of programming them and the low-level control of the hardware required to achieve good performance. This paper proposes a programming approach, SafeGPU, that aims to make GPU data-parallel operations accessible through high-level libraries for object-oriented languages, while maintaining the performance benefits of lower-level code. The approach provides data-parallel operations for collections that can be chained and combined to express compound computations, with data synchronization and device management all handled automatically. It also integrates the design-by-contract methodology, which increases confidence in functional program correctness by embedding executable specifications into the program text. We present a prototype of SafeGPU for Eiffel, and show that it leads to modular and concise code that is accessible for GPGPU non-experts, while still providing performance comparable with that of hand-written CUDA code. We also describe our first steps towards porting it to C#, highlighting some challenges, solutions, and insights for implementing the approach in different managed languages. Finally, we show that runtime contract-checking becomes feasible in SafeGPU, as the contracts can be executed on the GPU.

Keywords: GPGPU, parallel computing, runtime code generation, generative programming, object-orientation, managed languages, design-by-contract, program correctness

1. Introduction

Graphics Processing Units (GPUs) are being increasingly leveraged as sources of inexpensive parallel-processing power, with application areas as diverse as scientific data analysis, cryptography, and evolutionary computing [1, 2]. Consisting of thousands of cores, GPUs are throughput-oriented devices that are especially well-suited for realizing data-parallel algorithms—algorithms performing the same tasks on multiple items of data—with potentially significant performance gains to be achieved.

[*]Corresponding author
[**]Research done while employed by the Chair of Software Engineering, ETH Zürich.
Email addresses: alexey.kolesnichenko@inf.ethz.ch (Alexey Kolesnichenko), chris_poskitt@sutd.edu.sg (Christopher M. Poskitt), sebastian.nanz@inf.ethz.ch (Sebastian Nanz)

The CUDA [3] and OpenCL [4] languages support the programming of GPUs for applications beyond graphics in an approach now known as General-Purpose computing on GPUs (GPGPU). They provide programmers with fine-grained control over hardware at the C++ level of abstraction. This control, however, is a double-edged sword: while it facilitates advanced, hardware-specific fine-tuning techniques, it does so at the cost of working within very restrictive and low-level programming models. Recursion, for example, is among the standard programming concepts prohibited. Furthermore, dynamic memory management is completely absent, meaning that programmers themselves must explicitly manage the allocation of memory and the movement of data. While possibly acceptable for experienced GPU programmers, these issues pose a significant difficulty to others, and are an obstacle to more widespread adoption.

Such challenges have not gone unnoticed: there has been a plethora of attempts to reduce the burden on programmers. Several algorithmic skeleton frameworks for C++ have been extended—or purpose built—to support the orchestration of GPU computations, expressed in terms of programming patterns that leave the parallelism implicit [5, 6, 7, 8, 9]. Furthermore, higher-level languages have seen new libraries, extensions, and compilers that allow for GPU programming at more comprehensible levels of abstraction, with various degrees of automatic device and memory management [10, 11, 12, 13, 14, 15].

These advances have made strides in the right direction, but the burden on the programmer can be lifted further still. Some approaches (e.g., [14]) still necessitate an understanding of relatively low-level GPU concepts such as barrier-based synchronization between threads; a mechanism that can lead easily to perplexing concurrency faults such as data races or barrier divergence. Such concepts can stifle the productivity of programmers and remain an obstacle to broadening the adoption of GPGPU. Other approaches (e.g., [10]) successfully abstract away from them, but require programmers to migrate to dedicated languages. Furthermore, to our knowledge, no existing approach has explored the possibility of integrating mechanisms or methodologies for specifying and monitoring the correctness of high-level GPU code, missing an opportunity to support the development of reliable programs. Our work has been motivated by the challenge of addressing these issues within a high-level language without depriving programmers of the potential performance boosts for data-parallel problems [16].

This paper proposes SafeGPU, a programming approach that aims to make GPGPU accessible through high-level libraries for object-oriented languages, while maintaining the performance benefits of lower-level code. Our approach aims for users to focus entirely on functionality: programmers are provided with primitive data-parallel operations (e.g., `sum`, `max`, `map`) for collections that can be combined to express complex computations, with data synchronization and low-level device management all handled automatically. We present a prototype of SafeGPU for Eiffel [17], built upon a new binding with CUDA, and show that it leads to modular and concise code that is accessible for GPGPU non-experts, as well as providing performance comparable with that of hand-written CUDA code. This is achieved by deferring the generation of CUDA kernels such that the execution of pending operations can be optimized by combining them. We also present a report of our first steps towards porting SafeGPU to C#, highlighting some challenges, solutions, and insights for implementing the approach in different managed languages.

Furthermore, to support the development of safe and functionally correct GPU code, we integrate the design-by-contract [18] methodology that is native to Eiffel (and provided by the Code Contracts library [19] for C#). In particular, SafeGPU supports the annotation of high-level GPU programs with executable preconditions, postconditions, and invariants, together specifying the properties that should hold before and after the execution of methods. In languages supporting design-by-contract, such annotations can be checked dynamically at runtime, but the significant overhead incurred means that they are often disabled outside of debugging. With SafeGPU, contracts can be constructed from the data-parallel primitives, allowing for them to be monitored at runtime with little overhead by executing them on the GPU.

The contribution of this work is thus an approach and library for GPU programming that:

- embraces the *object-oriented paradigm*, shielding programmers from the low-level requirements of the CUDA model without depriving them of the performance benefits;
- is *modular* and *efficient*, supporting the programming of compound computations through the composition of primitive operations with a dedicated kernel optimization strategy;
- supports the writing of *safe* and *functionally correct code* via contracts, monitored at runtime with little overhead.

This is a revised and extended version of our GPCE '15 paper [20], with the following new content:

- first steps towards porting SafeGPU to a second language, C#, accompanied by additional examples;
- the integration of our C# port with Code Contracts, a library-based contract framework (in contrast to the natively supported contracts of Eiffel);
- initial support for transferring class-based data (i.e., beyond primitive data) to the GPU;
- a new section on implementing SafeGPU in managed languages, focusing on data transfer, translating customized program logic, and obtaining deterministic memory management;
- an expanded API section, with additional operation tables for vectors and matrices;
- an expanded discussion on future work, in particular, our plans to support multi-GPU systems.

The rest of the paper is organized as follows. Section 2 provides an overview of the SafeGPU approach, its capabilities, and how it is implemented. Section 3 explores the CUDA binding and library APIs in more detail. Section 4 describes how design-by-contract is integrated. Section 5 discusses in detail key aspects and challenges of the prototype implementations. Section 6 presents our kernel generation and optimization strategies. Section 7 evaluates performance, code size, and contract checking across a selection of benchmark programs. Section 8 reviews some related work. In Section 9, we conclude, and propose some future work.

2. Overview of the SafeGPU Approach

In this section we provide an overview of the programming style supported by SafeGPU, present a simple example, and explain how the integration with CUDA is achieved (see Section 5 for an extended discussion on implementation issues).

2.1. Programming Style

CUDA kernels—the functions that run on the GPU—are executed by an array of threads, with each thread executing the same code on different data. Many computational tasks fit to this execution model very naturally (e.g., matrix multiplication, vector addition). Many tasks, however, do not, and can only be realized with non-trivial reductions. This difficulty increases when writing complex, multistage algorithms: combining subtasks into a larger kernel can be challenging, and there is little support for modularity.

In contrast, SafeGPU emphasizes the development of GPU programs in terms of simple, compositional "building blocks." For a selection of common data structures including collections, vectors, and matrices, the library provides sets of built-in primitive operations. While individually these operations are simple and intuitive to grasp (e.g., `sum`, `max`, `map`), they can also be combined and chained together to generate complex GPU computations, without the developer ever needing to think about the manipulation of kernels. The aim is to allow for developers to focus entirely on functionality, with the library itself responsible for generating kernels and applying optimizations (e.g., combining them). This focus on functionality extends to correctness, with SafeGPU supporting the annotation of programs with contracts that can be monitored efficiently at runtime.

Before we expand on these different aspects of the library, consider the simple example for Eiffel in Listing 1, which illustrates how a SafeGPU program can be constructed in practice.

```
matrix_transpose_vector_mult (matrix: G_MATRIX[DOUBLE]; vector: G_VECTOR[DOUBLE]): G_MATRIX
    [DOUBLE]
  require
    matrix.rows = vector.count
  do
    Result := matrix.transpose.right_multiply (vector)
  ensure
    Result.rows = matrix.columns
    Result.columns = 1
  end
```

Listing 1: Transposed matrix-vector multiplication in SafeGPU for Eiffel

The method takes as input a matrix and a vector, then returns the result of transposing the matrix and multiplying the vector. The computation is expressed in one line through the chaining of two compact, primitive operations from the API for matrices—`transpose` and `right_multiply` —from which the CUDA code is automatically generated. Furthermore, because the latter of the operations is only defined for inputs of certain sizes ($N \times M$ matrix; M dimension vector), the method is annotated with a precondition in the **`require`** clause, expressing that the size of the input vector should be equal to the number of rows in the matrix (rows, not columns, since it will be transposed). Similarly, the postcondition in the **`ensure`** clause expresses the expected dimensions of the resulting matrix. Both of these properties can be monitored at runtime, with the precondition checked upon entering the method, and the postcondition checked upon exiting.

2.2. CUDA Integration

SafeGPU provides two conceptual levels of integration with CUDA: a binding and a library level. The binding level provides a minimalistic API to run raw CUDA code within the high-level language, similar to bindings like PyCUDA [21] and JCUDA [22], and is intended for experienced users who need more fine-grained control over the GPU. The library level is built on top of the binding, and provides the data structures, primitive operations, contracts, and kernel-generation facilities that form the focus of this paper.

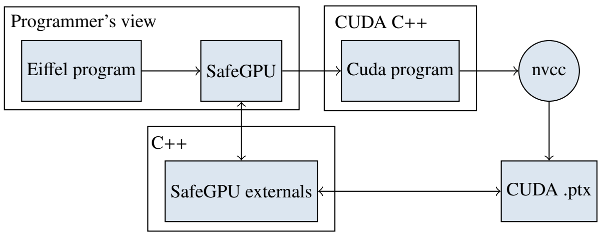


Figure 1: Runtime integration of CUDA with SafeGPU for Eiffel

In Eiffel, the runtime integration with CUDA (Figure 1) is achieved using the language's built-in mechanisms for interfacing with C++, allowing it to call the CUDA-specific functions it needs for initialization, data transfers, and device synchronization. These steps are handled automatically by SafeGPU for both the binding and library levels, minimizing the amount of boilerplate code. Given a source kernel, whether handwritten at the binding level or generated from the library one, the `nvcc` compiler generates a `.ptx` file containing a CUDA module that the library can then use to launch the kernel.

In languages such as C#, which have existing CUDA bindings, the integration of SafeGPU is simplified as the kernels the library generates can be passed to those bindings directly.

3. Design of the API

In the following, we describe in more detail the two levels of SafeGPU's API. First, we consider the binding level, which allows expert users to run CUDA code from within an object-oriented language. Then we turn to the library level, and in particular, its three basic classes for collections, vectors, and matrices. The operations provided by the API are described in the context of our principal implementation for Eiffel.

3.1. CUDA Binding

The binding API provides handles to access the GPU and raw memory. Programming with this API requires effort comparable to plain CUDA solutions, so it is therefore not a user-level API; its main purpose is to provide functionality for the library API built on top of it.

Table 1 provides details about the API's classes. The two main classes are `CUDA_KERNEL` and `CUDA_DATA_HANDLE`. The former encapsulates a CUDA kernel; the latter represents a contiguous sequence of uniform objects, e.g., a single-dimensional array.

3.2. Collections

Collections are the most abstract container type provided by SafeGPU: the majority of bulk operations—operating on an entire collection—are defined here. Collections are array-based, i.e., they have bounded capacity and count, and their items can be accessed by index. Collections do not automatically resize, but new ones with different sizes can be created using the methods of the class.

The key methods of the collection API are given in Table 2 and described in the following. Note that in Eiffel, **`like Current`** denotes the type of the current object.

Table 1: Binding API

class	description
`CUDA_DATA_HANDLE`	Represents a handle to a device memory location. Supports scalar, vector, and multi-dimensional data. Can be created from (and converted to) standard `ARRAY`s.
`CUDA_INTEROP`	Encapsulates low-level device operations, such as initialization, memory allocation, and data transfer.
`CUDA_KERNEL`	Represents a CUDA kernel, ready for execution. Can contain an arbitrary number of `CUDA_DATA_HANDLE` kernel inputs, one of which is used as output. Can be launched with configurable shared memory.
`LAUNCH_PARAMS`	Encapsulates the grid setup and shared memory size required to launch a `CUDA_KERNEL`.
`KERNEL_LOADER`	Responsible for loading CUDA kernels into the calling process. If necessary, performs a kernel compilation. Can load kernels from a file or from a string.

A SafeGPU collection can be created using the method `from_array`, which creates its content from that of an Eiffel array: as an array's content is contiguous, a single call to CUDA's analogue of `memcpy` suffices. Individual elements of the collection can then be accessed through the method `item`, and the total number of elements is returned by `count`. The method `concatenate` is used to join the elements of two containers and the method `subset` resizes a given collection to a subset.

The core part of the API design consists of methods for transforming, filtering, and querying collections. All of these methods make use of Eiffel's functional capabilities in the form of *agents*, which represent operations that are applied in different ways to all the elements of a collection (in the C# port, we use *delegates*—see Section 5). Agents can be one of three types: *procedures*, which express transformations to be applied to elements (but do not return results); *functions*, which return results for elements (but unlike procedures, are side-effect free); or *predicates*, which are Boolean expressions.

To construct a new collection from an existing one, the API provides the transformation methods `for_each` and `map`. The former applies a procedure agent to each element of the collection, whereas the latter applies a function agent. For example, the call

```
c.for_each (agent (a: INT) do a := a * 2 end)
```

represents an application of `for_each` to an integer collection `c`, customized with a procedure that doubles every element. In contrast, the call

```
c.map (agent (a: INT): DOUBLE do Result := sqrt (a) end)
```

creates from an integer collection `c` a collection of doubles, with each element the square root of the corresponding element in `c`.

To filter or query a collection, the API provides the methods `filter`, `for_all`, and `exists`,

Table 2: Collection API

`from_array (array: ARRAY[T])`	Creates an instance of a collection, containing items from the standard Eiffel array provided as input.
`item (i: INT): T`	Access to a single element.
`count: INT`	Queries the number of elements in the collection.
`concatenate (other: like Current): like Current`	Creates a new container consisting of the elements in the current object followed by those in `other`.
`subset (start, finish: INT): like Current`	Creates a subset of the collection that shares the same memory as the original.
`for_each (action: PROCEDURE[T]): like Current`	Applies the provided procedure to every element of the collection.
`map (transform: FUNCTION[T, U]): COLLECTION[U]`	Performs a projection operation on the collection: each element is transformed according to the specified function.
`filter (condition: PREDICATE[T]): like Current`	Creates a new collection containing only items for which the specified predicate holds.
`for_all (condition: PREDICATE[T]): BOOLEAN`	Checks whether the specified predicate holds for all items in the collection.
`exists (condition: PREDICATE[T]): BOOLEAN`	Checks whether the specified predicate holds for at least one item in the collection.
`new_cursor: ITERATION_CURSOR [T]`	Implementation of `ITERABLE[T]`; called upon an iteration over the collection.
`update`	Forces execution of all pending operations associated with the current collection. The execution is optimized whenever possible.

which evaluate predicate agents with respect to every element. An example of filtering is

```
c.filter (agent (a: INT) do Result := a < 5 end)
```

which creates a new collection from an integer collection `c`, containing only the elements that are less than five. The method `for_all` on the other hand does not create a new collection, but rather checks whether a given predicate holds for every element or not; the call

```
c.for_all (agent (i: T) do Result := pred (i) end)
```

returns **`True`**, for example, if some (unspecified) predicate `pred` holds for every element of the collection `c` (and **`False`** otherwise). The method `exists` is similar, returning **`True`** if the predicate holds for at least one element in the collection (and **`False`** otherwise).

The queries `for_all` and `exists` are particularly useful in contracts, and can be parallelized effectively for execution on the GPU. We discuss this use further in Section 4.

Collections are embedded into Eiffel's container hierarchy by implementing the `ITERABLE` interface, which allows the enumeration of their elements in foreach-style loops (**`across`** in Eiffel terminology). Enumerating is efficient: upon a call to `new_cursor`, the collection's content is copied back to main memory in a single action.

Finally, the special method `update` forces execution of any pending kernel operations (see Section 6).

3.3. Vectors

Vectors are a natural specialization of collections. Besides the inherited operations of collections, they provide a range of numerical operations.

The API for vectors is presented in Table 3. It allows for computing the average value `avg` and `sum` of the elements of arbitrary vectors, as well as computing the minimum `min` and maximum `max` elements. Furthermore, `is_sorted` will check whether the elements are sorted. These functions are all implemented by multiple reductions on the device side (we remark that these computations via reduction do not do more work than their corresponding sequential computations).

Table 3: Vector API: vector-only operations

`sum: T`	Computes the sum of the vector elements.
`min: T`	Computes the minimum of the vector elements.
`max: T`	Computes the maximum of the vector elements.
`avg: T`	Computes the average of the vector elements.
`is_sorted: BOOLEAN`	Checks whether the vector is sorted.
`plus (other: VECTOR[T]): VECTOR[T]`	Adds the provided vector to the current vector and returns the result.
`minus (other: VECTOR[T]): VECTOR[T]`	Subtracts the provided vector from the current vector and returns the result.
`in_place_plus (other: VECTOR[T])`	Adds the provided vector to the current vector and modifies the current vector to contain the result.
`in_place_minus (other: VECTOR[T])`	Subtracts the provided vector from the current and modifies the current vector to contain the result.
`multiplied_by (factor: T): VECTOR[T]`	Multiplies the current vector by the provided scalar.
`divided_by (factor: T): VECTOR[T]`	Divides the current vector by the provided scalar. The scalar should not be zero.
`compwise_multiply (other: VECTOR[T]): VECTOR[T]`	Multiplies the current vector by another component-wise.
`compwise_divide (other: VECTOR[T]): VECTOR[T]`	Divides the current vector by another component-wise. No zero elements are allowed in the second vector.

All the numerical operations such as `plus` and `minus` (alongside their in-place variants), as well as `multiplied_by` and `divided_by` (alongside their component-wise variants) are defined as vector operations on the GPU, e.g., a call to `plus` performs vector addition in a single action on the device side. Note that aliases can be used for value-returning operations, e.g., `v * n` instead of `v.multiplied_by (n)`.

An important requirement in using and composing vector operations is keeping the dimensions of the data synchronized. Furthermore, certain arithmetic operations are undefined on certain elements; `divided_by`, for example, requires that elements are non-zero. Such issues are managed through contracts built-in to the API that can be monitored at runtime, shielding developers from inconsistencies. We discuss this further in Section 4.

3.4. Matrices

The matrix API is strongly tied to the vector API: the class uses vectors to represent rows and columns. On the device side, a matrix is stored as a single-dimensional array with row-wise alignment. Thus, a vector handle for a row can be created by adjusting the corresponding indices. The column access pattern is more complicated, and is implemented by performing a copy of corresponding elements into new storage.

The matrix-only methods of the API are given in Table 4. Table 5 provides the specialized operations inherited from the collection API, and describes how they are tailored to matrices.

In the API, the queries `rows` and `columns` return the dimensions of the matrix, whereas `item`, `row`, and `column` return the part of the matrix specified. Single-column or single-row matrices can be converted to vectors simply by making the appropriate call to `row` or `column`.

Similar to vectors, the API provides both conventional and in-place methods for addition and subtraction. Beyond these primitive arithmetic operations, the API provides built-in support for matrix-matrix multiplication (method `multiply`) since it is a frequently occurring operation in GPGPU. The implementation optimizes performance through use of the shared device memory.

Also supported are left and right matrix-vector multiplication (respectively `left_multiply` and `right_multiply`), component-wise matrix multiplication and division (`compwise_multiply` and `compwise_divide`), matrix transposition (`transpose`), and `submatrix` creation.

Like the other API classes, matrix methods are equipped with contracts in order to shield the programmer from common errors, e.g., mismatching dimensions in matrix multiplication.

4. Design-by-Contract Integration

To support the development of safe and functionally correct code, SafeGPU integrates the design-by-contract methodology [18], i.e., the annotation of methods with executable pre- and postconditions, expressing precisely the properties that should hold upon entry and exit. These can be monitored at runtime to help ensure the correctness of programs. In the context of GPU programs, in which very large amounts of data might be processed, "classical" (i.e., sequential) contracts take so long to evaluate that they need to be disabled outside of debugging. With SafeGPU, however, contracts can be expressed using the primitive operations of the library itself, and thus can be executed on the GPU—where the data is sitting—without diminishing the performance of the program (see our benchmarks in Section 7.3).

Contracts are supported by several object-oriented languages. Our principal implementation of SafeGPU for Eiffel takes advantage of the fact that the specification and runtime checking of contracts is supported natively by the language. For our port to C#, contracts are instead supported via a library—Code Contracts [19]—which provides a number of advanced specification features including contracts for interfaces, abstract base classes, inheritance, and methods with

Table 4: Matrix API: matrix-only operations

`rows: INT`
: Queries the total number of rows in the matrix.

`columns: INT`
: Queries the total number of columns in the matrix.

`row (i: INT): VECTOR[T]`
: Queries a live view of the elements in i-th row of the current matrix. Changes in the view will affect the original matrix.

`column (j: INT): VECTOR[T]`
: Queries a live view of the elements in j-th column of the current matrix. Changes in the view will affect the original matrix.

`multiply (matrix: MATRIX[T]): MATRIX[T]`
: Performs matrix-matrix multiplication between the current matrix and the provided one. Creates a new matrix to store the result.

`left_multiply (vector: VECTOR[T]): MATRIX[T]`
: Multiplies the provided row-vector with the current matrix.

`right_multiply (vector: VECTOR[T]): MATRIX[T]`
: Multiplies the current matrix with the provided column-vector.

`transpose: MATRIX[T]`
: Returns a transposition of the current matrix. Creates a new matrix to store the result. An in-place version is also available.

`compwise_multiply (scalar: T): MATRIX[T]`
: Multiplies each element in the matrix by the provided scalar. Creates a new matrix to store the result. An in-place version is also available.

`compwise_divide (scalar: T): MATRIX[T]`
: Divides each element in the matrix by the provided scalar. Creates a new matrix to store the result. An in-place version is also available.

`submatrix (start_row, start_column, end_row, end_column: INTEGER): MATRIX[T]`
: Creates a submatrix, starting at (start_row, start_column) and ending at (end_row, end_column). A new matrix is created to store the result.

Table 5: Matrix API: specialized collection operations

`from_array (array: ARRAY[T]; rows, columns: INTEGER)`	Creates an instance of a matrix, containing items from the standard Eiffel array provided as input. The number of rows and columns is specified in the constructor.
`item (i, j: INT): T`	Access to a single element in a matrix.
`count: INT`	Queries the total number of elements in the matrix.
`for_each (action: PROCEDURE[T]): like Current`	Applies the provided procedure to every element of the matrix.
`map (transform: FUNCTION[T, U]): MATRIX[U]`	Performs a projection operation on the matrix: each element is transformed according to the specified function.
`filter (condition: PREDICATE[T]): like Current`	Creates a new matrix containing only items for which the specified predicate holds.
`for_all (condition: PREDICATE[T]): BOOLEAN`	Checks whether the specified predicate holds for all items in the matrix.
`exists (condition: PREDICATE[T]): BOOLEAN`	Checks whether the specified predicate holds for at least one item in the matrix.
`new_cursor: ITERATION_CURSOR [T]`	Implementation of `ITERABLE[T]`; called upon an iteration over the matrix. The iteration is row-wise.
`update`	Forces execution of all pending operations associated with the current matrix. The execution is optimized whenever possible.

multiple return statements (which is forbidden in Eiffel). Most importantly for SafeGPU, the library also provides runtime contract checking via a post-compilation step. Similar contract libraries exist for other object-oriented languages, e.g., JML for Java [23].

4.1. Contracts in SafeGPU

Contracts are utilized by SafeGPU programs in two ways. First, they are built-in to the library API; several of its methods are equipped with pre- and postconditions, providing correctness properties that can be monitored at runtime "for free" (i.e., without requiring additional user annotations). Second, when composing the methods of the API to generate more complex, compound computations, users can define and thus monitor their own contracts expressing the intended effects of the overall computation.

The API's built-in contracts are motivated easily by vector and matrix mathematics, for which several operations are undefined on input with inconsistent dimensions or input containing zeroes. Consider for example Listing 2, which contains the signature and contracts of the library method for component-wise vector division. Calling `v1.compwise_divide (v2)` on vectors `v1` and `v2` of equal size results in a new vector, constructed from `v1` by dividing its elements by the corresponding elements in `v2`. The preconditions in the **`require`** clause assert that the vectors are of equal size (via `count`, from the collection API) and that all elements of the second vector are non-zero (via `for_all`, customized with a predicate agent). The postcondition in the **`ensure`** clause characterizes the effect of the method by asserting the expected relationship between the resulting vector and the input (retrieved using the **old** keyword).

```
compwise_divide (other: VECTOR[T]): VECTOR[T]
  require
    other.count = count
    other.for_all(
      agent (el: T) do Result := el /= {T}.zero end)
  ensure
    Current = old Current
    Result * other = Current
  end
```

Listing 2: Contracts for component-wise vector division in SafeGPU for Eiffel

Built-in and user-defined contracts for GPU collections are typically classified as one of two types. *Scalar contracts* are those using methods with execution times independent of the collection size. A common example is `count`, which records the number of elements a collection contains. *Range contracts* are those using methods that operate on the elements of a collection and thus have execution times that grow with the collection size. These include library methods such as `sum`, `min`, `max`, and `is_sorted`. The CUDA programs generated for such operations usually perform multiple reductions on the GPU. Other common range contracts are those built from `for_all` and `exists`, equipped with predicate agents, expressing properties that should hold for every (resp. at least one) element of a collection. These are easily parallelized for execution on the GPU, and unlike their sequential counterparts, can be monitored at runtime for very large volumes of data without diminishing the overall performance of the program (see Section 7.3).

The Eiffel implementation of SafeGPU provides a straightforward way to monitor user-defined contracts on the GPU: simply express them in the native **require** and **ensure** clauses of methods, using the primitive operations of the library. This is analogous to classical design-by-contract, in which methods are used in both specifications and implementations.

The C# port requires contracts to be expressed via library calls—`Contract.Requires` and `Contract.Ensures`—rather than in native clauses. It is important that the preconditions are called at the beginning of the method body (since they are executed as normal function calls), but for postconditions, the binary is rewritten to ensure they are executed at the exit point(s) of the body, hence they can be expressed anywhere in the method. It is conventional however to list them immediately after the preconditions.

4.2. Example: Quicksort in SafeGPU

In the following, we will consider SafeGPU implementations of quicksort (in both Eiffel and C#), since the example demonstrates built-in and user-defined contracts, as well as scalar and range contracts.

Listing 3 contains the implementation and contracts of `quicksort` in Eiffel SafeGPU. The implementation utilizes two methods provided by the collection API: `concatenate`, to efficiently concatenate two vectors; and `filter`, to find items less than, greater than, or equal to the pivot. The three calls to `filter` are customized with predicate agents expressing these relations. We remark that since inline agents cannot access local variables in Eiffel, the `pivot` is passed as an argument. This is denoted by `(?, pivot)` at the end of each agent expression: here, the `?` corresponds to `item`, expressing that it should be instantiated with successive elements of the collection; `pivot` corresponds to `a_pivot`, expressing that the latter should always take the value of the former. At runtime, the built-in contracts of these two library methods can be monitored,

```
quicksort (a: G_VECTOR[REAL_32]): G_VECTOR[REAL_32]
  require
    a.count > 0
  local
    pivot: DOUBLE
    left, mid, right: G_VECTOR[REAL_32]
  do
    if (a.count = 1) then
      Result := a
    else
      pivot := a[a.count // 2]

      left := a.filter (agent (item: REAL_32; a_pivot: REAL_32): BOOLEAN do Result := item
           < a_pivot end (?, pivot))
      right := a.filter (agent (item: REAL_32; a_pivot: REAL_32): BOOLEAN do Result := item
            > a_pivot end (?, pivot))
      mid := a.filter (agent (item: REAL_32; a_pivot: REAL_32): BOOLEAN do Result := item =
            a_pivot end (?, pivot))

      Result := quicksort (left).concatenate (mid).concatenate (quicksort (right))
    end
  ensure
    Result.is_sorted
    Result.count = a.count
  end
```

Listing 3: Quicksort in SafeGPU for Eiffel

but they only express correctness conditions localized to their use, and nothing about their compound effects. The overall postcondition of the computation can be expressed as a user-defined postcondition of `quicksort`, here asserting—using the `is_sorted` and `count` methods of the vector API—that the resulting vector is sorted and of the same size. This can be monitored at runtime to increase confidence that the user-defined computation is correct.

Listing 4 contains the implementation and contracts (as library calls) of `Quicksort` in the C# port of SafeGPU. Note that this implementation is more general in that it uses collections instead of vectors, and can work with any `struct` in which values can be compared and translated by SafeGPU (see Section 5.2). Note also that we use delegates, the C# counterpart to Eiffel's agents, as well as lambda expressions to create these delegates in-place (since lambda expressions in C# are allowed to access local variables, the syntax is slightly more compact). As the program operates on generic collections, we have to provide a comparison function to `IsSorted` so that it is able to compare two arbitrary objects in a collection. Again, this is achieved by using lambda expressions to create a delegate in-place.

5. Implementation of SafeGPU

In this section, we discuss three of the most important issues for implementing the SafeGPU approach in a managed language. First, how primitive and class-based data can be transferred to the GPU. Second, how customized, functional computations can be expressed and supported. Finally, how to achieve deterministic memory management in the presence of garbage collection.

These issues are discussed in the both the context of our principal SafeGPU implementation for Eiffel as well as our initial port of the library for C#, in the hope that comparing the challenges and solutions between them provides some general insights for implementing the library in other managed programming languages.

```
public static GCollection<T> Quicksort<T>(GCollection<T> data) where T : struct,
     IComparable<T>
{
  Contract.Requires(data.Count > 0);
  Contract.Ensures(Contract.Result<GCollection<T>>().Count == data.Count);
  Contract.Ensures(Contract.Result<GCollection<T>>().IsSorted((a, b) => a.CompareTo(b)));
  if (data.Count == 1) {
    return data;
  }

  T pivot = data[data.Count / 2];

  GCollection<T> left = data.Filter(d => d.CompareTo(pivot) == -1);
  GCollection<T> right = data.Filter(d => d.CompareTo(pivot) == 0);
  GCollectionr<T> mid = data.Filter(d => d.CompareTo(pivot) == 1);

  return Quicksort(left).Concat(mid).Concat(Quicksort(right));
}
```

Listing 4: Quicksort in SafeGPU for C#

5.1. Transferring Primitive and Class-Based Data

Transferring data from the host to the device is a necessary precursor to performing GPU computations. In C++ with raw CUDA, managing these transfers is relatively straightforward, but low-level and laborious. Operations such as `cudaMemcpy`, `cudaMalloc`, and `cudaFree` are provided to allocate, copy, and free memory.

SafeGPU handles this programming overhead for the user, but in languages such as Eiffel and C#, data transfer is made more complicated by the presence of a managed heap. In a naive implementation, two steps—and thus additional overhead—are required to realize it. First, the data is transferred from the managed heap into some fixed and contiguous structure. Second, this is then transferred to the device using low-level CUDA operations via the binding API.

The first step and its additional overhead, however, can be skipped entirely if the managed language provides a mechanism to directly access raw memory (e.g., via pointer-like constructs), and the representation of the data already has some known contiguous structure. This is often the case for arrays of primitive numerical type, e.g., integers, and floating points. A `memcpy` counterpart is typically available since their representation in memory is fixed across languages. In the C# port, we use a language mechanism that provides "unsafe" access to the memory of arrays of primitives. In our Eiffel implementation, direct access is also provided to the contents of arrays, but with the caveat that the array must be fixed during the `memcpy` call. If the array is not fixed, Eiffel's garbage collection mechanisms can interfere with the transfer and affect the consistency of the data.

Data typed according to custom classes is much more challenging to transfer. The CUDA implementation must be able to match the representation in memory, despite not have supporting definitions for custom classes. Furthermore, in general, classes can point to other classes, potentially requiring the (impractical) transfer of the whole object graph. Many classes of interest for GPU computing, however, are not sophisticated structures, but are rather more simple and just organize primitive data into a structure more suitable for the problem. SafeGPU thus focuses its support for class-based data on that which has a simple structure, i.e., based on primitives and value types.

Currently, our support for class-based data transfer has only been introduced into the C# port of SafeGPU, as the language provides convenient built-in mechanisms for managing the data. We support simple classes, i.e., those containing primitives, and value types from this definition. Using the P/Invoke feature of C#, the memory layout of such data is copied to unmanaged memory, where it is no longer typed. Then, we use reflection to collect meta-information about the structure being transferred, in particular, the number and types of fields (we do not translate methods at this point). Finally, a C++ counterpart of the C# structure is generated that can be handled by CUDA. We remark that while reflection in general can have some overhead, we attempt to minimize it by using the technique in a limited way, i.e., once per class, and without reflections in cycles.

Listing 5 exemplifies a simple application of class-based data transfer in SafeGPU. The method `DoStuff` operates on collections of `Complex` numbers, which are defined by a custom `struct` consisting of two doubles for the real and imaginary components of the numbers. The method chains together some `Map` transformations on the input collection and returns the result. To transfer the contents of the collection to the device, SafeGPU first copies its contents to unmanaged memory (no `memcpy` operation is available here, so it must loop across the elements), then copies this data from unmanaged memory to the device. Finally, it uses reflection to collect meta-information about the fields (`Re` and `Im`) in order to generate a counterpart in C++ to the original C# `struct`.

```
struct Complex
{ // omitting constructor and getters/setters for simplicity
  Double Re;
  Double Im;
}

GCollection<double> DoStuff(GCollection<Complex> collection )
{
  return collection. // any number of tranformations can be chained
    Map(c => new Complex {Re = c.Re + c.Im, Im = c.Im}).
      Map(complex => Math.Abs(complex.Re));
}
```

Listing 5: A SafeGPU for C# method operating on data typed to a custom structure

5.2. *Translating Customized Program Logic*

Providing a library of common operations for common collections is an important first step towards providing GPGPU capabilities at the abstraction level of an object-oriented language. In the SafeGPU approach, however, we do not want programmers to be strictly limited to the operations that we have provided. An important aspect of our approach is the ability to express customized computations in a functional style and apply them to whole collections.

As discussed in Section 3.2, the SafeGPU API provides programmers with methods that operate on the contents of entire collections. Methods such as `map` are generic transformations: their actual behavior on the contents of collections is customizable. This is achieved by passing a user-defined function abstraction (i.e., a typed function pointer) as a parameter of the transformation. In Eiffel, we support agents as function abstractions; in the C# port, we support delegates. By translating these function abstractions to C++ and CUDA, our framework supports the execution of customized program logic on the GPU.

In the following we illustrate how function abstractions can be translated to the GPU using the example of delegates in our C# port of SafeGPU. Our solution relies on the powerful support provided by C# and .NET for runtime introspection and analysis, and in particular, the expression trees framework [24]. With this support, it is possible to dynamically access and modify arbitrary

C# expressions during program execution, which allows SafeGPU to capture the customized program logic that the user wishes to use in a collection transformation.

Listing 6 shows how simple expressions can be created in the expression tree framework. The first expression captures a double constant; the second, a mathematical expression over variables; and the third, a function taking an integer input and returning a string (expressed by the first and second generic arguments, respectively). The string is generated in-place by a lambda-expression, which creates a formal variable `a` and calls the `ToString` operation on it. The variable is implicitly typed as an integer, which helps to keep the expression syntactically simple. All three expressions are represented in the framework as tree-like data structures (the nodes being expressions), and can be compiled and modified at runtime.

```
{
  Expression<double> ex1 = 5.2;

  Expression<double> ex2 = a + b / 5.0;

  Expression<Function<int, string>> strExpr = (a) => a.ToString();
}
```

Listing 6: Example expression trees in C#

SafeGPU uses the framework to extract tree representations of delegates. Consider the signature of `Map` in the C# API:

```
GCollection<T> Map (Expression<Func<T,T>> transformer);
```

When a call to `Map` is processed by SafeGPU, the expression trees framework is used to extract an AST representation of `transformer`, which in turn can be translated to C++ / CUDA.

There are some restrictions on what can be translated and the types of functional expressions that can be created. The expression tree framework, for example, does not support lambdas with statement bodies. Furthermore, methods must operate on either primitive types or the types that SafeGPU can translate itself (we cannot translate any arbitrary .NET method to C++ / CUDA).

Support for customizable program logic can be generalized to other managed languages if analogous mechanisms exist for runtime analysis of program code. Unfortunately, such mechanisms are lacking in Eiffel, meaning that agent expressions (i.e., Eiffel's function objects) are translated much less elegantly by SafeGPU: at present, we treat them as strings and must manually parse them. (Note that example usages of Eiffel's agents can be found in Sections 3.2 and 4.)

5.3. Deterministic Memory Management in Languages with Garbage Collection

In C++ / CUDA, the programmer has full and explicit control over the device memory. In languages with managed memory such as Eiffel and C#, one must consider how to deterministically dispose of external resources in the presence of garbage collection, which can occur at (seemingly) random time intervals, or perhaps not happen at all (e.g., if the garbage collector assesses that there is enough memory). Since the host and device memories are disjoint, the garbage collector might not become aware when the device no longer has enough memory.

A closely related problem to this is the avoidance of leaking memory between allocation and deallocation in the presence of exceptions. We investigated how this problem was solved in an unmanaged language, and used the solution as inspiration. C++ tackles it using the RAII (Resource Acquisition Is Initialization) idiom [25]. The essential idea is to use stack allocation and variable scope to ensure safe resource management. For example, in Listing 7, `locker` is called whenever the thread of execution leaves the scope encompassing it, e.g., during exception propagation. RAII is a very common idiom in C++: dynamic memory, file system descriptors, and concurrency primitives can all be managed using it.

```
{
  MutexLocker locker(new Mutex());
  ...
} // locker is called whenever the execution leaves the scope, whether during a normal
    execution or during an exception propagation
```

Listing 7: A possible RAII idiom usage

The guarantees provided by RAII would be useful for implementing a translation to C++ / CUDA, but unfortunately, RAII is not directly applicable to languages with managed memory and garbage collection. However, managed languages often provide substitute mechanisms that are similar. For this to work, the runtime must be aware that some managed objects store handlers (e.g., memory addresses, descriptors) of resources in unmanaged memory. Typically, this is achieved by implementing a special interface or inheriting from a special base class.

In C#, this is the IDisposable interface, and the language has special support for it: if a class or interface implements it, then their objects can be used in so-called “using-blocks” which emulate C++ scoping. Such a block is given in Listing 8: disposal is called whenever execution leaves the scope of the block. Java has `java.lang.AutoCloseable` and try-with-resources, which is very similar to the using-blocks of C#. Eiffel has the Disposable base class.

```
using (new ResourceHandler()) {
  ...
} // Disposal is called whenever the execution leaves the scope, whether during a normal
    execution or during an exception propagation
```

Listing 8: A using-block in C#

6. Kernel Generation and Optimization

In this section we describe how SafeGPU translates individual methods of the API to CUDA kernels, how data is managed, and how the library optimizes kernels for compound computations.

6.1. Kernel Generation and Data Transfer

Generating CUDA kernels for calls of individual library methods is straightforward. Each method is associated with a kernel template, which the library instantiates with respect to the particular collection and parameters of the method call. The SafeGPU runtime (as described in Section 2.2) then handles its execution on the GPU via Eiffel’s mechanisms for interfacing with C++ or the existing CUDA binding for C#.

Transferring data to and from the GPU is expensive, so the library attempts to minimize the number of occurrences. The only time that data is transferred to the GPU is upon calling the method `from_array`, which creates a GPU collection from a standard Eiffel or C# array. Once the data is on the GPU, it remains there for arbitrarily many kernels to manipulate and query (including those corresponding to contracts). Operations that create new collections from existing ones (e.g., `filter`, `map`) do so without transferring data away from the GPU; this occurs only for methods that specifically query them.

6.2. Execution Plans and Kernel Optimization

While the primitive operations in isolation already support many useful computations (e.g., matrix multiplication, vector addition), the heart of SafeGPU is in its support for combining and chaining such operations to implement multistage algorithms on the GPU. The main challenge for a library aiming to provide this support is to do so without performance becoming incommensurate with that of manually written CUDA kernels. A naive solution is to generate one kernel per method call and launch them one after the other. With SafeGPU, however, we adopt a deferred execution model, analyze pending kernels, and attempt to generate more efficient CUDA code by combining them.

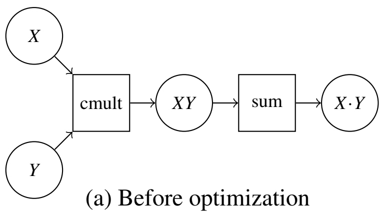
By default, a method call is not executed, but rather added to a list of pending actions for the corresponding collection. There are three ways to trigger its execution: (1) perform a function call that returns a scalar value, e.g., `sum`; (2) perform a call to `to_array` which creates a standard Eiffel or C# array from the GPU collection; or (3) perform a call of the special method `update`, which forces the execution of any pending kernels.

Consider for example the problem of computing the dot product (or inner product) of two vectors, which can be solved by combining vector multiplication and vector summation as in Listing 9. Here, the result is obtained by chaining the `a.compwise_multiply (b)` method—which produces an anonymous intermediate result—with `vector.sum`. In this example, the computation is deferred until the call of `sum`, which returns the sum of the elements in the vector.

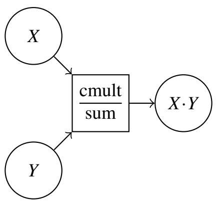
```
dot_product (a, b: G_VECTOR[DOUBLE]): DOUBLE
  require
    a.count = b.count
  do
    Result := a.compwise_multiply (b).sum
    -- component-wise vector multiplication, followed by summing the elements
  end
```

Listing 9: Combining primitives to compute the dot product in SafeGPU for Eiffel

The benefit of deferring execution until necessary is that the kernel code can be optimized. Instead of generating kernels for every method call, SafeGPU uses some simple strategies to merge deferred calls and thus handle the combined computation in fewer kernels. Before generating kernels, the optimizer constructs an execution plan from the pending operations. The plan takes the form of a DAG, representing data and kernels as two different types of nodes, and representing dependencies as edges between them. The optimizer then traverses the DAG, merging kernel vertices and collapsing intermediate dependencies where possible. Upon termination, the kernel generation takes place on the basis of the optimized DAG.



(a) Before optimization

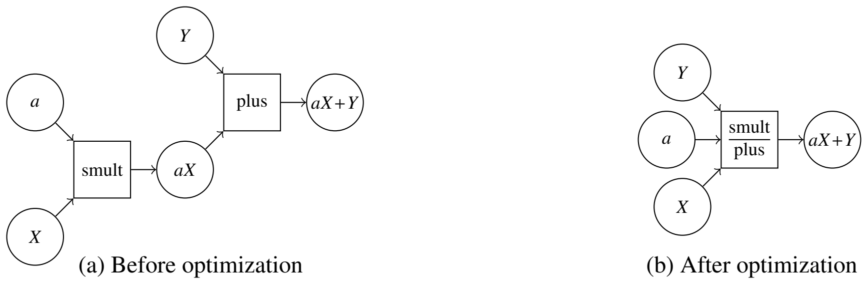


(b) After optimization

Figure 2: Execution plans for the dot product method

We illustrate a typical optimization in Figure 2, which shows the execution plans for the dot product method of Listing 9. The plan in Figure 2a is the original one extracted from the pending operations; this would generate two separate kernels for multiplication and summation (cmult and sum) and launch them sequentially. The plan in Figure 2b, however, is the result of an optimization; here, the deferred cmult kernel is combined with sum. The combined kernel generated by this optimized execution plan would perform component-wise vector multiplication first, followed by summation, with the two stages separated using barrier synchronization. This simple optimization pattern extends to several other similar cases in SafeGPU.

The optimizer is particularly well-tuned for computations involving vector mathematics. In some cases, barriers are not needed at all; the optimizer simply modifies the main expression in the kernel body, leading to more efficient code. For example, to compute $aX + Y$ where a is a scalar value and X, Y are vectors, the optimizer just slightly adjusts the vector addition kernel, replacing `X[i] + Y[i]` with `a∗X[i] + Y[i]`. Such optimizations also change the number of kernel arguments, as shown in Figure 3.



(a) Before optimization (b) After optimization

Figure 3: Execution plans for vector mathematics

At present, all our optimizations are of these simple forms. The execution plan framework, however, could provide a foundation for applying more speculative and advanced optimizations to improve the performance of SafeGPU further still. Investigating such optimizations remains an important piece of future work.

6.3. Example: Gaussian Elimination

To illustrate the usefulness of the optimizer on a larger example, consider Listing 10, which provides an implementation of Gaussian elimination (i.e., for finding the determinant of a matrix) in SafeGPU. Note in particular the inner loop, which applies two transformations in sequence to a given row of the matrix:

```
matrix.row (i).divided_by (pivot)
matrix.row (i).in_place_minus (matrix.row (step))
```

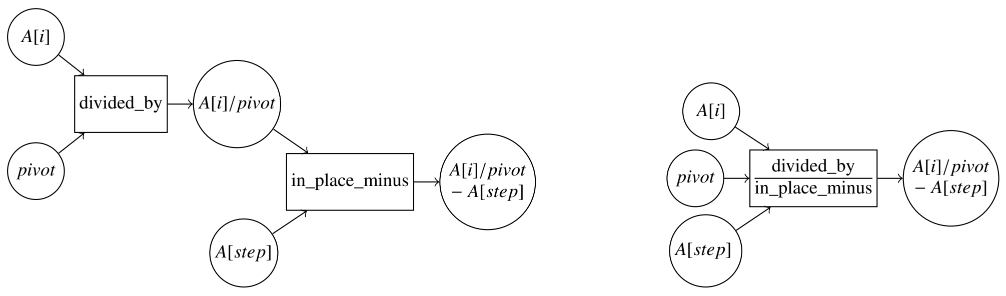
First, every element in the row is divided by a pivot (which an earlier check prevents from being zero); following this, another row of the matrix is subtracted from it in a component-wise fashion. The optimizer is able to combine these two steps into a single modified component-wise subtraction kernel, applying the transformation `(A[i] / pivot) − A[step]` in one step (here, `A[x]` denotes row `x` of matrix `A`). This optimization is depicted in Figure 4.

```
gauss_determinant (matrix: G_MATRIX[DOUBLE]): DOUBLE
  require
    matrix.rows = matrix.columns
  local
    step, i: INTEGER
    pivot: DOUBLE
  do
    Result := 1
    from
      step := 0
    until
      step = matrix.rows
    loop
      pivot := matrix (step, step)
      Result := Result * pivot

      if not double_approx_equals (pivot, 0.0) then
        matrix.row (step).divided_by (pivot)
      else
        step := matrix.rows
      end
      from
        i := step + 1
      until
        i = matrix.rows
      loop
        pivot := matrix (i, step)
        if not double_approx_equals (pivot, 0.0) then
          matrix.row (i).divided_by (pivot)
          matrix.row (i).in_place_minus (matrix.row (step))
        end
        i := i + 1
      end

      step := step + 1
    end
  end
```

Listing 10: Gaussian elimination in SafeGPU for Eiffel



(a) Before optimization (b) After optimization

Figure 4: Execution plans for the inner loop of Gaussian elimination

7. Evaluation

To evaluate SafeGPU, we prepared a set of benchmark problems to solve on the GPU, each with functionally equivalent implementations in sequential Eiffel, SafeGPU for Eiffel, and raw CUDA in C++. To establish a baseline, we covered some problems that have well-established implementations available in the NVIDIA SDK, such as vector addition and matrix multiplication. Beyond this baseline, we also considered larger examples constructed by chaining the primitive operations of our library, such as Gaussian elimination and quicksort. Across our benchmark set, we made three different comparisons (which we expand upon in the following subsections):

1. the performance of SafeGPU against CUDA and sequential Eiffel;
2. the conciseness of SafeGPU against sequential Eiffel;
3. the performance overhead of runtime contract checking in SafeGPU against checking traditional sequential contracts in Eiffel.

The six benchmark programs we considered were vector addition, dot product, matrix multiplication, Gaussian elimination, quicksort, and matrix transposition. We implemented the benchmarks ourselves for SafeGPU and sequential Eiffel (both with contracts, wherever possible). We did not implement but rather relied on a selection of sources for the plain CUDA code: vector addition and matrix multiplication were taken from the NVIDIA SDK; dot product and quicksort were adapted from code in the same repository; Gaussian elimination came from a parallel computing research project [26]; and finally, matrix transposition came from a post [27] on NVIDIA's Parallel Forall blog.

The SafeGPU implementation and all the benchmarks are available to download online [17]. Listings for the SafeGPU implementations of quicksort and Gaussian elimination are also provided in this paper (Listings 3 and 10, respectively).

We remark that we use our principal, Eiffel implementation of SafeGPU in these experiments, given that at the time of writing, our C# port remains an early prototype. Given the (intended) similarities of the two implementations, any significant differences in performance might bring about some interesting insights into language-specific overheads (but would not otherwise affect the investigation here, which asks whether one *can* provide the functionality of SafeGPU in some object-oriented library without paying a large price in performance).

7.1. Performance

The primary goal of our first experiment was to assess the performance overhead caused by SafeGPU's higher level of abstraction. To measure this, we compared the execution times of benchmarks in SafeGPU against those in plain CUDA for increasingly large sizes of input. Furthermore, we compared our benchmarks against functionally equivalent solutions in sequential Eiffel, allowing us to ascertain the input sizes necessary for GPU solutions to outperform them. We remark that since performance was the focus of this first experiment, runtime contract checking was completely disabled across all benchmarks.

All experiments were performed on the following hardware: Intel Core i7 8 cores, 2.7 GHz; NVIDIA QUADRO K2000M (2 GB memory, compute capability 3.0). In our measurements, we are reporting wall time. Furthermore, we measure only the relevant part of the computation, omitting the time it takes to generate the inputs.

The results of our performance comparison are presented in Figure 5. The problem size (x-axis) is defined for both vectors and matrices as the total number of elements they contain (our benchmarks use only square matrices, hence the number of rows or columns is always the square root). The times (y-axis) are given in seconds, and are the medians of ten runs.

While sequential Eiffel is faster than SafeGPU and plain CUDA on relatively small inputs (as expected, due to the overhead of launching the GPU), it is outperformed by both when the size of the data becomes large. This happens particularly quickly for the non-linear algorithm (e) in comparison to the others. For matrix-matrix multiplication and Gaussian elimination, sequential Eiffel took far too long to terminate on inputs of size 10^7 and above, and hence these data points are omitted.

Across most of the six benchmarks, the performance of SafeGPU is very close to that of plain CUDA, adding support to our argument that using our library does not lead to performance incommensurate with that of handwritten CUDA code. The Gaussian elimination benchmark displays the largest difference between SafeGPU and plain CUDA, on inputs of size 10^6 and above. This is due to the need for the SafeGPU implementation to use nested loops, which have the effect of additional kernel launches. This could be addressed in the future by API extensions, or the introduction of more speculative optimization strategies designed for loops. Note that in some benchmarks (especially on smaller inputs), SafeGPU sometimes slightly outperforms plain CUDA, which we believe is due to differences between the memory managers of Eiffel and C++.

7.2. Code Size

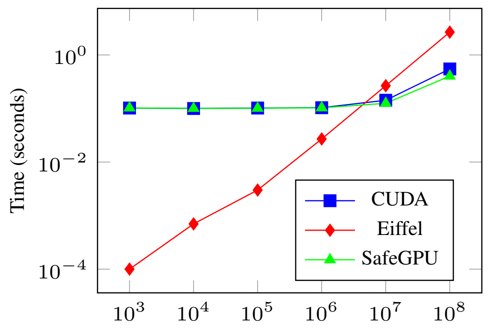
The goal of our second experiment was to assess the conciseness of SafeGPU programs. To measure this, we compared the lines of code (LOC) required for the main methods of these programs (and any auxiliary methods) against the LOC of functionally equivalent sequential Eiffel methods. Note that we do not compare against plain CUDA programs, because this is not a particularly interesting comparison to make: it is known that higher-level languages are more compact than those at the C/C++ level of abstraction [28], and CUDA programs in particular are dominated by explicit memory management that is not visible in SafeGPU or Eiffel. Our CUDA benchmarks are typically around 200 LOC code long (and sometimes more).

Our results are presented in Table 6. The programs written using our library are quite concise (as expected for a high-level API); more interestingly, they are more compact than traditional sequential Eiffel programs. This difference is explained by the usage of looping constructs. In sequential Eiffel, loops are frequently used to implement the benchmarks. With SafeGPU, however, loops are often avoided due to the presence of bulk operations in the API, i.e., operations that apply function abstractions to all the data present in a collection. We should note that this is not always the case, as loops were required to implement the library version of the Gaussian elimination benchmark.

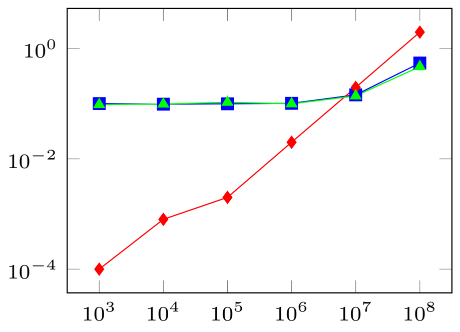
We remark that while these results suggest that SafeGPU programs are more compact, we do not yet know whether typical programmers can write them more productively. In future work, we would like to perform a study on users themselves in order to determine whether the abstractions and programming style of our approach allow for users to write programs productively, regardless of their conciseness.

7.3. Contract Overhead

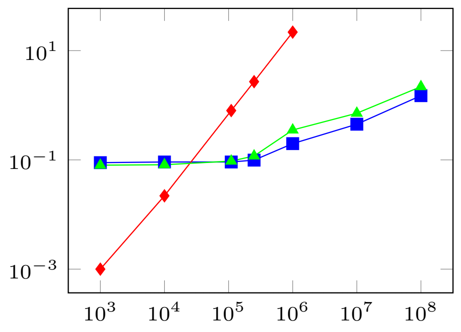
The goal of our final experiment was to compare the cost of checking SafeGPU contracts on the GPU against the cost of checking traditional sequential Eiffel contracts. To allow a more fine-grained comparison, we measured the contract checking overhead in three different modes: (1) preconditions only; (2) pre- and postconditions only; and finally, (3) full contract checking, i.e., additionally checking class invariants at method entry and exit points. Note that our SafeGPU benchmarks were annotated only with pre- and postconditions; invariants, however, are present in the core Eiffel libraries that were required to implement the sequential programs (these libraries also include some additional pre- and postconditions, making a full like-for-like comparison with SafeGPU challenging). Across the benchmarks and for increasingly large sizes of input,



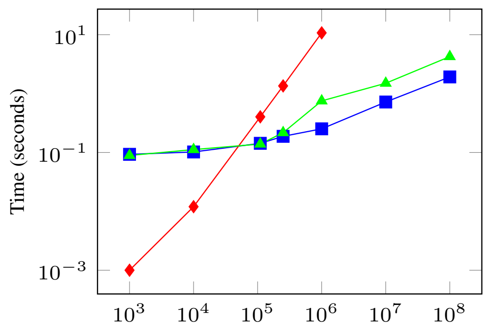
(a) Vector Addition



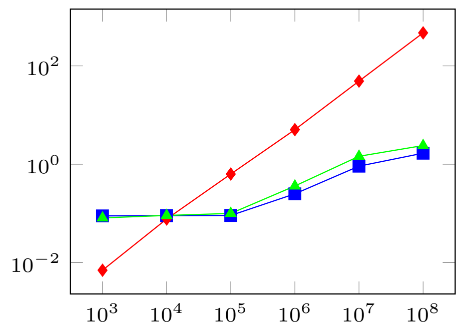
(b) Dot Product



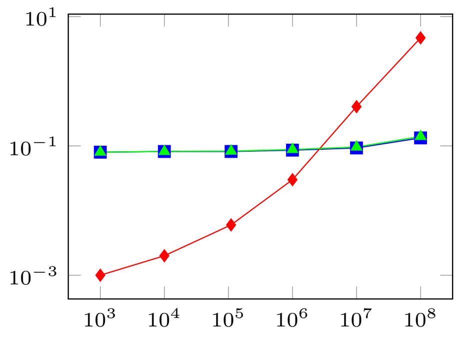
(c) Matrix-Matrix Multiplication



(d) Gaussian Elimination



(e) Quicksort



(f) Matrix Transpose

Figure 5: SafeGPU performance evaluation (*x*-axis: input size in no. of elements)

Table 6: LOC comparison

problem	Eiffel	SafeGPU	ratio
Vector Addition	18	8	2.3
Dot Product	16	6	2.7
Matrix-Matrix Multiplication	32	6	5.3
Gaussian Elimination	98	47	2.1
Quicksort	63	22	2.9
Matrix Transpose	27	8	3.4

Table 7: Contract checking overhead comparison

problem		10^3		10^4		10^5		10^6		10^7		10^8	
		Eiffel	SafeGPU	Eiffel	SafeGPU	Eiffel	SafeGPU	Eiffel	SafeGPU	Eiffel	SafeGPU	Eiffel	SafeGPU
Vector Addition	pre	1.00	0.92	1.42	0.96	3.50	0.96	3.92	0.95	3.98	1.02	4.12	1.06
	pre & post	1.00	0.92	1.42	0.96	3.66	0.96	3.93	0.95	3.98	1.02	4.29	1.06
	full	1.00	0.92	2.86	0.96	7.00	0.96	7.81	0.95	7.82	1.02	7.97	1.06
Dot Product	pre	1.00	1.02	1.25	0.99	4.00	0.97	3.95	1.01	4.00	1.10	4.01	0.95
	pre & post	1.00	1.02	1.25	0.99	4.00	0.97	3.95	1.01	4.15	1.10	4.10	0.98
	full	1.00	1.02	1.88	0.99	7.25	0.97	7.33	1.01	7.46	1.10	7.48	0.98
Matrix-Matrix Multiplication	pre	4.00	1.05	4.47	1.01	4.55	0.99	4.54	0.99	-		-	
	pre & post	4.00	1.05	4.47	1.01	4.59	0.99	4.57	0.99	-		-	
	full	5.00	1.05	6.73	1.01	6.79	1.01	6.76	0.99	-		-	
Gaussian Elimination	pre	2.22	0.99	4.50	0.97	4.70	1.01	4.71	1.01	-		-	
	pre & post	2.77	0.99	4.50	0.97	4.70	1.04	4.73	1.09	-		-	
	full	4.44	0.99	6.67	0.97	6.96	1.04	6.96	1.09	-		-	
Quicksort	pre	2.14	1.02	2.26	1.05	2.64	1.00	3.03	1.01	3.03	1.02	-	
	pre & post	2.28	1.02	2.27	1.05	2.70	1.02	3.02	1.07	3.04	1.08	-	
	full	3.64	1.02	4.14	1.05	5.07	1.02	6.38	1.07	6.49	1.09	-	
Matrix Transposition	pre	2.00	1.05	2.06	1.01	2.40	1.02	3.71	1.01	3.86	1.02	4.02	1.01
	pre & post	2.00	1.05	2.06	1.01	2.40	1.03	3.96	1.11	4.05	1.12	4.27	1.14
	full	4.15	1.03	5.60	1.01	6.10	1.03	7.88	1.10	8.12	1.12	10.44	1.13

we computed ratios expressing the performance overhead resulting from enabling each of these three modes against no contract checking at all. The ratios are based on medians of ten runs (an effect of using medians is that some ratios can be less than 1).

Our data is presented in Table 7, where a ratio *X* can be interpreted as meaning that the program was *X* times slower with the given contract checking mode enabled. The comparison was unable to be made for some benchmarks with the largest inputs (indicated by dashes), as it took far too long for the sequential Eiffel programs to terminate. We remark that vector addition, dot product, and matrix-matrix multiplication have only scalar contracts; Gaussian elimination, quicksort, and matrix transposition have a combination of both scalar and range contracts (see Section 4 for their definitions).

There is an encouraging difference between the contract-checking overhead in sequential Eiffel and SafeGPU: while the former cannot maintain reasonable contract performance on larger inputs (the average slowdown for the "full" mode across benchmarks with input size 10^6, for example, is 7.19), SafeGPU has for the most part little-to-no overhead. Disabling invariant-checking leads to improvements for sequential Eiffel (which, unlike SafeGPU, relies on invariant-equipped library classes), but the average slowdown is still significant (now 4.03, for input size 10^6). Across these benchmarks, postcondition checking adds little overhead to sequential Eiffel above checking preconditions only (which has an average slowdown of 3.98 for input size 10^6). SafeGPU performs consistently well in all modes of the experiment, with slowdown close to 1 across the first three benchmarks. The other three benchmarks perform similarly for precondition checking, but as they include more elaborate postconditions (e.g., "the vector is sorted"), checking both pre- and postconditions can lead to a small slowdown on large data (1.14 in the worst case for this experiment). Overall, the results lend support to our claim that SafeGPU contracts can be monitored at runtime without diminishing the performance of the program, even with large amounts of data. Unlike sequential Eiffel programs, contract checking need not be limited to periods of debugging.

8. Related Work

There is a vast and varied literature on general-purpose computing with GPUs. We review a selection of it, focusing on work that particularly relates to the overarching themes of SafeGPU: the *generation* of low-level GPU kernels from higher-level programming abstractions, and the *correctness* of the kernels to be executed.

8.1. GPU Programming and Code Generation

At the C++ level of abstraction, there are a number of algorithmic skeleton and template frameworks that attempt to hide the orchestration and synchronization of parallel computation. Rather than code it directly, programmers express the computation in terms of some well-known patterns (e.g., map, scan, reduce) that capture the parallel activities implicitly. SkePU [5], Muesli [6], and SkelCL [9] were the first algorithmic skeleton frameworks to target the deployment of fine-grained data-parallel skeletons to GPUs. While they do not support skeleton nesting for GPUs, they do provide the programmer with parallel container types (e.g., vectors, matrices) that simplify memory management by handling data transfers automatically. Arbitrary skeleton nesting is provided in FastFlow [7] (resp. Marrow [8]) for pipeline and farm (resp. pipeline, stream, loop), but concurrency and synchronization issues are exposed to the programmer. NVIDIA's C++ template library Thrust [29], in contrast, provides a collection of data-parallel primitives (e.g., scan, sort, reduce) that can be composed to implement complex algorithms on the GPU. While similar in spirit to SafeGPU, Thrust lacks a number of its abstractions and container types; data can only be modeled by vectors, for example.

Higher-level programming languages benefit from a number of CUDA and OpenCL bindings (e.g., Java [22], Python [21]), making it possible for their runtimes to interact. These bindings typically stay as close to the original models as possible. While this allows for the full flexibility and control of CUDA and OpenCL to be integrated, several of the existing challenges are also inherited, along with the addition of some new ones; Java programmers, for example, must manually translate complex object graphs into primitive arrays for use in kernels. Rootbeer [15], implemented on top of CUDA, attempts to alleviate such difficulties by automatically serializing objects and generating kernels from Java code. Programmers, however, must still essentially work in terms of threads—expressed as special kernel classes—and are responsible for instantiating and passing them on to the Rootbeer system for execution on the GPU.

There are several dedicated languages and compilers for GPU programming. Lime [10] is a Java-compatible language equipped with high-level programming constructs for task, data, and pipeline parallelism. The language allows programmers to code in a style that separates computation and communication, and does not force them to explicitly partition the parts of the program for the CPU and the parts for the GPU. CLOP [13] is an embedding of OpenCL in the D language, which uses the standard facilities of D to generate kernels at compile-time. Programmers can use D variables directly in embedded code, and special constructs for specifying global synchronization patterns. The CLOP compiler then generates the appropriate boilerplate code for handling data transfers, and uses the patterns to produce efficient kernels for parallel computations. Nikola [12] is an embedding of an array computation language in Haskell, which compiles to CUDA and (like SafeGPU) handles data transfer and other low-level details automatically. Other languages are more domain-specific: StreamIt [30], for example, provides high-level abstractions for stream processing, and can be compiled to CUDA code via streaming-specific optimizations [11]; and VOBLA [31], a domain-specific language (DSL) for programming linear algebra libraries, restricts what the programmer can write, but generates highly optimized OpenCL code for the domain it supports. Finally, Delite [32] is a compiler framework for developing embedded DSLs themselves, providing common components (e.g., parallel patterns, optimizations, code generators) that can be re-used across DSL implementations, and support for compiling these DSLs to both CUDA and OpenCL.

A key distinction of SafeGPU is the fact that GPGPU is offered to the programmer without forcing them to switch to a dedicated language in the first place: both the high-level API and the CUDA binding are made available through a library, and without need for a special-purpose compiler. Firepile [14] is a related library-oriented approach for Scala, in which OpenCL kernels are generated using code trees constructed from function values at runtime. Firepile supports objects, higher-order functions, and virtual methods in kernel functions, but does not support programming at the same level of abstraction as SafeGPU: barriers and the GPU grid, for example, are exposed to developers.

8.2. Correctness of GPU Kernels

To our knowledge, SafeGPU is the first GPU programming approach to integrate the specification and runtime monitoring of functional properties directly at the level of an API. Other work addressing the correctness of GPU programs has tended to focus on analyzing and verifying kernels themselves, usually with respect to concurrency faults (e.g., data races, barrier divergence).

PUG [33] and GPUVerify [34, 35] are examples of static analysis tools for GPU kernels. The former logically encodes program executions and uses an SMT solver to verify the absence of faults such as data races, incorrectly synchronized barriers, and assertion violations. The latter tool verifies race- and divergence-freedom using a technique based on tracking reads and writes in shadow memory, encoded in Boogie [36].

Blom et al. [37] present a logic for verifying both data race freedom and functional correctness of GPU kernels in OpenCL. The logic is inspired by permission-based separation logic: kernel code is annotated with assertions expressing both their intended functionality, as well as the resources they require (e.g., write permissions for particular locations).

Other tools seek to show the presence of data races, rather than verify their absence. Examples include GKLEE [38] and KLEE-CL [39], both based on dynamic symbolic execution.

9. Conclusion

We presented SafeGPU: a contract-based, modular, and efficient approach for library-based GPGPU in object-oriented languages, demonstrated through a prototype implementation for Eiffel and an initial port for C#. The techniques of deferred execution and execution plan optimization helped to keep the library performance on par with raw CUDA solutions. Unlike CUDA programs, SafeGPU programs are concise and equipped with contracts, thereby contributing to program safety. We also found that GPU-based contracts can largely avoid the overhead of assertion checking. In contrast to classical, sequential contracts, it is feasible to monitor them outside of periods of debugging: data size is not an issue anymore.

This work can be extended in a variety of directions. In the current implementation, the optimizer is tailored to linear algebra and reduction/scan problems. Global optimizations could be introduced, such as changing the order of operations, or handling loops in a more efficient way. Furthermore, as shown in Section 7, GPU computing is not yet fast enough on “small” data sets. This could be resolved by introducing a hybrid computing model, in which copies of data are maintained on both the CPU and GPU. This could allow for switching between CPU and GPU executions depending on the runtime context.

To provide better support for task parallelism, SafeGPU could be integrated with a thread-based library. We could also investigate the integration of SafeGPU with a higher-level concurrency model, such as Eiffel’s SCOOP [40], which provides contract-based and transaction-like reasoning over concurrent and distributed [41] objects.

There is a need to evaluate SafeGPU further on a broader set of benchmarks, to gain a better understanding of where the library is useful and where further research is necessary. We also plan to investigate its use in larger case studies, in particular, applying the library to speed up embarrassingly-parallel evolutionary algorithms used in test data generation (as outlined in [42]).

Finally, in the work presented, we focused on single-GPU systems. In practice, however, multi-GPU systems are becoming increasingly ubiquitous. Integrating (and thus benefiting from) multiple accelerator devices in future versions of the SafeGPU approach is hence a particularly important item of future work. Multi-accelerator systems (possibly from different manufacturers, with different computing capabilities) bring a a range of new challenges. How do you, for example, distribute your computations? Simply transferring your data—as we did—to a single-GPU system is no longer sufficient. How do you manage the load? How do you deal with data dependencies across several devices? And how do you choose the best device for the current (sub-)problem?

A possible pathway to supporting multi-GPU systems is to work with “data chunks”, representing parts of the original data. This would also allow for the processing of data arrays that are too large for just a single device. Working with chunks, however, requires a more complex orchestration of computations: the framework must carefully manage partial transfers, assemble data back from chunks, attempt to avoid inter-chunk data dependencies, and manage the balance of work across devices (solutions to such challenges might take inspiration from existing research in the setting of distributed computing). Furthermore, chunked data and multi-GPU systems might lead to a new class of kernel optimizations not possible in the current setting of single-GPU systems.

Acknowledgements. We thank the anonymous referees of GPCE ’15 and COMLAN for their comments and criticisms, which helped to improve the final version of this paper. This work has been supported in part by ERC Grant CME #291389.

References

[1] NVIDIA: GPU Applications, `http://www.nvidia.com/object/gpu-applications.html` (accessed: Jun. 2016).

[2] S. Yoo, M. Harman, S. Ur, GPGPU test suite minimisation: search based software engineering performance improvement using graphics cards, Empirical Software Engineering 18 (3) (2013) 550–593.

[3] NVIDIA: CUDA Parallel Computing Platform, `http://www.nvidia.com/object/cuda_home_new.html` (accessed: Jun. 2016).

[4] Khronos OpenCL Working Group, The OpenCL specification: Version 1.2, `https://www.khronos.org/registry/cl/specs/opencl-1.2.pdf` (2012).

[5] J. Enmyren, C. W. Kessler, SkePU: A multi-backend skeleton programming library for multi-GPU systems, in: Proceedings of the 4th International Workshop on High-level Parallel Programming and Applications (HLPP ’10), ACM, 2010, pp. 5–14.

[6] S. Ernsting, H. Kuchen, Algorithmic skeletons for multi-core, multi-GPU systems and clusters, International Journal of High Performance Computing and Networking 7 (2) (2012) 129–138.

[7] M. Goli, H. González-Vélez, Heterogeneous algorithmic skeletons for FastFlow with seamless coordination over hybrid architectures, in: Proceedings of the 21st Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP ’13), IEEE, 2013, pp. 148–156.

[8] R. Marqués, H. Paulino, F. Alexandre, P. D. Medeiros, Algorithmic skeleton framework for the orchestration of GPU computations, in: Proceedings of the 19th International Conference on Parallel Processing (Euro-Par ’13), Vol. 8097 of LNCS, Springer, 2013, pp. 874–885.

[9] M. Steuwer, S. Gorlatch, SkelCL: Enhancing OpenCL for high-level programming of multi-GPU systems, in: Proceedings of the 12th International Conference on Parallel Computing Technologies (PaCT ’13), Vol. 7979 of LNCS, Springer, 2013, pp. 258–272.

[10] C. Dubach, P. Cheng, R. Rabbah, D. F. Bacon, S. J. Fink, Compiling a high-level language for GPUs: (via language support for architectures and compilers), in: Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI ’12), ACM, 2012, pp. 1–12.

[11] A. H. Hormati, M. Samadi, M. Woh, T. Mudge, S. Mahlke, Sponge: Portable stream programming on graphics engines, in: Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS ’11), ACM, 2011, pp. 381–392.

[12] G. Mainland, G. Morrisett, Nikola: Embedding compiled GPU functions in Haskell, in: Proceedings of the 3rd ACM SIGPLAN Symposium on Haskell (Haskell ’10), ACM, 2010, pp. 67–78.

[13] D. Makarov, M. Hauswirth, CLOP: A multi-stage compiler to seamlessly embed heterogeneous code, in: Proceedings of the 14th International Conference on Generative Programming: Concepts and Experiences (GPCE ’15), ACM, 2015, pp. 109–112.

[14] N. Nystrom, D. White, K. Das, Firepile: Run-time compilation for GPUs in Scala, in: Proceedings of the 10th International Conference on Generative Programming and Component Engineering (GPCE ’11), ACM, 2011, pp. 107–116.

[15] P. C. Pratt-Szeliga, J. W. Fawcett, R. D. Welch, Rootbeer: Seamlessly using GPUs from Java, in: Proceedings of the 14th International Conference on High Performance Computing and Communication & 9th International Conference on Embedded Software and Systems (HPCC-ICESS ’12), IEEE, 2012, pp. 375–380.

[16] T. Rompf, K. J. Brown, H. Lee, A. K. Sujeeth, M. Jonnalagedda, N. Amin, G. Ofenbeck, A. Stojanov, Y. Klonatos, M. Dashti, C. Koch, M. Püschel, K. Olukotun, Go meta! A case for generative programming and DSLs in performance critical systems, in: Proceedings of the 1st Summit on Advances in Programming Languages (SNAPL '15), Vol. 32 of LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015, pp. 238–261.

[17] SafeGPU Repository, `https://bitbucket.org/alexey_se/eiffel2cuda`.

[18] B. Meyer, Object-Oriented Software Construction, 2nd Edition, Prentice-Hall, 1997.

[19] Code Contracts, `http://research.microsoft.com/en-us/projects/contracts/` (accessed: Jun. 2016).

[20] A. Kolesnichenko, C. M. Poskitt, S. Nanz, B. Meyer, Contract-based general-purpose GPU programming, in: Proceedings of the 14th International Conference on Generative Programming: Concepts and Experiences (GPCE '15), ACM, 2015, pp. 75–84.

[21] A. Klöckner, N. Pinto, Y. Lee, B. C. Catanzaro, P. Ivanov, A. Fasih, PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation, Parallel Computing 38 (3) (2012) 157–174.

[22] Y. Yan, M. Grossman, V. Sarkar, JCUDA: A programmer-friendly interface for accelerating Java programs with CUDA, in: Proceedings of the 15th International Euro-Par Conference on Parallel Processing (Euro-Par '09), Vol. 5704 of LNCS, Springer, 2009, pp. 887–899.

[23] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, E. Poll, An overview of JML tools and applications, International Journal on Software Tools for Technology Transfer 7 (3) (2005) 212–232.

[24] Expression Trees, `https://msdn.microsoft.com/en-us/library/bb397951.aspx` (accessed: Jun. 2016).

[25] B. Stroustrup, The Design and Evolution of C++, Addison-Wesley, 1994.

[26] Linear Algebra: Gaussian Elimination, `http://www.cs.rutgers.edu/~venugopa/parallel_summer2012/ge.html` (accessed: Jun. 2016).

[27] M. Harris, An efficient matrix transpose in CUDA C/C++, `http://devblogs.nvidia.com/parallelforall/efficient-matrix-transpose-cuda-cc/` (accessed: Jun. 2016).

[28] S. Nanz, C. A. Furia, A comparative study of programming languages in Rosetta Code, in: Proceedings of the 37th International Conference on Software Engineering (ICSE '15), IEEE, 2015, pp. 778–788.

[29] NVIDIA: CUDA Toolkit Documentation – Thrust, `http://docs.nvidia.com/cuda/thrust/` (accessed: Jun. 2016).

[30] W. Thies, M. Karczmarek, S. P. Amarasinghe, StreamIt: A language for streaming applications, in: Proceedings of the 11th International Conference on Compiler Construction (CC '02), Vol. 2304 of LNCS, Springer, 2002, pp. 179–196.

[31] U. Beaugnon, A. Kravets, S. van Haastregt, R. Baghdadi, D. Tweed, J. Absar, A. Lokhmotov, VOBLA: A vehicle for optimized basic linear algebra, in: Proceedings of the ACM SIGPLAN Conference on Languages, Compilers and Tools for Embedded Systems (LCTES '14), ACM, 2014, pp. 115–124.

[32] A. K. Sujeeth, K. J. Brown, H. Lee, T. Rompf, H. Chafi, M. Odersky, K. Olukotun, Delite: A compiler architecture for performance-oriented embedded domain-specific languages, ACM Transactions on Embedded Computing Systems 13 (4s) (2014) 134.

[33] G. Li, G. Gopalakrishnan, Scalable SMT-based verification of GPU kernel functions, in: Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '10), ACM, 2010, pp. 187–196.

[34] A. Betts, N. Chong, A. Donaldson, S. Qadeer, P. Thomson, GPUVerify: A verifier for GPU kernels, in: Proceedings of the 27th ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '12), ACM, 2012, pp. 113–132.

[35] A. Betts, N. Chong, A. F. Donaldson, J. Ketema, S. Qadeer, P. Thomson, J. Wickerson, The design and implementation of a verification technique for GPU kernels, ACM Transactions on Programming Languages and Systems 37 (3) (2015) 10.

[36] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, K. R. M. Leino, Boogie: A modular reusable verifier for object-oriented programs, in: Proceedings of the 4th International Symposium on Formal Methods for Components and Objects (FMCO '05), Vol. 4111 of LNCS, Springer, 2005, pp. 364–387.

[37] S. Blom, M. Huisman, M. Mihelčić, Specification and verification of GPGPU programs, Science of Computer Programming 95 (2014) 376–388.

[38] G. Li, P. Li, G. Sawaya, G. Gopalakrishnan, I. Ghosh, S. P. Rajan, GKLEE: Concolic verification and test generation for GPUs, in: Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '12), ACM, 2012, pp. 215–224.

[39] P. Collingbourne, C. Cadar, P. H. J. Kelly, Symbolic crosschecking of data-parallel floating-point code, IEEE Transactions on Software Engineering 40 (7) (2014) 710–737.

[40] S. West, S. Nanz, B. Meyer, Efficient and reasonable object-oriented concurrency, in: Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE '15), ACM, 2015, pp. 734–744.

[41] M. Schill, C. M. Poskitt, B. Meyer, An interference-free programming model for network objects, in: Proceedings

of the 18th IFIP International Conference on Coordination Models and Languages (COORDINATION '16), Vol. 9686 of LNCS, Springer, 2016, pp. 227–244.

[42] A. Kolesnichenko, C. M. Poskitt, B. Meyer, Applying search in an automatic contract-based testing tool, in: Proceedings of the 5th International Symposium on Search-Based Software Engineering (SSBSE 2013), Vol. 8084 of LNCS, Springer, 2013, pp. 318–323.