Diss. ETH No. 24006

Seamless Heterogeneous Computing: Combining GPGPU and Task Parallelism

A thesis submitted to attain the degree of
DOCTOR OF SCIENCES of ETH ZURICH
(Dr. sc. ETH Zurich)

presented by
Alexey Kolesnichenko
Specialist in Mathemathics and System Programming, Lomonosov Moscow State University, Russia

born on
January 17th, 1989

citizen of
Russia

accepted on the recommendation of

Prof. Dr. Bertrand Meyer, examiner
Dr. Judith Bishop, co-examiner
Prof. Dr. Torsten Hoefler, co-examiner
Dr. Christopher M. Poskitt, co-examiner

2016

Acknowledgments

I want to thank all the people who supported me through my PhD studies and helped me to finish this work.

My family, my parents, Sergey and Galina, who supported my decision to pursue a PhD study abroad and never doubted me, and my wife, Ksenia, who was always there for me.

Bertrand Meyer, who welcomed me to the Chair of Software Engineering, helped me with invaluable guidance and gave me a lot of freedom to pursue my ideas.

I want to thank Judith Bishop for an interesting collaboration and new inspirations and Torsten Hoefler who provided me with a place to finish my thesis and kindly agreed to be my second supervisor.

A special thanks to all colleagues (past and present) and friends. Sebastian Nanz, who mentored me and always had a great advises to share. Chris Poskitt, for being a terrific collaborator and always supporting me during the most tough deadlines. Andrey Rusakov, for interesting discussions and helpful insights. And to Nadia Polikarpova, Marco Trudel, Yu Pei, Yi Wei, Cristiano Calcagno, Georgiana Caltais, Claudia Günthart, Benjamin Morandi, Đurica Nikolič, Martin Nordio, Marco Piccioni, Mischael Schill, Jiwon Shin, Scott West, Christian Estler, Chandrakana Nandi, Stephan van Staden, Nikolai Tillmann, M. Purves.

Lastly, I want to thank committee members, who agreed to read and evaluate this thesis.

Financing

The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework

Programme (FP7/2007-2013) / ERC Grant agreement no. 291389

Contents

1 Introduction **1**

1.1 Background and Motivation 1

1.1.1 GPGPU Computing: Background 3

1.1.2 Task Parallelism: Background 4

1.2 Hypothesis 5

1.3 Contributions 6

1.4 Structure 8

1.5 Publication History 8

2 SafeGPU: Overview and Background **11**

2.1 Programming Style 11

2.1.1 CUDA Integration 12

2.2 Related Work 13

2.2.1 GPU Programming and Code Generation 13

3 Implementation and API **17**

3.1 Design of the API 17

3.1.1 CUDA Binding 18

3.1.2 Collections 19

3.1.3 Vectors 21

3.1.4 Matrices 23

3.2 Transferring Primitive and Class-Based Data 26

3.3 Translating Customized Program Logic 27

3.4 Deterministic Memory Management in Languages with Garbage Collection 29

3.5 Kernel Generation and Optimization 30

3.5.1 Kernel Generation and Data Transfer 30

3.5.2 Execution Plans and Kernel Optimization 31

3.5.3 Example: Gaussian Elimination 33

4 Contracts and Performance **37**
4.1 Design-by-Contract 37
4.2 Design-by-Contract in GPGPU 39
4.2.1 Contracts in SafeGPU 39
4.2.2 Example: Quicksort in SafeGPU 41
4.3 Evaluation 42
4.3.1 Performance 43
4.3.2 Code Size 44
4.3.3 Contract Overhead 46
4.4 Related Work 49
4.4.1 Correctness of GPU Kernels 49

5 SCOOP **51**
5.1 Overview of SCOOP Processors 52
5.2 Separate Calls 52
5.2.1 Reasoning in SCOOP 53
5.2.2 Passive Regions 55
5.3 Design-by-Contract in SCOOP 55

6 Task Parallelism Integration **57**
6.1 Introduction to Task Cancellation 57
6.2 Classification of Task Cancellation Techniques 58
6.2.1 Client-Based Cancellation 60
6.2.2 Supplier-Based Cancellation 62
6.2.3 Client/Supplier Combination 62
6.3 Cooperative Cancellation in SCOOP 64
6.3.1 Choosing a cancellation mechanism for SCOOP 64
6.3.2 SCOOP with cooperative cancellation 65
6.4 Asynchronous Event-Based Programming 68
6.4.1 Background 68
6.4.2 Asynchronous events in SCOOP 69
6.4.3 Example 72
6.5 Task Parallelism and GPGPU 74
6.6 Related Work 77

7 Conclusions **79**

Abstract

Concurrent and parallel computing is ubiquitous today. Modern computers are shipped with multicore CPUs and often equipped with powerful GPUs, which can be used for general-purpose computing (GPGPU) as well. CPUs have evolved into highly sophisticated devices, with many intricate performance-enhancing optimizations, that are targeted for task-parallel computations. However, the number of CPU cores is usually quite limited, and using CPU-based computations may not be the best fit for the data-centric parallelism. GPUs, on the other hand, feature hundreds and thousands of simpler cores, providing a viable alternative to CPUs for data-centric computations. In order to write efficient applications, programmers should be able to use these heterogeneous devices together.

The programming model for GPUs is quite different from traditional CPU-based programs. While there are some solutions for using GPGPU in high-level programming languages via various bindings by executing native code blocks, these solutions lack many aspects of high-level languages and are error-prone.

Task parallel computing on CPUs is much more challenging than sequential computing, with new kinds of bugs, both related to correctness and performance. Many of these issues are caused by programming models based on low-level concepts, such as threads, and lack of documentation on important aspects, such as task cancellation and subtle differences in semantics for similar APIs across languages.

In this thesis we explore two complementary approaches, which allow software developers to solve these problems within a flexible and general framework. We investigate how programmers can benefit from the advantages of CPUs and GPUs together, and within a high-level programming language.

For data-centric problems, we propose SafeGPU, a high-level library that abstracts away the technicalities of low-level GPU programming, while retaining the speed advantage. Our approach is modular: the user benefits from combining low-level primitives into high-level code. This modularity allows us to introduce an optimizer, which analyzes code blocks and produces optimized GPU routines. SafeGPU also integrates the design-by-contract methodology, which increases confidence in functional program correctness by embedding executable specifications into the program text. Finally, we show that with SafeGPU contracts can be monitored at runtime without diminishing the performance of the program, even with large amounts of data.

For task parallel problems, we propose an extension of SCOOP, a concurrent object-oriented programming model, in order to leverage its strong safety and reasoning guarantees. Our extension includes a safe mechanism for task cancellation and asynchronous event-based programming. We address the lack of documentation for task cancellation by providing a survey on existing techniques and classify them based on use cases. Finally, we show how our extended version of SCOOP cooperates with SafeGPU to provide a seamless heterogeneous experience for programmers.

ZUSAMMENFASSUNG

Parallelrechner sind heute allgegenwärtig. Moderne Computer sind mit Multicore-CPUs und oft mit leistungsstarken GPUs ausgestattet, welche ebenfalls für Allzweck-Berechnungen auf Grafikprozessoreinheiten (GPGPU) verwendet werden können. Die CPUs sind heute hochentwickelte Instrumente mit unzähligen leistungsverbessernden Optimierungen. Die Anzahl CPU Kerne ist jedoch normalerweise eher limitiert und CPU-basierte Rechnungsprozesse sind möglicherweise nicht die beste Lösung für datenzentrische Parallelverarbeitungen.

GPUs hingegen beinhalten Tausende von einfacheren Verarbeitungskernen und stellen hiermit für datenzentrische Parallelverarbeitungen eine geeignete Alternative zu CPUs dar. Das Programmiermodell für GPUs unterscheidet sich von traditionellen CPU-basierten Programmen. Zwar existieren verschiedene Lösungen für die Anwendung von GPGPU in höheren Programmiersprachen durch das Binding von ausführenden nativen Code-Blöcken, jedoch sind bei diesen viele Aspekte höherer Programmiersprachen vorenthalten. Dazu sind die meisten Lösungen fehleranfällig. Eine andere Quelle für Fehler sowie Performance-Probleme könnte der schlecht organisierte Task-Parallelismus sein.

In dieser Arbeit behandeln wir zwei komplementäre Ansätze, welche es Softwareentwicklern erlauben, diese Probleme zu lösen und ihnen eine flexible Methode zur Verfügung stellt, die in vielen unterschiedlichen Situationen anwendbar sein könnte. Wir untersuchen, wie Programmierer von den Vorteilen von CPUs und GPUs im Zusammenspiel und innerhalb einer höheren Programmiersprache profitieren können. Weiter erweitern wir unser Modell mit einem Task Cancellation-Verfahren, welches eine sichere und transparente Art darstellt, einen Task abzubrechen.

Wir beginnen damit, datenzentrische Probleme aufzugreifen und führen die SafeGPU ein, eine High-Level Library, welche die technischen Einzelheiten des low-level GPU Programmierens wegabstrahiert und gleichzeitig den Geschwindigkeitsvorteil beibehält. Unser Ansatz ist modular aufgebaut: der Benutzer profitiert von der Integration von low-level Primitiven in einen

Hochsprachen-Code. Diese Modularität erlaubt es uns, einen Optimierer einzuführen, welcher Code Blocks analysiert und optimierte GPU-Abläufe produziert.

SafeGPU integriert zudem die Design-By-Contract Methodologie, welche das Vertrauen in die funktionsgemässe Richtigkeit des Programms erhöht, indem ausführbare Spezifikationen im Programmtext eingebettet werden. Zuletzt zeigen wir, dass Runtime Contract-Checking in SafeGPU realisierbar wird, da die Contracts in der GPU ausgeführt werden können.

Als weiteren Aspekt unserer Doktorarbeit diskutieren wir die Task-basierten Probleme und untersuchen den wichtigen Fall der Task Cancellation, welcher für das gleichzeitige Ausführen mehrerer Tasks sehr bedeutsam ist. Wir schlagen eine Erweiterung für SCOOP vor (Simple Concurrent Object Oriented Programming Modell, als Teil der Eiffel Programmiersprache), welche einen sicheren und einfachen Weg für Task Cancellation darstellt und wir zeigen, dass dieser Mechanismus zusammen mit SafeGPU angewendet werden kann, um das Strukturieren rechnerischer Prozesse zu vereinfachen.

Chapter 1

Introduction

1.1 Background and Motivation

Modern day applications are very different from the ones that were written years before. They are operating with increasingly large volumes of data, development is shifting to Web and Mobile platforms, and users are very demanding when it comes to responsiveness and performance.

Since the requirements for performance are increasing, no modern application can afford to be single-threaded anymore. CPUs are no longer getting faster cores over time – instead they are being packed with more cores. Using parallel and concurrent computing is hence imperative for achieving these goals. Multiple CPU cores can help multi-threaded applications run faster (in the presence of heavy computations) and achieve a smoother user experience by offloading long computations from the UI thread and/or by processing another task while waiting for a long IO request.

CPU cores are highly sophisticated devices, featuring many intricate optimizations (multi-level caching, branch predictors, and vectorisation to mention a few) that are targeted for general-purpose computing, but they are no longer the only devices that can be used to achieve high performance these days. An alternative has emerged in the form of general-purpose computing on graphics processing units (GPGPU). GPUs have evolved into efficient computing devices, with many more cores than CPUs.

Using GPGPU for the right task (a canonical example is matrix-matrix multiplication) can lead to a massive performance boost compared to parallel computation on a CPU. These gains are so impressive that many new application domains are starting to use GPGPU. One of the most notable examples of GPGPU adoption can be found in mainstream frameworks for deep learning [2, 54].

Despite the need for concurrent and parallel applications on modern hardware architectures, writing them can still be challenging. The techniques that are widely used, such as threads, have been around for a long time and have not seen much evolution since their invention.

Many software developers are still using threads (or slight variations), even in major languages such as Java [55] and Python [16]. Not every developer knows the details of POSIX threads in C or their counterpart in their language of choice. But these details can be crucial for writing correct and performant programs: concurrent and parallel programming bring a whole new set of correctness and performance bugs, which were not present in the case of sequential programming. Concurrency bugs such as race conditions, atomicity violations, deadlocks, starvation, performance degradation can be hard to detect (unlike sequential case, bug occurrence can be non-deterministic), to debug (there is little to no support from the tooling) and to fix (one has to take concurrent aspects into account).

As for computing on graphical cards, despite the advantages, it does not come for free: the computation model is quite different from that for CPUs, mainly because GPUs are inherently single instruction, multiple data (SIMD) devices. Therefore, tools (compilers and debuggers), programming practices, and previous experience in programming languages cannot be applied to GPGPU computing. One has to use a special C-like language, such as CUDA, OpenCL or AMP [14, 31, 50], that exposes the SIMD computation model. GPGPU can also suffer from concurrency-like bugs, but specific to the context of the model. Errors such as incorrect barrier synchronization can make performance even worse – and performance is the principal reason for using the model.

In this thesis we explore how these two vastly different devices with their respective computing models can be used together to help developers achieve good performance of their programs and have reasonable guarantees for their correctness in a high-level language. Moreover, we attempt to push the usage of GPGPU computing beyond well-established domains, such as matrix math, into more general software engineering problems. We believe that a good solution should be modular in a sense that it allows to build more complex program block via combining primitive blocks – the property that is absent when using threading or CUDA programs. Another important idea behind our work is that programmers should be able to reason about their parallel programs as if they were sequential, and have confidence that programs are doing what they should and without paying too much attention to where the execution is actually happening (CPU or GPU). Lastly, the performance of our solution should be on par with a reasonable counterpart developed in a low-level languages. However, we do not expect to have an optimal performance in all cases, since we are not focusing on a single domain.

There is a need for a simple and modular approach which will help to implement these things in high level languages by bringing together all features of modern concurrency, but making it transparent and user friendly. The main goal of this approach should be to express all concepts in terms which high level developers are used to.

We focus on two contrasting, but complementary, approaches which help us to achieve the goals of allowing a wide spectrum of software developers (not necessary experts in parallel computing) to benefit from heterogeneous concurrency in their applications, expressed in a high-level language.

First, we develop a framework, SafeGPU, to simplify GPU computing, backed up by NVIDIA's CUDA, and make it more accessible to a wide range of software developers that would like to benefit from potential performance gains without investing additional effort to learn low-level device-dependent intricacies. The framework can be used alone or can benefit from integration with a task-based framework, for ease of orchestration and structuring computations. Second, we take advantage of SCOOP [49] to leverage its safety guarantees and simple reasoning for concurrent programming. We extend SCOOP with a task cancellation mechanism that is crucial for orchestrating non-trivial programs and solving a problem that arises frequently during concurrent programming.

We will show later in this thesis we can use aspects of task cancellation and GPGPU together to solve problems with heterogeneous concurrency.

We provide more background on these two approaches in the following.

1.1.1 GPGPU Computing: Background

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 [52, 72]. 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.

The CUDA [50] and OpenCL [31] 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 rather restrictive and low-level programming models. Recursion, for example, is among the standard programming concepts prohibited. Furthermore, automatic memory management is not there, meaning that programmers themselves must explicitly manage the allocation and de-allocation of memory and the movement of data, the potential source of many faults. While possibly acceptable for experienced GPU programmers, these issues act as a significant obstacle to the general programming public, less skilled in CUDA intricacies, preventing the wide adoption of GPGPU.

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 custom built—to support the orchestration of GPU computations, expressed in terms of programming patterns that leave the parallelism implicit [21, 22, 25, 43, 65]. 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 or the ability to auto-heterogenize [20, 29, 41, 42, 53, 58, 26].

These advances have made strides in the right direction, but the burden on the programmer can be lifted further. Some approaches (e.g., [53]) still require considerable understanding of relatively low-level GPU concepts such as barrier-based synchronization between threads; a mechanism that can easily lead to perplexing concurrency faults such as data races, atomicity violations 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., [20]) 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 [59].

1.1.2 Task Parallelism: Background

When talking about complex tasks which require CPU and GPU processing it is important to keep in mind the benefits and perils of task parallelism. Task parallelism frameworks are now flourishing on the market, but properly utilizing them requires considerable work and learning for developers: documentation is sometimes lacking and the API is not always clear, providing subtly ambiguous functions that may look similar, but feature very different runtime behavior.

We shall use SCOOP, a concurrent object-oriented programming model based on message-passing, which abstracts away low-level concepts of parallel programming such as threads, semaphores, monitors etc. Such an abstraction allows us to create a high level modular design which is one of our architectural goals.

The SCOOP model is based on the high-level concept of processors that own a number of objects, which may communicate within and across the boundaries of handlers. Concurrency happens only when there is communication across regions, which simplifies reasoning, compared to the threading model, because thread interference is limited. A set of call validity rules also guarantees freedom from atomicity violations and low-level data-races. We describe the model in more detail later in the thesis.

However, the SCOOP model lacks some aspects, important for efficient concurrent programming. One of these is task cancellation. Proper task cancellation contributes to the overall performance and safety of applications, allowing tasks to be ended in a consistent state, so one application can continue execution.

To choose an appropriate mechanism, we provide an overview of existing cancellation methods, analyzing techniques from different programming languages and concurrency libraries. This knowledge is then applied to provide a novel task cancellation technique for SCOOP that aligns with its design. The technique implemented is based on the idea of cooperative cancellation where both the canceling and the canceled task must cooperate in order to succeed.

1.2 Hypothesis

Programmers should easily benefit from the benefits of GPU and CPU based computations, depending on the problem.

GPGPU suits well for inherently parallel problems such as matrix computations, while CPUs can leverage its advanced optimizations such as branch predictions or caching to deal with task-based problems with complex control flow. Using both computational approaches appropriately should bring a noticeable performance gain to modern applications. Hence our research hypotheses:

- GPGPU can be done in a high-level language, without compromising too much performance compared to a native implementation, and with better safety guarantees. The programming experience can be made uniform with the language, encapsulating low-level aspects.

- Task-parallelism models can be extended to facilitate modern application design, allowing them to be used to complement GPGPU for a broad class of problems.

We prove the first hypothesis by introducing SafeGPU, describing its API and showing that performance is on par with CUDA. We prove the second hypothesis by extending SCOOP with a task cancellation model and asynchronous events, then using it together with SafeGPU.

1.3 Contributions

This thesis 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 to enable 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 [61], 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 performance gain 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 [44] methodology that is native to Eiffel (and provided by the Code Contracts library [24] 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 without diminishing the performance of the program, even with large amounts of data

Secondly, this thesis addresses the the challenges of task cancellation in task-parallel languages. Cancellable tasks are mainly used for interrupting long-running or outdated tasks, but the pattern can also be used as a building block for more high-level patterns, such as MapReduce, or it can be used directly to orchestrate building blocks in SafeGPU.

In this thesis we provide an overview of existing cancellation approaches, extracting techniques from different programming languages and concurrency libraries, classifying them, and discussing their strong and their weak points. This knowledge is then applied to provide a novel task cancellation technique for SCOOP [44, 49], an object-oriented concurrency model. The technique is based on the idea of cooperative cancellation where both the canceling and the canceled task must cooperate in order to succeed.

We further improve task cancellation in SCOOP by enhancing basic skeleton with event support, another important technique from the world of asynchronous programming, used in many applications, such as user interface programming or robotics systems[60]. We modify the synchronous event mechanism so that it can be used in the SCOOP environment and demonstrate by example combining the use of events and task cancellation.

Finally, we show how SafeGPU can benefit from the task cancellation and event-based programming: the flexibility of this combined approach is greatly increased, and the advantages of both approaches are maintained.

The contributions of this work therefore advance the state-of-the-art in approaches for task-based and data-centric parallel programming with the following:

- we created a library that embraces the *object-oriented paradigm*, shielding programmers from the low-level requirements of the CUDA model without depriving them of the performance benefits;

- the library is *modular* and *efficient*, supporting the programming of compound computations through the composition of primitive operations with a dedicated kernel optimization strategy;

- our framework supports the writing of *safe* and *functionally correct code* via contracts, monitored at runtime with little overhead;

- our approach offers initial support for transferring class-based data (i.e., beyond primitive data) to the GPU;

- we port the SafeGPU approach to C# and enhance it with Code Contracts, a library-based contract framework (in contrast to the natively supported contracts of Eiffel);

- we describe how the implementation of SafeGPU could be adopted to other managed languages, focusing on data transfer, translating customized program logic, and obtaining deterministic memory management;
- we survey task cancellation — a key aspect of task parallelism— and provide a novel taxonomy of task cancellation patterns across several major languages and concurrent libraries;
- we develop a novel approach for task cancellation for the SCOOP programming model, extend it with asynchronous event support, and show how it can be used together with SafeGPU

1.4 Structure

The rest of the thesis is organized as follows.

Chapter 2 provides an overview of the SafeGPU approach and its capabilities.

Chapter 3 discusses key aspects and challenges of the prototype implementation and explores the CUDA bindings and library API in more detail. This chapter also presents our kernel generation and optimization strategies.

Chapter 4 describes how design-by-contract is integrated into SafeGPU and evaluates performance, code size, and contract checking across a selection of benchmark programs.

Chapter 5 introduces the basics of SCOOP, a simple concurrent object oriented programming model.

Chapter 6 explores task cancellation and event-based programming in the context of SCOOP, and how they can be used together with SafeGPU.

Finally, Chapter 7, draws conclusions and proposes some future work.

1.5 Publication History

The results described in the thesis are based on the following publications. The author of the thesis is the first author, unless mentioned otherwise:

- SafeGPU: Contract-and Library-Based GPGPU for Object-Oriented Languages [35]
- Contract-Based General-Purpose GPU Programming [36]
- How to Cancel a Task [33]

- Concurrency patterns in SCOOP (Master thesis supervisor) [64]

Chapter 2

SafeGPU: Overview and Background

In this Chapter 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 3 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 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 2.1, 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 2.1: 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.

2.1.1 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 such as PyCUDA [32] and JCUDA [71], 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 thesis.

In Eiffel, the runtime integration of SafeGPU with CUDA (Figure 2.1) is achieved using Eiffel built-in mechanisms for interfacing with C++, allowing SafeGPU 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

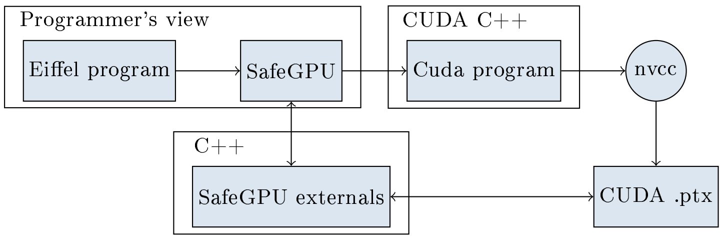


Figure 2.1: Runtime integration of CUDA with SafeGPU for Eiffel

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.

2.2 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 theme of SafeGPU: the *generation* of low-level GPU kernels from higher-level programming abstractions.

2.2.1 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 [21], Muesli [22], and SkelCL [65] 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 [25] (resp. Marrow [43]) for pipeline and farm (resp. pipeline, stream, loop), but concurrency and synchronization issues are exposed to the programmer. NVIDIA's C++ template library Thrust [51], 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 [71], Python [32]), 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 [58], 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 [20] 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 [42] 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 [41] 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 [69], for example, provides high-level abstractions for stream processing, and can be compiled to CUDA code via streaming-specific optimizations [29]; and VOBLA [9], 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 [67] 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 [53] 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.

Chapter 3

Implementation and API

In this chapter, we provide a detailed description of the SafeGPU API, focusing on design choices and highlighting the most important interfaces and methods.

After that 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 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.

Finally, we describe how SafeGPU translates individual API methods into CUDA kernels, how data is managed and how the the framework generates optimized code for compound operations.

3.1 Design of the API

In the following section, 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.

3.1.1 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 3.1 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.

Table 3.1: 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.

3.1.2 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 3.2 and described in the following. Note that in Eiffel, **like Current** denotes the type of the current object.

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. This is the main method to initialize the device memory with the data residing in RAM. Once the data resides on the device memory, `copy_from_separate` can be used to create inexpensive copies of the collection (or its descendants).

Unlike `from_array`, this method doesn't perform expensive `memcpy`-like copy internally (only *pointer* the device data is effectively copied), resulting in sharing the device data between two objects. This method parameter has **separate** type – serving to the purpose of SCOOP integration – enabling inexpensive construction of SafeGPU objects across processor boundaries. We discuss SCOOP integration in more detail in Section 6.5.

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 3). 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)
```

`from_array (array: ARRAY[T])`
: Creates an instance of a collection, containing items from the standard Eiffel array provided as input.

`copy_from_separate (other: separate like Current)`
: Creates a shallow copy of the other collection (or its descendant).

`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.

Table 3.2: Collection API

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`, 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 3.5).

3.1.3 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 3.3. It enables the computing of the average value `avg` and the `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

`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.

Table 3.3: Vector API: vector-only operations

sequential computations).

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 operator 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.

3.1.4 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 3.4. Table 3.5 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

`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 3.4: Matrix API: matrix-only 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.

Table 3.5: Matrix API: specialized collection operations

dimensions in matrix multiplication.

3.2 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, albeit 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 structure 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 3.1 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 3.1: A SafeGPU for C# method operating on data typed to a custom structure

3.3 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 3.1.2, 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 customization 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 [23]. 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 3.2 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 3.2: Example expression trees in C#

SafeGPU uses the framework to extract tree's 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 3.1.2 and Chapter 4.)

3.4 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 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++ avoids leaking resources by adopting the RAII (Resource Acquisition Is Initialization) idiom [66]. The essential idea is to use stack allocation and variable scope to ensure safe resource management. For example, in Listing 3.3, `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 3.3: 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 these substitutes 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 awareness is achieved by implementing a special interface or inheriting from a special base class.

In C#, the `IDisposable` interface is used to denote that an object implementing it might contain some unmanaged resource, 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 3.4: disposal is called whenever execution leaves the scope of the block. Java has `java.lang.AutoCloseable` and try-with-resources, which are 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 3.4: A using-block in C#

3.5 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.

3.5.1 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 2.1.1) 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 (Note that `copy_from_separate` does not perform an expensive copy: it only copies the pointer to the data, already residing in the device memory). 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.

3.5.2 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.

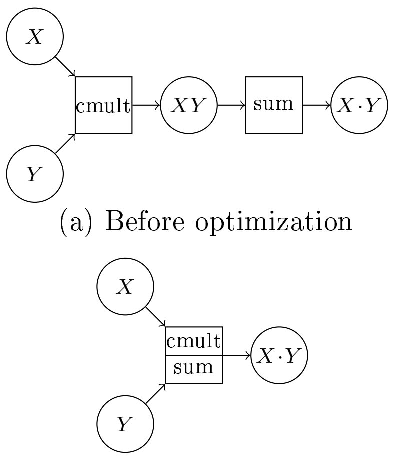
By default, a method call is not executed, but rather added to a list of pending actions for the corresponding collection. There are three ways to trigger its execution: (1) perform a function call that returns a scalar value, e.g., `sum`; (2) perform a call to `to_array` which creates a standard Eiffel 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 3.5. 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.

The benefit of deferring execution until necessary is that the kernel code

```
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 3.5: Combining primitives to compute the dot product in SafeGPU for Eiffel



(a) Before optimization

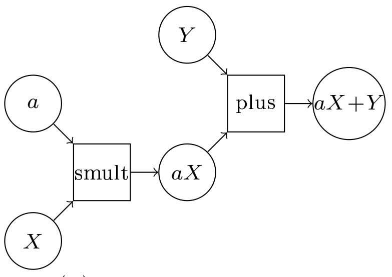
(b) After optimization

Figure 3.1: Execution plans for the dot product method

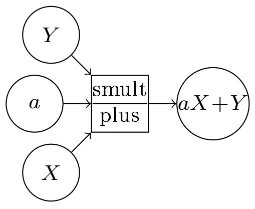
can be optimized. Instead of generating kernels for every method call, SafeGPU uses some simple strategies to merge deferred calls and thus handle the combined computation in fewer kernels. Before generating kernels, the optimizer constructs an execution plan from the pending operations. The plan takes 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.

We illustrate a typical optimization in Figure 3.1, which shows the execution plans for the dot product method of Listing 3.5. The plan in Figure 3.1a 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 3.1b, 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 3.2.



(a) Before optimization



(b) After optimization

Figure 3.2: 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.

3.5.3 Example: Gaussian Elimination

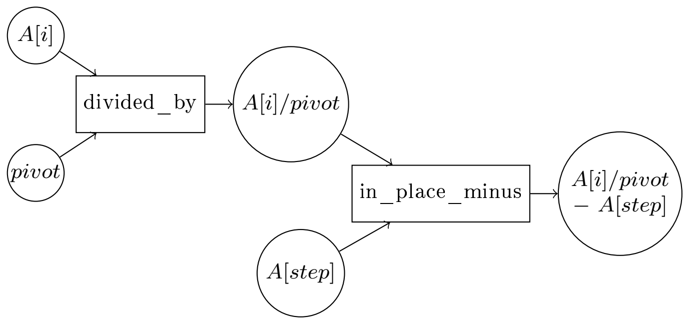
To illustrate the usefulness of the optimizer on a larger example, consider Listing 3.6, which provides an implementation of Gaussian elimination (i.e., for finding the determinant of a matrix) in SafeGPU. Note in particular the inner loop, which applies two transformations in sequence to a given row of the matrix:

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

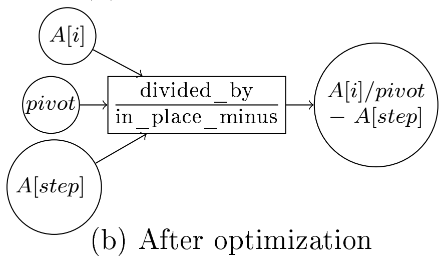
First, every element in the row is divided by a pivot (which an earlier check prevents from being zero); following this, another row of the matrix is subtracted from it in a component-wise fashion. The optimizer is able to combine these two steps into a single modified component-wise subtraction kernel, 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 3.3.

```
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 3.6: Gaussian elimination in SafeGPU for Eiffel



(a) Before optimization



(b) After optimization

Figure 3.3: Execution plans for the inner loop of Gaussian elimination

Chapter 4

Contracts and Performance

To support the development of safe and functionally correct code, SafeGPU integrates the design-by-contract methodology [44]. This methodology uses preconditions and postconditions to document (or programmatically assert) the change in state caused by a piece of a program.

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.

4.1 Design-by-Contract

The idea of design-by-contract is simple: if the execution of a task relies on a routine call to handle one of its subtasks, it is necessary to specify the relationship between the client (the caller) and the supplier (the called routine) as precisely as possible. This specification is done with assertions which can be any be of following types: preconditions, postconditions, class invariants.

Preconditions are conditions that must be true prior to running a method. Each method should have its own precondition. These conditions can apply to one of two things: data members or parameters. Usually, they apply to parameters.

While preconditions serve to let the caller of a method know when it's safe to call a method, postconditions let the caller of a method know what happened after calling the method. Postconditions typically indicate how the data members have changed, how the parameters passed in have changed and what the value of the return type is.

Preconditions and preconditions are parts of routine declarations as shown in Listing 4.2.

```
deposit(amount: INTEGER)
  -- Header comment
  require
    amount > 0 -- precondition
  do
    -- implementation omitted
  ensure
    Postcondition
  end
```

Listing 4.1: A routine equipped with assertions.

In this Eiffel notation, the **require** and **ensure** clauses (as well as the header comment) are optional. Each assertion is a list of Boolean expressions, separated by semicolons: here a semicolon is equivalent to a Boolean "and" but allows individual identification of the assertion clauses. These assertions can be monitored at runtime to help ensure the correctness of programs.

Class invariants are conditions that are true before and after running a routine (except constructors and destructors). Class invariants are useful because they help to avoid writing unnecessary code, and also help the programmer to think about what kind of behavior a class ought to have. A class invariant is a property that applies to all instances of the class, transcending particular routines.

```
initial-deposit: INTEGER;
deposits, withdrawals: TRANSACTION_LIST

record_deposit (d: INTEGER) is
  do
    Update the deposits list
  end -- record_deposit

balance: INTEGER is
    -- Current balance
  do
    balance := initial_deposit +
           deposits.sum -
           withdrawals.sum
  end --balance

invariant
  balance := initial_deposit + deposits.sum
        - withdrawals.sum
```

Listing 4.2: A routine equipped with assertions.

In practice, the question of when class invariants should hold can become quite complicated for certain object-oriented patterns [56]. Hence, we focus on preconditions and postconditions for the rest of the chapter.

4.2 Design-by-Contract in GPGPU

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 resides—without diminishing the performance of the program (see our benchmarks in Section 4.3.3).

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 [24]—which provides a number of advanced specification features including contracts for interfaces, abstract base classes, inheritance, and methods with multiple return statements (which is not permitted in Eiffel). Most importantly for SafeGPU, the library also provides runtime contract checking via a post-compilation step. A number of other object-oriented languages provide varios degrees of contract support, either via libraries (e.g., JML for Java 1.0 [13]), or natively (e.g., Spec# [8] or D [1]).

4.2.1 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 4.3, 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 4.3: 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 4.3.3).

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 design 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.

4.2.2 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.

```
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 4.4: Quicksort in SafeGPU for Eiffel

Listing 4.4 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, 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 4.5 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 3.3). 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.

```
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 4.5: Quicksort in SafeGPU for C#

4.3 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 [40]; and finally, matrix transposition came from a post [28] on NVIDIA's Parallel Forall blog.

The SafeGPU implementation and all the benchmarks are available to download online [61]. Listings for the SafeGPU implementations of quicksort and Gaussian elimination are also provided in this thesis (Listings 4.4 and 3.6, 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).

4.3.1 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. In the next experiment, we show the effect of keeping it. on.

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 4.1. 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.

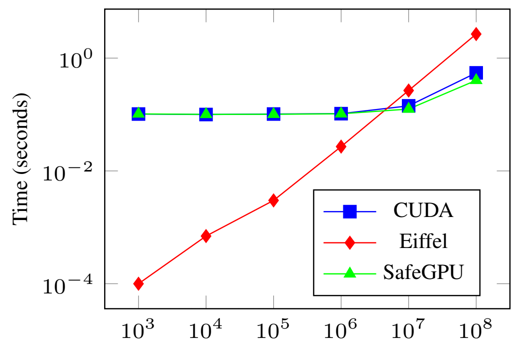
While sequential Eiffel is faster than SafeGPU and plain CUDA on relatively small inputs (as expected, due to the overhead of launching the GPU), it is outperformed by both when the size of the data becomes large. This happens particularly quickly for the non-linear algorithm (e) in comparison to the 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++.

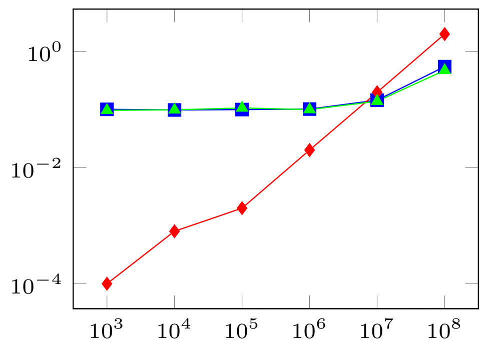
4.3.2 Code Size

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 [48], 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).

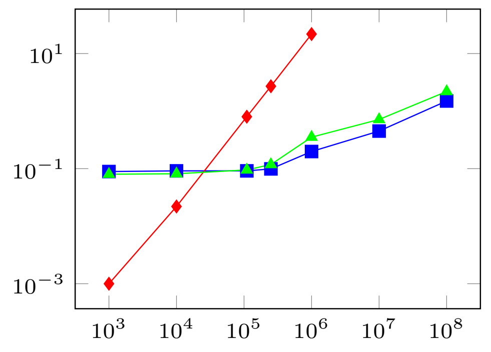
Our results are presented in Table 4.1. The programs written using our



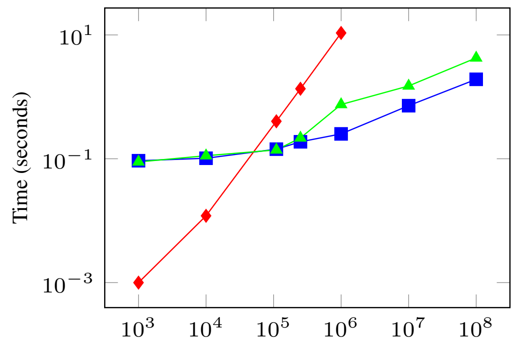
(a) Vector Addition



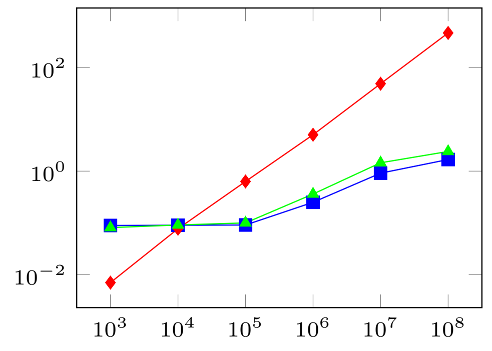
(b) Dot Product



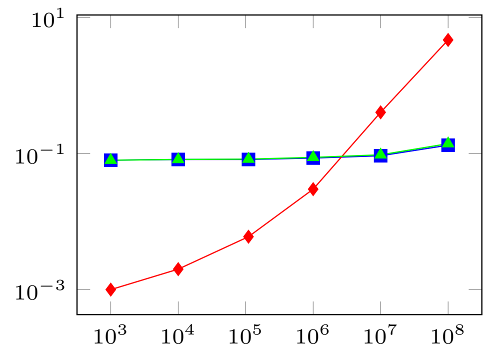
(c) Matrix-Matrix Multiplication



(d) Gaussian Elimination



(e) Quickstort



(f) Matrix Transpose

Figure 4.1: SafeGPU performance evaluation (x-axis: input size in no. of elements)

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.

4.3.3 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, 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 4.2, 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

Table 4.1: 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 4.2: 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

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.

4.4 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 theme of SafeGPU: the *correctness* of the kernels to be executed, and the maintenance of acceptable performance.

4.4.1 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 [38] and GPUVerify [10, 11] 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 [7].

Blom et al. [12] 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 [39] and KLEE-CL [15], both based on dynamic symbolic execution.

Chapter 5

SCOOP

So far we have explored how data-centric computations can utilize GPGPU to their benefit. We were using mostly sequential code that was running in parallel on graphical cards. Now we explore how to turn sequential bits of code (like in SafeGPU), into concurrent tasks by adopting a model for task-centric concurrency in an object-oriented setting. This compliments SafeGPU, by providing means to orchestrate executions with more flexibility (e.g. enabling cancellation of still running computations, or asynchronous progress notifications), that would not be possible within a pure data-centric model.

SCOOP [44, 49, 70], Simple Concurrent Object Oriented Programming model, is a concurrency model, based on message passing in the context of an object-oriented language. An extension, `D−SCOOP` [63] generilizes the model for distributed systems.

SCOOP aims to provide simple reasoning about concurrent execution, similar to sequential programming: inference-free reasoning over multiple (concurrent) objects, pre- and post-condition guarantees for blocks of code. SCOOP achieves this by encapsulating low-level aspects of concurrency programming (threads, semaphores, etc.), similar to how SafeGPU encapsulates CUDA code. With a minimal language change (only one new keyword – **`separate`**), SCOOP provides a higher-level API and strong reasoning guarantees which are ensured by the underlying implementation.

There are are two major execution models for SCOOP: backed by a single request queue (RQ) [47] or by a queue of queues (QoQ) [70]. While the QoQ implementation provides some noticeable gains in the case of heavy contention for objects, the reasoning behind this SCOOP variant is more involved. Since we are using the SCOOP model mainly for task-centric computing (with no heavy contentions for locks), we use the RQ variant in this chapter and for the rest of the thesis to simplify the discussion.

This chapter presents an overview of its most important aspects, focusing on the non-distributed version. We describe SCOOP's fundamental abstractions, reasoning guarantees, and show how they are applied to express concurrency in some simple examples.

5.1 Overview of SCOOP Processors

In order to achieve safe multitasking, SCOOP restricts concurrent access to shared memory. Thus the heap memory is partitioned into regions, which are individual sets of objects in the process memory.

Each object in SCOOP is associated with a *processor* (usually implemented as a thread). A processor is always attached to exactly one region, and is responsible for performing operations on all the objects it handles. Features of objects that reside on different processors can be executed in parallel.

Objects, potentially residing on different processors, must explicitly specify this possibility via the type system, using the keyword **`separate`**. To request method calls on objects of separate type, programmers must make the calls within so-called *separate blocks*. These blocks exist in one of the following forms:

- Method bodies that have at least one separate object as a formal argument
- An explicit separate block: `separate x as y do ... end`

Splitting the heap into several regions, where each region by itself is sequential and direct access from one region into another is not allowed (no uncontrolled communication via shared memory), gets rid of data races and atomicity violations.

5.2 Separate Calls

Regions in a concurrent SCOOP program can communicate via a variant of message passing: separate calls. Separate call is a feature call whose target is of a separate type. Listing 5.1 shows an example of simple separate calls.

In concurrent systems it is important to control resource accesses that can be shared among simultaneously executing processors. SCOOP, instead of using "critical sections", where only one thread is allowed to be executing

```
  do_call (foo: separate FOO; bar: separate BAR)
    do
      foo.do_something_async (11)
      foo.do_something_else_async
    end
```

Listing 5.1: Example of separate calls in SCOOP

the critical section at a time, relies on the mechanism of argument passing to assure exclusive access. As a result, there is a restriction placed on separate calls. A separate call on object x is valid if it is happening inside a separate block as defined above:

- inside a method body where x is a formal argument.
- inside a explicit separate block, listing x as a parameter.

In either of these cases we call the argument *controlled*. Examples of invalid and valid calls are shown in Listing 5.2.

```
my_separate_attribute: separate SOME_TYPE
another_separate_attribute: separate SOME_TYPE
                           ...
    calling_routine
            -- One routine
        do
            my_separate_attribute.some_feature    -- Invalid call: Feature
                call on separate attribute, which is not controlled
            enclosing_routine (my_separate_attribute) -- Separate attribute
                passed as argument
          separate another_separate_attribute as x
          do
            x.another_feature -- Valid call : inside a separate block.
            my_separate_attribute -- Invalid call: not a parameter of this
                separate block.
          end


        end

    enclosing_routine (a_arg: separate SOME_TYPE)
            -- Another routine
        do
            a_arg.some_feature     -- Valid call: Feature call on separate
                argument, which is controlled
        end
```

Listing 5.2: Example of valid and invalid separate calls

5.2.1 Reasoning in SCOOP

SCOOP restricts the order in which calls in separate blocks are executed, in order to help programmers reason about concurrency in their code and avoid subtle concurrency errors. Concretely, within a separate block (as defined above), method calls requested on separate objects are logged by their processors in the order that they appear in the program text, and no intervening requests will be logged from other processors. These guarantees are effective regardless of the number of separate objects and processors appearing within a separate block. Hence, SCOOP programs simplify reasoning about concurrency: the sequential reasoning applies to separate blocks, independently of the rest of the program.

SCOOP guarantees can be ensured via different runtime implementations. As stated before, we are using the RQ variant here for the sake of simplicity. In the RQ runtime each processor is associated with a single request queue. For a processor to be able to log a request on the queue of another processor, it must first acquire the lock protecting that queue. Once the lock is acquired, requests can be added to the queue without interference. Upon entering a **`separate`** `x, y, ...` block, the processor simultaneously acquires locks on the request queues associated with the processors of `x`, `y` and only releases them after the execution leaves the separate block. This solution successfully prevents interference from other processors that are not the part of the separate block. The canonical problem of the dining philosophers — where concurrent processes (philosophers) must acquire exclusive use of shared resources (forks) without causing cyclic deadlock — provides an example (in Listing 5.3) of how RQ guarantees can be used to write deadlock-free code.

```
eat (left, right: separate FORK)
    -- Eat, having acquired `left' and `right' forks.
  do
      -- Take forks.
    left.pick (Current)
    right.pick (Current)
      -- Eat.
      -- Put forks back.
    left.put (Current)
    right.put (Current)
  end
```

Listing 5.3: Dining philosophers in SCOOP

This code snippet solves the problem of dining philosopher without deadlocking because the locks on the request queues of the processors for `left` and `right` forks are acquired atomically, and no partial resource holding is possible.

5.2.2 Passive Regions

Regions can be made passive [46], meaning that there is no active processor associated with it. One can think of passive processors as a form of controlled shared memory, that doesn't violate SCOOP's reasoning guarantees.

Different processors can own this region by calling an object in it, but only one processor at the time (after acquiring control). The calling processor assumes the identity of the passive processor during the execution and ceases the identity once it is finished with the execution. After the region is adopted by a processor, calls become synchronous, which can help to reduce the number of threads and minimize runtime overhead.

One can declare an object with a passive region by using a specialized form of creation routine: `create <NONE> foo.make`.

5.3 Design-by-Contract in SCOOP

An important part of SCOOP is design-by-contract: preconditions, postconditions, and class invariants are frequently used to extend software interfaces into software specification.

The role of the precondition is somewhat different in SCOOP than in traditional sequential Eiffel. In non-concurrent Eiffel we view the precondition of a routine as defining a set of obligations on potential callers of the routine. That is, it is the set of conditions that must be true before correct execution of the routine can be expected. So, we could look at the precondition clauses in sequential Eiffel as correctness conditions. In SCOOP preconditions additionally take the role of a wait condition. Wait conditions are useful for cases where the caller can't guarantee that a property on an object is true at the time of the call, but it knows that it will eventually become true. In the case of a wait condition failure, the current processor will stall its execution, release the locks on its arguments, and wait until the precondition is fulfilled. The supplier processor signals to it once the condition could be re-checked.

The producer-consumer problem serves as an illustration of all aforementioned ideas. The main entities `producer`, `consumer`, and `buffer` are shown in Listing 5.4. The keyword **`separate`** specifies that the referenced objects may be handled by a processor different from the current one. A *creation instruction* on a separate entity such as `producer` will create an object on another processor; by default the instruction also creates that processor.

```
producer: separate PRODUCER
consumer: separate CONSUMER
buffer: separate BUFFER [INTEGER]

  consume (a_buffer: separate BUFFER [INTEGER])
  -- consume an item from the buffer
  require
    not (a_buffer.count = 0)
  local
    consumed_item: INTEGER
  do
    consumed_item := a_buffer.item
  end
```

Listing 5.4: Producer-consumer example in SCOOP

A consumer accesses an unbounded buffer in a feature call `a_buffer.item`. To ensure exclusive access, the consumer must lock the buffer before accessing it. Such locking requirements of a feature must be expressed in the formal argument list: any target of separate type within the feature must occur as a formal argument; the arguments' processors are locked for the duration of the feature execution, thus preventing data races. Such targets are called *controlled*. For instance, in `consume`, `a_buffer` is a formal argument; the consumer has exclusive access to the buffer while executing `consume`.

Condition synchronization relies on preconditions (after the **`require`** keyword) to express wait conditions. Any precondition makes the execution of the feature wait until the condition is true. For example, the precondition of `consume` delays the execution until the buffer is not empty.

Chapter 6

Task Parallelism Integration

Task parallelism is ubiquitous in modern applications for event-based, distributed, or reactive systems. In this type of programming, the ability to cancel a running task arises as a critical feature. Although a variety of cancellation techniques exist, a comprehensive account of their characteristics is missing. This thesis provides a classification of task cancellation patterns, as well as a detailed analysis of their advantages and disadvantages. One promising approach is cooperative cancellation, where threads must be continuously prepared for external cancellation requests. Based on this pattern, we propose an extension of SCOOP, an object-oriented concurrency model.

We further improve on the proposed technique by proposing a combination of cancellable tasks with asynchronous events and illustrate the usefulness of the combined usage of two models.

Finally, we illustrate how SafeGPU-based implementation can benefit from task parallelism to provide better control over the computation.

6.1 Introduction to Task Cancellation

Task parallelism has become part of a professional developerâĂŹs toolbox, and programming frameworks for this domain are sprouting up to help them express their intentions in a safe and concise manner. At the same time, learning to proficiently use such frameworks is far from easy. They offer a confusing variety of abstractions and constructs, often providing similar but subtly different functionality. Frequently, the only source of information is code examples where the relevance of the constructs cannot be sufficiently discussed. Too little research is spent on consolidating the various approaches by explaining commonalities and differences which would help developers learn to use new frameworks more quickly and aid designers in developing their frameworks further.

This thesis strives to address these deficiencies, focusing on a central problem in task parallelism: task cancellation techniques. Cancellable tasks are mainly used for interrupting long-running or outdated tasks, but the pattern can also be used as a building block for more high-level patterns, such as MapReduce. We provide an overview of existing cancellation approaches, extracting techniques from different programming languages and concurrency libraries, classifying them, and discussing their strong and their weak points. This knowledge is then applied to provide a novel task cancellation technique for SCOOP [44, 49], an object-oriented concurrency model. The technique is based on the idea of cooperative cancellation where both the canceling and the canceled task must cooperate in order to succeed.

We improve task cancellation in SCOOP by enhancing the basic skeleton with event support, another important technique from the world of asynchronous programming, used in many applications, such as user interface programming or robotics systems (Roboscoop [60], for example uses a combination of asynchronous events and task cancellation, inspired by this work). We modify the synchronous event so that they can be used in SCOOP environment and demonstrate on example the usefulness of combined use of events with task cancellation.

Finally, we show how SafeGPU can benefit from the task cancellation and event-based programming: the flexibility of combined approach is greatly increased, while keeping the advantages of both elements in the blend.

6.2 Classification of Task Cancellation Techniques

A *task* denotes the abstraction of an execution, such as a CPU thread, a thread pool, or a remote machine. The design of a programming model for task parallelism has to deal with the cancellation of tasks, a highly reusable pattern which can be applied to stand-alone applications, client-server systems, and distributed clusters alike. Without proper support for task cancellation, the developer has to write the synchronization code by hand, an task activity to subtle errors.

Various approaches to canceling a running task have been implemented in programming languages and libraries and described in theory. However, so far there is little evaluation and comparison of the proposed techniques. To provide a foundation for discussing them, we have examined a number of

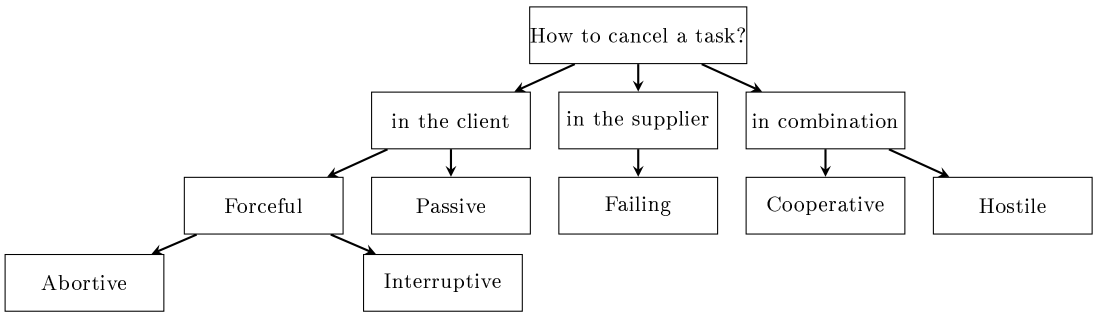


Figure 6.1: Execution plans for the dot product method

popular languages (Java, Python, C# TPL, Pthreads, etc.) and provide a taxonomy in Figure 6.1.

Client-based cancellation describes techniques where the control over the cancellation process lies entirely with the client (the canceling task):

- *Forceful cancellation* The client forces the supplier (the canceled task) to stop without the possibility to resist:
  - *Abortive cancellation* The supplier is terminated immediately.
  - *Interruptive cancellation* The supplier is allowed to reach a safe point before being terminated.
- *Passive cancellation* The client stops waiting for the result of a supplier, allowing it to continue on its own.

Supplier-based cancellation describes techniques where the control over the cancellation process lies entirely with the supplier:

- *Failing* The supplier encounters an unrecoverable error and needs to inform its clients.

Client/supplier combination describes techniques where client and supplier must act together in order to succeed with the cancellation:

- *Cooperative cancellation* The client asks the supplier to terminate, which decides itself how and when it should terminate.
- *Hostile cancellation* The supplier may resist a cancellation request of the client, and interrupt the client instead.

In the following, we discuss each of the approaches and provide examples of languages where they are employed.

6.2.1 Client-Based Cancellation

Abortive cancellation. Immediate termination does not give the canceled thread a chance to respond. As an example, consider the Java code in Listing 6.1, where `t.stop`() aborts the thread.

```
Thread t = new Thread(){ @override void run(){...} }
t.start();
...
t.stop(); // aborts the running thread
```

Listing 6.1: Aborting a thread-based task in Java

The advantage of the approach is clearly its simplicity. However, the approach is unsafe because aborting a running thread can leave a program in an inconsistent state. Consider the money transfer example in Listing 6.2.

```
void transfer(Account from, Account to, int amount) {
  synchronized{
    from.withdraw(amount); // if stopped here, money is lost
    to.deposit(amount);
  }
}
```

Listing 6.2: Unsafe cancellation using abortion

In this example, a synchronized block is used to guarantee that no thread interferes with the transfer. However, if a running thread is aborted during execution and is forced to unlock all of the monitors that it has locked, the transferred money may be lost and the remaining execution started in an inconsistent state. The Pthreads library [57] with `set_cancellalation mode` set to `PTHREAD_CANCEL_ASYNCHRONOUS` is a further example of abortive cancellation.

Interruptive cancellation. Using this technique, a running task is aware of potential interruption and usually cannot ignore it. However, a task cannot be canceled in every execution state, but only at so-called *safe points*: places where certain program invariants hold such that the execution may be interrupted safely. Usually a programmer must specify these places by hand, either by calling a library function, or handling a specific type of exception [19]. A special case of this technique allows interrupting a task at only one point in its lifetime: when the task has not been started yet. While it may seem not very useful, in some languages (Scala [62], Python [16]) this is the only built-in cancellation mechanism.

Consider the example in the Pthreads library[1] in Listing 6.3.

[1]Pthreads supports two cancellation modes: Deferred (as in the example) and Asynchronous. The latter one is an example of *aborting* tasks, with no safety guarantees.

```
// setting the cancellation mode to interruption
pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, NULL);
...
void* CancellablePthread(void* argument){
  ...
  pthread_testcancel(); // the execution can be safely canceled here
}
```

Listing 6.3: Cancellation points in Pthreads

At a safe point for cancellation, the call `pthread_testcancel()` checks on potential cancellation requests. Additionally, some of the blocking system calls are also considered to be cancellation points in Pthreads [57]. Java's thread interruptions[2] and `thread.Abort()` in C# (unlike the method's name is suggesting) are another examples of interruptive cancellation [30, 3].

Its potential safety guarantees are a benefit of this approach: *if* the approach is applied correctly, a program can be considered to be in a consistent state after a cancellation. Writing correct interruption-aware code is however difficult [55, 3] as a programmer has to remember subtle rules (e.g. in Pthreads some I/O calls are interruptible, others are not) and maintain a program's invariants by hand.

Passive cancellation. This technique is different from the forceful methods in that a canceling task does not need to become active: it simply stops waiting for a task result, while the running task is still being executed.

As an example, consider downloading a file over a network, illustrated in Listing 6.4 with C#'s Task Parallel Library (TPL). The call to a `StartDownload()` is asynchronous and returns only a handle to a future (an object, representing a computation that is still being computed [6]), represented by the `Task` class. After some time the downloader's result might not be needed anymore, i.e. the execution is abandoned (in the **`if`** branch).

```
void PassiveCancellation(string url){
  Downloader downloader = new Downloader(url);
  Task<byte[]> bytesFuture = downloader.StartDownload();
  ...
  if(noNeedToDownload) {
    // the download is not needed anymore
    return; // data is still being downloaded ...
  }
  else {
    // the download is still needed
    var result = bytesFuture.Result; // fetching the result
  }
}
```

Listing 6.4: Passive cancellation in the Task Parallel Library (TPL) of C#

[2]User-defined code may ignore interruption [55], but only between calls to library methods (which will not ignore it).

Obviously, this approach is not uniformly applicable; for example, we might still want to cancel a state-changing procedure. It is important to know in advance that the task will eventually be completed, i.e. listening to a TCP-socket cannot be canceled in this way. Another disadvantage is that the running task continues to consume machine resources. However, in a distributed setting this approach can find its application: consider a framework for a distributed computing, such as MapReduce. Often for the last piece of work several tasks are spawned [18] but only a single result will be used. In this case, there is no need to write sophisticated cancellation code, and it is valid to "forget" about the remaining executing tasks.

6.2.2 Supplier-Based Cancellation

This class of techniques deals with the special case that cancellation is not requested by a client but that a failure happens in the supplier, i.e. it cannot fulfill its obligations to clients; the supplier therefore needs to terminate. To indicate a failure, exceptions are typically used in object-oriented programming environments. Hence, this case boils down to the problem of exception handling in concurrent environments [45], which is not the focus of this thesis.

6.2.3 Client/Supplier Combination

Cooperative cancellation. A gentle way to stop a task is to cooperate and *ask* it to do so. The rationale for this approach is simple: a task is the abstraction of an execution, and hence should contain the information about how and when it should be stopped.

In other words, a task must be ready to be canceled at any time by external request. C#'s Task Parallel Library (TPL) follows this pattern, where a single point of cooperation is denoted by two classes: `CancellationTokenSource`, a generator of `CancellationToken`, which itself is a concrete request to cease the execution. An example is given in Listing 6.5.

This technique provides solid general structure for writing a cancellable tasks (see Listing 6.6), with a guarantee that no invariants will be violated. Unlike in interruptive cancellation, the programmer does not need to remember subtle rules of a concrete library or language. Cooperative cancellation also does not require any runtime support. Unfortunately, one cannot use the true power of this technique unless libraries support this pattern too (as far as we know, to date only limited support is introduced in C#). As another disadvantage, the latency between a cancellation request and actual cancellation is increased.

```
void Client(){
    var cts = new CancellationTokenSource();  // create the token source
    // pass the token to the cancelable operation
    Task.Run(() => Supplier(cts.Token));
    ...
    cts.Cancel(); // request cancellation
}
void Supplier(CancellationToken token) {
    for (int i = 0; i < 100000; i++) {
        // some work
        if (token.IsCancellationRequested) {
            break; // potentially perform cleanup, terminate
        }
    }
}
```

Listing 6.5: Cooperative cancellation in TPL

```
  void function run(CancelRequest cancel)
    while(not is_done){
      if cancel is requested
        exit
      loop_once
    }
  //need to be specified for concrete task.
  void function loop_once;
  boolean function is_done;
```

Listing 6.6: General structure of a cancellable task in cooperative cancellation

Hostile cancellation. While in the previous paragraph client and supplier are cooperating in order to succeed, in hostile techniques there is a struggle between the canceling and the canceled task. We describe these techniques on the example of *duels*, a mechanism which was theoretically described in [44]. We use the terminology from [44] in the rest of this chapter.

The key insight in this approach is that a canceling task (a "challenger", in the original) might not be strong enough to request an actual cancellation. If the canceling task is worthy enough, its request is fulfilled (the task is "killed"); if not, it gets an exception itself (therefore the approach is named a duel). In other words, dueling is a two-way interruption, where the result depends on which of the tasks is stronger.

To specify its preferences, a supplier can be in one of the two modes: either *retain* or *yield*. The former means that the task refuses to be canceled, and the latter specifies that it is OK to be interrupted. On the side of the client, there are also two options available: *demand* and *insist*. The first is more impatient, the second more gentle. The complete set of rules is shown in Table 6.1.

The dueling mechanism is useful in environments where executions are prioritized. For example, one can imagine a robotics system that needs to

	retain	*yield*
demand	the client is interrupted	the supplier is interrupted
insist	the client waits	the supplier is interrupted

Table 6.1: Dueling rules

handle simultaneously a variety of different tasks: route planning, controlling the motors, etc. These tasks can arise non-deterministically and compete for processing units, attempting to cancel other activities. However, it is completely unacceptable that a low priority task succeeds in canceling a more important one (for example, the data collection routine should not be able to cancel a task that adjusts the speed of the motors). In this case, a proper setup of dueling rules could both permit a cancellation request from high priority challengers and provide security from cancellation for important computational tasks.

6.3 Cooperative Cancellation in SCOOP

6.3.1 Choosing a cancellation mechanism for SCOOP

The main goal of SCOOP is to provide an easy-to-use model for expressing concurrency, with a focus on the correctness of the resulting programs. Any cancellation mechanism proposed for SCOOP must be designed in this spirit.

Clearly, *abortive cancellation* is an error-prone pattern, and it does not go well with SCOOP's focus on design-by-contract mechanisms. *Interruptive cancellation* has no strict correctness guarantees and can be complicated to use, which does not correspond to SCOOP's simplicity principle. In other concurrency models, where stricter techniques are not favored, interrupting may be a viable option. As mentioned, *passive cancellation* does not need to be explicitly implemented. As it is highly depended on particular usage scenarios, and has no guarantees that it will succeed (the termination of a passively canceled thread is not ensured), it also partly contradicts SCOOP design principles.

A concurrent object-oriented language needs to have well-defined rules about exception handling. The SCOOP implementation is discussed in [45, 70].

As a simple and safe approach, *cooperative cancellation* is a natural candidate to be implemented in SCOOP. It can be implemented using a library approach, thus even eliminating the need to modify the compiler. *Dueling* could be considered as an alternative to cooperative cancellation. However, while duels are only a good fit for specific scenarios, and less suitable in others, we prefer cooperative cancellation as a general-purpose approach.

6.3.2 SCOOP with cooperative cancellation

It is instructive to try to directly implement the approach introduced in Listing 6.6 in SCOOP. One can start with an abstract[3] class `CANCELLABLE_EXECUTOR` and introduce a `CANCEL_REQUEST` as a shared object that propagates a cancellation request; the descendants need to define the termination criteria in `is_done` and a single loop iteration `loop_once`.

An example of using this implementation of cooperative cancellation is shown in Listing 6.7. Unfortunately, this attempt does not work because in SCOOP a cross-processor call of `run` would force the executing processor to block on executing the loop for the entire execution. Thus all subsequent cancellation requests would be queued in the processor's request queue, effectively making `CANCELLABLE_EXECUTOR` useless. This happens because `CANCELLABLE_EXECUTOR` is both responsible for listening to cancellation requests and the execution itself.

```
executor: separate CONCRETE_CANCELLABLE_EXECUTOR
cancel: separate CANCEL_REQUEST --shared between two processors
...
-- launching an execution
executor.run (cancel) -- execution started on different processor

cancel.request -- canceling an execution
```

Listing 6.7: Usage of `CANCELLABLE_EXECUTOR`

This problem can be solved by decoupling the listening and the execution logic; the design is provided in Figure 6.2. The `CANCELLABLE_EXECUTOR` is now responsible only for listening for cancellation requests; the actual execution is handled by a different processor. To represent a concrete execution, a concretization of the deferred class `EXECUTION_UNIT` is needed. The `CANCEL_REQUEST` may not be actually **`separate`**, but keeping it this way provides additional flexibility for the case when a cancellation request is coming from a client residing on processor separate from `CANCELLABLE_EXECUTOR`.

In this design, `EXECUTION_UNIT` is a deferred class, only responsible for performing a single-loop iteration and termination criteria[4]. A task's life cycle is expressed in `CANCELLABLE_EXECUTOR` (see Listing 6.8, Listing 6.9), with the following methods:

[3]**`deferred`** in Eiffel notation.

[4]This functionality could also be implemented with Eiffel *agents* (function objects).

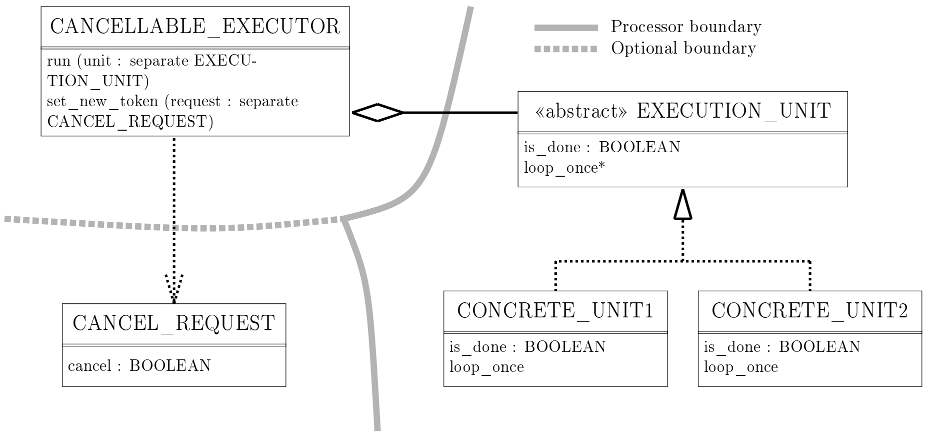


Figure 6.2: SCOOP cancellation design

- `make` creates an empty cancellation request. At this point, execution cannot be canceled from the outside.
- `set_new_token(a_token: separate CANCELLATION_REQUEST)` sets a new cancellation request, enabling a cancellation.
- `run(a_unit: separate EXECUTION_UNIT)` accepts the execution unit (where execution details are encapsulated) and starts the cancellation-aware execution, according to Listing 6.6.
- `loop_once(...)`, `is_done(...)` are wrapper methods[5] used to obtain control over separate objects, effectively preventing race conditions.
- `live` implements the listening for cancellation, from Listing 6.6.

As soon as a cancellation is requested, the loop body can be executed at most once more. After one cancellation request, the instance of `CANCEL_REQUEST` becomes useless, therefore we provide `set_new_token` to refresh a request as many times as needed.

```
class
  CANCELLABLE_EXECUTOR

create
  make

feature
  make
  do
```

[5]One can also use inline separate, a syntactic sugar to avoid writing wrapper methods like these.

```
      ...
  end

  set_new_token(a_token : separate CANCELLATION_REQUEST)
  do
     ...
  end

  run(a_unit: separate EXECUTION_UNIT)
  do
    live(a_unit)
  end
...
end
```

Listing 6.8: Cancellable executor interface

```
class
  CANCELLABLE_EXECUTOR
...
feature{NONE}
  live(a_unit : separate EXECUTION_UNIT)
  do
    from
    until
      is_done(a_unit)
    loop
      if(check_cancel_requested(token)) then
        cancel_requested := TRUE
      else
        loop_once(a_unit)
      end
    end

    cancel_requested:= FALSE
  end

  check_cancel_requested(a_token : separate CANCELLATION_REQUEST) : BOOLEAN
  do
    Result := a_token.cancellation_requested
  end

  is_done(a_exec : separate EXECUTION_UNIT) : BOOLEAN
  do
    if(cancel_requested) then
      Result := TRUE
    else
      Result := a_exec.is_done
    end
  end

  loop_once(a_exec : separate EXECUTION_UNIT)
  do
    a_exec.action;
  end

  cancel_requested : BOOLEAN
  token : separate CANCELLATION_REQUEST

end
```

Listing 6.9: Basic Cancellable executor implementation

6.4 Asynchronous Event-Based Programming

The solution presented in the previous section solves the problem of reliable task cancellation in SCOOP. In practice the clients of the `CANCELLABLE_EXECUTOR` (potentially residing on different process) would benefit from custom notifications, informing them about specific events of interest during the execution, such as execution progress. Using events [4] is one of the options to efficiently implement such notifications.

6.4.1 Background

Event-driven programming was introduced to provide a framework for writing Graphical User Interfaces (GUI) in applications and reactive systems, found for example in robotics. Contrary to classical console style when the main routine is blocked until user inputs required information, in event-driven application there is a main loop which listens to user produced events such as a button click, mouse hover, keyboard key press, or even messages from other threads or programs.

When an event is detected, the main loop triggers a callback function (or its high-level counterpart, such as an agent or delegate) which contains processing, specific to this event. This processing might be provided by a software developer in the event handler. Code for the main loop and event checking is not specific to any application and is frequently provided by programming frameworks. Several programming environments provide a number of event templates, allowing developers to concentrate their efforts on writing the event handlers' code.

Applications with event loops can be implemented in a single thread or region, but this is sub-optimal: long processing time of an event handler stops processing of the entire event queue (in the case of GUI program, which leads to interface "freeze" and poor responsiveness); any error in the event handler terminates the entire program. Introducing task parallelism into event loops alleviates these problems: event handlers can be processed on a separate region, in a non-blocking way, vastly improving the responsiveness of the main event loop and application. Errors that happen in a separate region become more localized and isolated, and in the case of non-fatal severity, would allow execution to continue, instead of immediately terminating the program.

6.4.2 Asynchronous events in SCOOP

We adapt the design of `EVENT_TYPE` from [4, 49] to enable asynchronous event-driven programming in the presence of cancellable tasks in SCOOP. The adapted version supports asynchronous subscription, cancelling a subscription and event publishing. The updated `EVENT_TYPE` is shown in Listing 6.10.

```
class EVENT_TYPE [D -> TUPLE]

inherit ANY
    redefine
      default_create
    end

feature {NONE} -- Initialization
        default_create -- Create an object with an empty list of actions.
    do
      create actions.make
      actions.compare_objects
    end

feature -- Subscription
        subscribe (an_action: separate PROCEDURE [ANY, D])
    -- Add `an_action' to the subscription list.
    require
      an_action_not_void: an_action /= Void
      an_action_not_already_subscribed: not has (an_action)
    do
      actions.extend (an_action)
    ensure
      an_action_added: actions.count = old actions.count + 1 and has (
          an_action)
      index_at_same_position: actions.index = old actions.index
    end

        unsubscribe (an_action: separate PROCEDURE [ANY, D])
      -- Remove `an_action' from the subscription list.
    require
      an_action_not_void: an_action /= Void
      an_action_already_subscribed: has (an_action)
    local
      pos: INTEGER
    do
      pos := actions.index
      actions.start
      actions.search (an_action)
      actions.remove
      actions.go_i_th (pos)
    ensure
      an_action_unsubscribed: actions.count = old actions.count - 1 and not
          has (an_action)
      index_at_same_position: actions.index = old actions.index
    end

        unsubscribe_all -- Clear subscription list.
    do
      actions.wipe_out
      actions.start
    ensure
      all_action_unsubscribed: actions.count = 0
    end

        has (an_action: separate PROCEDURE [ANY, D]): BOOLEAN
    -- Is `an_action' in the subscription list?
    do
      Result := actions.has (an_action)
    end

feature -- Publication
        publish (arguments: D)
    -- Publish all actions from the subscription list
    -- with `arguments' argument list.
    require
    arguments_not_void: arguments /= Void
    do
      across actions as iterator
      loop
        call_async (iterator.item, arguments)
      end
    end

feature{NONE} -- Implementation
        actions: LINKED_LIST[separate PROCEDURE [ANY, D]]
    -- List of subscribed actions.

        call_async (an_action: separate PROCEDURE [ANY, D]; data: D)
      -- Wrapper for calling `an_action' separately.
    do
      an_action.call(data)
    end
end
```

Listing 6.10: Asynchronous EVENT_TYPE

With the help of asynchronous events, we can enhance the previously presented task cancellation technique with custom event support. We enhance the `CANCELLABLE_EXECUTOR` with two new events, that are useful for its clients. We highlight differences at Listing 6.11.

```
class
  CANCELLABLE_EXECUTOR
create
  make

feature
  --New section for events.
  execution_cancelled : EVENT_TYPE [TUPLE] -- signals to its subscribers
      that execution was cancelled.
  execution_completed : EVENT_TYPE [TUPLE]-- signals to its subscribers that
       execution was completed.

  make
  do
    create token.empty
    create execution_completed
    create execution_cancelled
  end

  -- run, set_new_token are the same as in base version
feature{NONE}
```

```
  live(a_unit : separate EXECUTION_UNIT)
  do
    from
    until
      is_done(a_unit)
    loop
      if(check_cancel_requested(token)) then
        cancel_requested := TRUE
        -- notifies all subsribers that execution was cancelled.
        execution_cancelled.publish ([])
      else
        loop_once(a_unit)
      end
    end

    if(not cancel_requested) then
        -- notifies all subscribers that execution was successfully
            completed.
      execution_completed.publish ([])
    end
    cancel_requested:= FALSE
  end

  -- the rest is the same as in the base version.
end
```

Listing 6.11: Cancellable executor with events

In practice it is often needed to be informed about the progress of a task that is still running. This is especially important if a task takes a long time to run, so it is important to know whether it is progressing or has stalled for some reason. We implemented progress reporting by extending the `EXECUTION_UNIT` Listing 6.12. Note that the event arguments are defined by a template argument; this allows sufficient flexibility when dealing with different setups, potentially requiring different data for progress reporting.

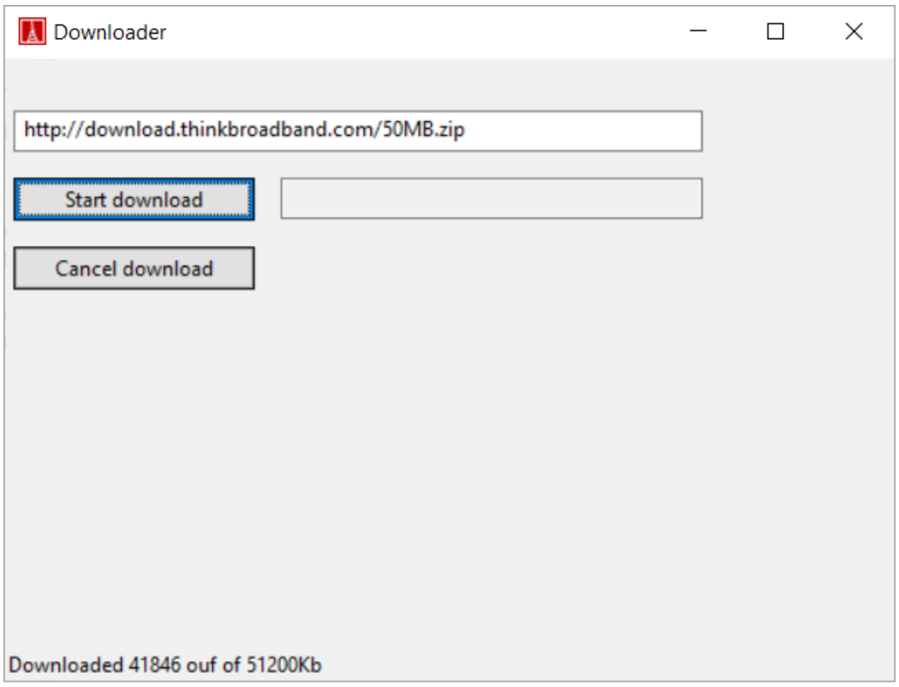
```
deferred class
  NOTIFYING_EXECUTION_UNIT[D -> TUPLE]

inherit
  EXECUTION_UNIT

feature
  execution_progress_changed : EVENT_TYPE [D]
  attribute
    create Result
  end

  action
  do
    internal_action
    execution_progress_changed.publish (progress_data)
  end

feature{NONE}
  progress_data : D
    deferred
  end
```

```
    internal_action
      deferred
    end
  end
```

Listing 6.12: Execution unit with progress reporting

6.4.3 *Example*

As an example of using cooperative cancellation in SCOOP, we present a downloader application that requests a URL, starts a background download process and provides progress reports. While still in progress, downloading can be canceled by the user. See Figure 6.3 for an illustration.



http://download.thinkbroadband.com/50MB.zip

Downloaded 41846 ouf of 51200Kb

Figure 6.3: Downloader application

The complete source code is available for download[6]; in the following description, we focus on key aspects of this application.

The `DOWNLOADER_UNIT`, responsible for downloading a single portion of bytes from a specified URL, is shown in Listing 6.13 (some code is omitted for brevity). Note that a **`separate STRING`** is required in the constructor to obtain control over it, as `DOWNLOADER_UNIT` resides on a different processor than its clients. The implementation is straightforward otherwise. Launching

[6]`https://github.com/xwat/tcl-eiffel`

an asynchronous download task is done in a pattern similar to Listing 6.7, applying the design in Section 6.3.2. One should create one CANCEL_REQUEST per launch: the cancel requests are designed to be used only once.

```
class DOWNLOADER_UNIT
inherit NOTIFYING_EXECUTION_UNIT[TUPLE[INTEGER, INTEGER]]
create
  make
feature

  make (a_url : separate STRING)
  do
    create http_downloder.make (create {HTTP_URL}.make(create {STRING}.
        make_from_separate(a_url)) )
    create parts.make
    http_downloder.set_read_mode
    http_downloder.open
    http_downloder.initiate_transfer
  end

  is_done :BOOLEAN
  do
    Result :=  http_downloder.bytes_transferred = http_downloder.count;
  end


feature{NONE}
  http_downloder : HTTP_PROTOCOL
  parts : LINKED_LIST[STRING]

  progress_data : TUPLE[INTEGER, INTEGER]
  do
    Result:= [bytes_2_kb(http_downloder.bytes_transferred), bytes_2_kb(
        http_downloder.count)]
  end

  internal_action
  do
    http_downloder.read

    if attached http_downloder.last_packet as last_p then
      parts.put_front(last_p)
    end

    if(is_done) then
      http_downloder.close
    end

  end

  bytes_2_kb(bytes : INTEGER) : INTEGER
  require
    non_negative : bytes >= 0
  do
    Result := bytes // 1024
  Ensure
    non_negative : Result >= 0
  end
end
```

Listing 6.13: Example: Downloader unit

6.5 Task Parallelism and GPGPU

In this section we provide an illustration of how GPGPU can be integrated with task parallelism, resulting in a program that uses SafeGPU, task cancellation and asynchronous events jointly. We illustrate this combination by extending the example of Gaussian elimination, introduced earlier in the thesis (See Listing 3.6). The basic version of Gaussian elimination is defined in a single function, useful for presentation but unfit for a task cancellation framework.

We start by introducing a class, `CANCELLABLE_GAUSS`, that accepts a **`separate`** `G_MATRIX[`**`DOUBLE`**`]` and sets up the initial state: initializing `determinant` and `step` with initial values. We also have added extra post-conditions to ensure the consistency of the state.

```
class CANCELLABLE_GAUSS
inherit NOTIFYING_EXECUTION_UNIT[DOUBLE]

create
  make_with_matrix

feature
  matrix : G_MATRIX[DOUBLE]
  determinant: DOUBLE
  cur_step : INTEGER

  make_with_matrix(a_matrix: separate G_MATRIX[DOUBLE])
    require
      -- Cannot write a_matrix.rows = a_matrix.columns
      -- since it would be a wait condition, not a correctness condition.
    local
      l_matrix : G_MATRIX[DOUBLE]
  do
    -- internally copies a pointer to device memory, no actual copy is
       happening.
    create l_matrix.copy_from_separate (a_matrix)
    init_state(l_matrix)
  ensure
    matrix = a_matrix -- this check is happening on GPU, and it is very fast
    cur_step = 0
    determinant = 1
  end
feature{NONE}
  init_state(a_matrix: G_MATRIX[DOUBLE])
  require
      a_matrix.rows = a_matrix.columns
      -- determinant of an empty matrix is not what we want to compute
      a_matrix.rows > 0
  do
    determinant := 1.0
    cur_step := 0
    matrix := a_matrix
  end
  ...
```

Listing 6.14: Cancellable gaussian elimination: setting up the state

Note that in SCOOP preconditions on separate arguments act as wait conditions and we can not write `require a_matrix.rows = a_matrix.columns` in the require clause of the creation routine – that would block infinitely in the case of non-square (or empty) matrix. Instead, we perform an inexpensive copy of `a_matrix` and delegate to internal method `init_state` to perform the correctness checks and initialization.

Next we need to implement event notification that would inform class clients about the progress and set the termination condition:

```
    ...
feature
  is_done
  do
    Result := step = matrix.rows
  end

feature{NONE}
  progress_data : DOUBLE
  do
    -- In Eiffel this is not integer division.
    Result := step / matrix.rows
  end
  ...
```

Listing 6.15: Cancellable gaussian elimination: termination

Finally, we implement the actual single-loop iteration for the Gaussian elimination process:

```
-- processes a single row
internal_action
  local
    i: INTEGER
    pivot: DOUBLE
  do
    loop
      pivot := matrix (step, step)
      determinant := determinant * pivot

      if not double_approx_equals (pivot, 0.0) then
        matrix.row (step).divided_by (pivot)
      else
        -- acts as premature exit.
        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
```

Listing 6.16: Cancellable gaussian elimination: inner loop

The implementation is stripped of the outer loop that is present at Listing 3.6, as the `NOTIFYING_EXECUTION_UNIT` implements it instead: `internal_action` is being called by `loop_once` method in the `CANCELLABLE_EXECUTOR` (see Listing 6.11) on each loop iteration. We defined the number of iterations by properly defining `is_done` (see Listing 6.15): "done" for this execution means that the current step of the gaussian eliminiation process is equal to the total number of rows in the matrix[7].

Finally, Listing 6.17 ties `CANCELLABLE_EXECUTOR` together with `CANCELLABLE_GAUSS` in a heterogeneous sample that illustrates how to set up event notification and launch the whole computation, that can be cancelled at ones convenience.

```
class GAUSS_USER
feature{NONE}
  executor : separate CANCELLABLE_EXECUTOR
  token : separate CANCELLATION_REQUEST
feature
  matrix: G_MATRIX[DOUBLE]
  -- notify subscribed clients about corresponding events during the
      execution.
  on_cancelled : EVENT_TYPE [TUPLE]
  on_completed : EVENT_TYPE [TUPLE]
  on_progress_changed : EVENT_TYPE [TUPLE [DOUBLE]]

  make(a_matrix: separate G_MATRIX[DOUBLE])
  do
    create token.empty
    create executor.make

    subscribe_executor (executor)

    create on_cancelled
    create on_completed
    create on_progress_changed
  end

  gauss_async
  do
    start_gauss(executor)
  end

  cancel
  do
    cancel_token(token)
  end

feature{NONE}
  start_gauss(a_executor : separate CANCELLABLE_EXECUTOR)
  local
```

[7]We can also get to that state also in the case of non-invertible matrix.

```
    new_token : separate CANCELLATION_REQUEST
    a_unit : separate CANCELLABLE_GAUSS
  do
    create a_unit.make(matrix)
    subscribe_unit(a_unit)
    create new_token.make
    token := new_token
    a_executor.set_new_token(token)
    a_executor.run(a_unit)

  end

  cancel_token(a_token : separate CANCELLATION_REQUEST)
  do
    a_token.request_cancellation
  end

  subscribe_unit(a_unit : separate DOWNLOADER_UNIT)
  do
    a_unit.execution_progress_changed.subscribe (agent on_progress_changed)
  end

  subscribe_executor(a_executor : separate CANCELLABLE_EXECUTOR)
  do
    a_executor.execution_cancelled.subscribe (agent on_cancelled)
    a_executor.execution_completed.subscribe (agent on_completed)

  end

  on_cancelled
  do
    on_cancelled.publish ([])
  end
  -- on_completed is the same as on_cancelled

  on_progress(progress: DOUBLE)
  do
    on_progress_changed.publish ([progress])
  end
end
```

Listing 6.17: Cancellable gaussian elimination: usage example

The combination of task parallelism and data-parallelism facilitated an elegant solution where the computationally-heavy parts are offloaded to the graphical card, and the whole computation remains responsive, can be cancelled at a safe point and is able to inform its clients about the progress.

6.6 Related Work

To the best of our knowledge, this work is the first to attempt a comprehensive classification and evaluation of task cancellation techniques and combined usage of GPGPU with task parallelism.

The work closest to task cancellation [55], Chapter 7, where some cancellation techniques are discussed for Java. In particular, cooperative cancellation with a shared variable or future is presented, along with rules to write a correct interrupt-aware code.

Further related work is also found in descriptions of individual techniques as part of language and library designs. The degree of support of task cancellation varies in such approaches. C# natively supports interruptive cancellation, and since the release of TPL, cooperative techniques were introduced [37]. C# also features an alternative to asynchronous events, in the form of asyncronous programming [5] which internally relies on TPL and method continuations, generated by the compiler.

Things get even more complicated when cancellation involves several tasks that need to agree on shutting down and terminate in a safe order. One approach that takes this into account is applied to OpenMP [68]. The authors introduce an abortive cancellation of already launched OpenMP tasks (which boils down to the cancellation mechanisms of Pthreads), with unrestricted possibility to cancel unstarted tasks. Their techique works on task groups, involving a child-parent relationship allowing cancellation of the whole group, starting from the root.

Python supports interruptive cancellation of non-started tasks via executors [16] and abortive cancellation of already started ones. Similiarly, Scala supports the `Cancellable` interface which allows canceling only non-started tasks [62].

Java supports interruptive cancellations natively [55]. The Pthreads library supports both abortion and interruption, depending on the setup [57].

Chapter 7

Conclusions

Parallel programming is hard. In this thesis we made an attempt to make it easier by exploring two different paths with a single goal in mind: a seamless heterogeneous experience, featuring both the flexibility of task-parallel computing and the computational power of data-parallel devices.

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.

SafeGPU provides a set of high-level abstractions, centered around concepts of generic collections and functional-style API, which facilitates modular design. Since SafeGPU exposes high-level API, the users are shielded from the perils of low-level GPGPU computing. Unlike CUDA programs, SafeGPU programs are concise and equipped with contracts, thereby contributing to program safety. To give flexibility to experienced GPGPU programmers SafeGPU also provides an alternative, low-level Binding API. 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.

To bring SafeGPU into a task-parallel world, we used the SCOOP model, that was extended in a number of ways to facilitate the development of asynchronous programs.

We classified techniques found in mainstream languages and parallel libraries based on the source where cancellation is actually happening. We extended SCOOP with a novel cancellation technique that does not compromise the safety guarantees of SCOOP and integrates. We adapted the synchronous version of events to SCOOP and enabled asynchronous event subscription and publication. We used the introduced mechanisms to develop several applications, such as non-blocking downloader GUI application, that reports about its progress. Finally, we illustrated how task parallelism can extend and complement SafeGPU with the example of cancellable Gaussian elimination, enabling more flexible programming approach.

This work can be extended in a variety of directions. In the current implementation, the optimizer in SafeGPU 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 4.3, 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. Some sort of latency hiding [27] might also complement a hybrid computing model.

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 [34]).

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.

Cooperative cancellation in SCOOP could be further extended to support task chaining (canceling an intermediate task causes all other tasks to be canceled) and precondition-aware tasks (these could effectively be deferred in the executing process). Another extension of this work could be to provide a formal model, for example based on graph transformation techniques [17], describing the control flow in different cancellation techniques.

In the thesis we used SafeGPU in combination with SCOOP, but internally, SafeGPU runs within a single SCOOP region. SafeGPU itself could be enhanced with task-parallel capabilities, potentially enabling interesting benefits such as wait-conditions evaluated on GPUs. With a recent extension, SCOOP supports distributed objects [63] which creates a possibility for a distributed version of SafeGPU, operating in a cluster of nodes, with some equipped with graphical cards, extending the proposed multi-GPU setup even more.

Bibliography

[1] Overview - D Programming Language. https://dlang.org/overview.html, accessed: November 2016.

[2] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Gregory S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian J. Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Józefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Gordon Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul A. Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda B. Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *CoRR*, abs/1603.04467, 2016.

[3] Joseph Albahari and Ben Albahari. *C# 3.0 in a Nutshell: A Desktop Quick Reference*. O'Reilly Media, Incorporated, 2007.

[4] Volkan Arslan, Piotr Nienaltowski, and Karine Arnout. *Event Library: An Object-Oriented Library for Event-Driven Design*, pages 174–183. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[5] Asynchronous Programming with Async and Await. https://msdn.microsoft.com/library/hh191443(vs.110).aspx, accessed: November 2016.

[6] Henry C. Baker, Jr. and Carl Hewitt. The incremental garbage collection of processes. In *Artificial Intelligence and Programming Languages*, pages 55–59. ACM, 1977.

[7] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-

oriented programs. In *Proceedings FMCO 2005*, volume 4111 of *LNCS*, pages 364–387. Springer, 2005.

[8] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The spec# programming system: An overview. In *Proceedings of the 2004 International Conference on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, CASSIS'04, pages 49–69, Berlin, Heidelberg, 2005. Springer-Verlag.

[9] Ulysse Beaugnon, Alexey Kravets, Sven van Haastregt, Riyadh Baghdadi, David Tweed, Javed Absar, and Anton 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)*, pages 115–124. ACM, 2014.

[10] Adam Betts, Nathan Chong, Alastair Donaldson, Shaz Qadeer, and Paul Thomson. GPUVerify: A verifier for GPU kernels. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '12)*, pages 113–132. ACM, 2012.

[11] Adam Betts, Nathan Chong, Alastair F. Donaldson, Jeroen Ketema, Shaz Qadeer, Paul Thomson, and John Wickerson. The design and implementation of a verification technique for GPU kernels. *ACM Transactions on Programming Languages and Systems*, 37(3):10, 2015.

[12] Stefan Blom, Marieke Huisman, and Matej Mihelčić. Specification and verification of GPGPU programs. *Science of Computer Programming*, 95:376–388, 2014.

[13] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 7(3):212–232, 2005.

[14] C++: AMP Overview. `https://msdn.microsoft.com/en-us/library/hh265136.aspx`, accessed: November 2016.

[15] Peter Collingbourne, Cristian Cadar, and Paul H. J. Kelly. Symbolic crosschecking of data-parallel floating-point code. *IEEE Transactions on Software Engineering*, 40(7):710–737, 2014.

[16] Concurrent futures in Python. `http://docs.python.org/dev/library/concurrent.futures.html`, accessed: November 2016.

[17] Claudio Corrodi, Alexander Heußner, and Christopher M. Poskitt. A graph-based semantics workbench for concurrent asynchronous programs. In *Proceedings International Conference on Fundamental Approaches to Software Engineering (FASE 2016)*, volume 9633 of *LNCS*, pages 31–48. Springer, 2016.

[18] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[19] Destroying Threads in C#. `http://msdn.microsoft.com/en-us/library/cyayh29d.aspx`, accessed: November 2016.

[20] Christophe Dubach, Perry Cheng, Rodric Rabbah, David F. Bacon, and Stephen 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)*, pages 1–12. ACM, 2012.

[21] Johan Enmyren and Christoph 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)*, pages 5–14. ACM, 2010.

[22] Steffen Ernsting and Herbert Kuchen. Algorithmic skeletons for multi-core, multi-GPU systems and clusters. *International Journal of High Performance Computing and Networking*, 7(2):129–138, 2012.

[23] Expression Trees. `https://msdn.microsoft.com/en-us/library/bb397951.aspx`, accessed: November 2016.

[24] Manuel Fähndrich, Michael Barnett, and Francesco Logozzo. Embedded contract languages. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 2103–2110. Association for Computing Machinery, Inc., 2010.

[25] Mehdi Goli and Horacio 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)*, pages 148–156. IEEE, 2013.

[26] T. Grosser and T. Hoefler. Polly-ACC: Transparent compilation to heterogeneous hardware. In *Proceedings of the the 30th International Conference on Supercomputing (ICS'16)*, ICS '16, pages 1:1–1:13, Jun. 2016.

[27] Tobias Gysi, Jeremia Bär, and Torsten Hoefler. dcuda: Hardware supported overlap of computation and communication. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SCâĂŹ16), accepted*, 2016.

[28] Mark Harris. An efficient matrix transpose in CUDA C/C++. `http://devblogs.nvidia.com/parallelforall/efficient-matrix-transpose-cuda-cc/`, accessed: November 2016.

[29] Amir H. Hormati, Mehrzad Samadi, Mark Woh, Trevor Mudge, and Scott 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)*, pages 381–392. ACM, 2011.

[30] Paul Hyde. *Java thread programming.* Sams Pub., 1999.

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

[32] Andreas Klöckner, Nicolas Pinto, Yunsup Lee, Bryan C. Catanzaro, Paul Ivanov, and Ahmed Fasih. PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation. *Parallel Computing*, 38(3):157–174, 2012.

[33] Alexey Kolesnichenko, Sebastian Nanz, and Bertrand Meyer. How to cancel a task. In *Proceedings of the 2013 International Conference on Multicore Software Engineering, Performance, and Tools (MUSEPAT'13)*, Lecture Notes in Computer Science, pages 61–72. Springer, 2013.

[34] Alexey Kolesnichenko, Christopher M. Poskitt, and Bertrand Meyer. Applying search in an automatic contract-based testing tool. In *Proceedings International Symposium on Search-Based Software Engineering (SSBSE 2013)*, volume 8084 of *LNCS*, pages 318–323. Springer, 2013.

[35] Alexey Kolesnichenko, Christopher M. Poskitt, and Sebastian Nanz. SafeGPU: Contract- and library-based GPGPU for object-oriented languages. *Computer Languages, Systems & Structures*, pages 1 – 21, 2016.

[36] Alexey Kolesnichenko, Christopher M. Poskitt, Sebastian Nanz, and Bertrand Meyer. Contract-based general-purpose GPU programming. In *Proceedings International Conference on Generative Programming: Concepts and Experiences (GPCE 2015)*, pages 75–84. ACM, 2015.

[37] Daan Leijen, Wolfram Schulte, and Sebastian Burckhardt. The design of a task parallel library. *Acm Sigplan Notices*, 44(10):227–242, 2009.

[38] Guodong Li and Ganesh 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)*, pages 187–196. ACM, 2010.

[39] Guodong Li, Peng Li, Geof Sawaya, Ganesh Gopalakrishnan, Indradeep Ghosh, and Sreeranga 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)*, pages 215–224. ACM, 2012.

[40] Linear Algebra: Gaussian Elimination. `http://www.cs.rutgers.edu/~venugopa/parallel_summer2012/ge.html`, accessed: November 2016.

[41] Geoffrey Mainland and Greg Morrisett. Nikola: Embedding compiled GPU functions in Haskell. In *Proceedings of the 3rd ACM SIGPLAN Symposium on Haskell (Haskell '10)*, pages 67–78. ACM, 2010.

[42] Dmitri Makarov and Matthias 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)*, pages 109–112. ACM, 2015.

[43] Ricardo Marqués, Hervé Paulino, Fernando Alexandre, and Pedro D. Medeiros. Algorithmic skeleton framework for the orchestration of GPU computations. In *Proceedings of the 19th International Conference on Parallel Processing (Euro-Par '13)*, volume 8097 of *LNCS*, pages 874–885. Springer, 2013.

[44] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice-Hall, 2nd edition, 1997.

[45] Benjamin Morandi, Sebastian Nanz, and Bertrand Meyer. Who is accountable for asynchronous exceptions? In *APSEC'12*, pages 462–471. IEEE Computer Society, 2012.

[46] Benjamin Morandi, Sebastian Nanz, and Bertrand Meyer. Safe and efficient data sharing for message-passing concurrency. In *International Conference on Coordination Languages and Models*, pages 99–114. Springer, 2014.

[47] Benjamin Morandi, Mischael Schill, Sebastian Nanz, and Bertrand Meyer. Prototyping a concurrency model. In *2013 13th International Conference on Application of Concurrency to System Design*, pages 170–179. IEEE, 2013.

[48] Sebastian Nanz and Carlo A. Furia. A comparative study of programming languages in Rosetta Code. In *Proceedings of the 37th International Conference on Software Engineering (ICSE '15)*, pages 778–788. IEEE, 2015.

[49] Piotr Nienaltowski. *Practical framework for contract-based concurrent object-oriented programming.* PhD thesis, ETH Zurich, 2007.

[50] NVIDIA: CUDA Parallel Computing Platform. `http://www.nvidia.com/object/cuda_home_new.html`, accessed: November 2016.

[51] NVIDIA: CUDA Toolkit Documentation – Thrust. `http://docs.nvidia.com/cuda/thrust/`, accessed: November 2016.

[52] NVIDIA: GPU Applications. `http://www.nvidia.com/object/gpu-applications.html`, accessed: November 2016.

[53] Nathaniel Nystrom, Derek White, and Kishen Das. Firepile: Run-time compilation for GPUs in Scala. In *Proceedings of the 10th International Conference on Generative Programming and Component Engineering (GPCE '11)*, pages 107–116. ACM, 2011.

[54] Caffe Overview. `http://caffe.berkeleyvision.org/`, accessed: November 2016.

[55] Tim Peierls, Brian Goetz, Joshua Bloch, Joseph Bowbeer, Doug Lea, and David Holmes. *Java Concurrency in Practice.* Addison-Wesley, 2005.

[56] Nadia Polikarpova, Julian Tschannen, Carlo A. Furia, and Bertrand Meyer. Flexible invariants through semantic collaboration. In *FM 2014: Formal Methods - 19th International Symposium, Singapore, May 12-16, 2014. Proceedings*, pages 514–530, 2014.

[57] POSIX threads specification. `http://man7.org/linux/man-pages/man7/pthreads.7.html`, accessed: November 2016.

[58] Philip C. Pratt-Szeliga, James W. Fawcett, and Roy 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)*, pages 375–380. IEEE, 2012.

[59] Tiark Rompf, Kevin J. Brown, HyoukJoong Lee, Arvind K. Sujeeth, Manohar Jonnalagedda, Nada Amin, Georg Ofenbeck, Alen Stojanov, Yannis Klonatos, Mohammad Dashti, Christoph Koch, Markus Püschel, and Kunle 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)*, volume 32 of *LIPIcs*, pages 238–261. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.

[60] Andrey Rusakov, Jiwon Shin, and Bertrand Meyer. Simple concurrency for robotics with the roboscoop framework. In *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1563–1569. IEEE, 2014.

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

[62] Scala Scheduler. `http://doc.akka.io/docs/akka/snapshot/scala/scheduler.html`, accessed: November 2016.

[63] Mischael Schill, Christopher M. Poskitt, and Bertrand Meyer. An interference-free programming model for network objects. In *Proceedings of the 18th IFIP International Conference on Coordination Models and Languages (COORDINATION '16)*, volume 9686 of *LNCS*, pages 227–244. Springer, 2016.

[64] Roman Schmocker and Alexey Kolesnichenko. Concurrency patterns in SCOOP. Master's thesis, ETH-Zürich, 2014.

[65] Michel Steuwer and Sergei 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)*, volume 7979 of *LNCS*, pages 258–272. Springer, 2013.

[66] Bjarne Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, 1994.

[67] Arvind K. Sujeeth, Kevin J. Brown, Hyoukjoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. Delite: A compiler architecture for performance-oriented embedded domain-specific languages. *ACM Transactions on Embedded Computing Systems*, 13(4s):134, 2014.

[68] Oussama Tahan, Mats Brorsson, and Mohamed Shawky. Introducing task cancellation to OpenMP. In *IWOMP'12*, pages 73–87. Springer-Verlag, 2012.

[69] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. StreamIt: A language for streaming applications. In *Proceedings of the 11th International Conference on Compiler Construction (CC '02)*, volume 2304 of *LNCS*, pages 179–196. Springer, 2002.

[70] Scott West, Sebastian Nanz, and Bertrand 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)*, pages 273–274. ACM, 2015.

[71] Yonghong Yan, Max Grossman, and Vivek 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)*, volume 5704 of *LNCS*, pages 887–899. Springer, 2009.

[72] Shin Yoo, Mark Harman, and Shmuel Ur. GPGPU test suite minimisation: search based software engineering performance improvement using graphics cards. *Empirical Software Engineering*, 18(3):550–593, 2013.

List of Tables

3.1 Binding API 18
3.2 Collection API 20
3.3 Vector API: vector-only operations 22
3.4 Matrix API: matrix-only operations 24
3.5 Matrix API: specialized collection operations 25

4.1 LOC comparison 47
4.2 Contract checking overhead comparison 48

6.1 Dueling rules 64

Listings

2.1 Transposed matrix-vector multiplication in SafeGPU for Eiffel 12
3.1 A SafeGPU for C# method operating on data typed to a custom structure 27
3.2 Example expression trees in C# 28
3.3 A possible RAII idiom usage 30
3.4 A using-block in C# 30
3.5 Combining primitives to compute the dot product in SafeGPU for Eiffel 32
3.6 Gaussian elimination in SafeGPU for Eiffel 34
4.1 A routine equipped with assertions. 38
4.2 A routine equipped with assertions. 38
4.3 Contracts for component-wise vector division in SafeGPU for Eiffel 40
4.4 Quicksort in SafeGPU for Eiffel 41
4.5 Quicksort in SafeGPU for C# 42
5.1 Example of separate calls in SCOOP 53
5.2 Example of valid and invalid separate calls 53
5.3 Dining philosophers in SCOOP 54
5.4 Producer-consumer example in SCOOP 55
6.1 Aborting a thread-based task in Java 60
6.2 Unsafe cancellation using abortion 60
6.3 Cancellation points in Pthreads 61
6.4 Passive cancellation in the Task Parallel Library (TPL) of C# 61
6.5 Cooperative cancellation in TPL 63
6.6 General structure of a cancellable task in cooperative cancellation 63
6.7 Usage of `CANCELLABLE_EXECUTOR` 65
6.8 Cancellable executor interface 66
6.9 Basic Cancellable executor implementation 67
6.10 Asynchronous EVENT_TYPE 69
6.11 Cancellable executor with events 70

6.12 Execution unit with progress reporting 71
6.13 Example: Downloader unit 73
6.14 Cancellable gaussian elimination: setting up the state 74
6.15 Cancellable gaussian elimination: termination 75
6.16 Cancellable gaussian elimination: inner loop 75
6.17 Cancellable gaussian elimination: usage example 76

List of Figures

2.1 Runtime integration of CUDA with SafeGPU for Eiffel 13

3.1 Execution plans for the dot product method 32
3.2 Execution plans for vector mathematics 33
3.3 Execution plans for the inner loop of Gaussian elimination . . 35

4.1 SafeGPU performance evaluation (x-axis: input size in no. of elements) 45

6.1 Execution plans for the dot product method 59
6.2 SCOOP cancellation design 66
6.3 Downloader application 72