CuSharp

A GPU Compute Framework for .NET

Project Type:	Bachelor Thesis
Project Team:	Adrian Locher Jason Benz
Project Advisor:	Philipp Kramer
Expert:	Christian Marrocco
Proofreader:	Thomas Corbat
Linguistic Proofreader:	AnneMarie O'Neill
Date:	15.06.2023

OST

Eastern Switzerland
University of Applied Sciences

Abstract

The number of computationally intensive applications is growing. For many easily parallelizable problems, GPUs offer better performance than CPUs. As a result, GPUs are now used not only for graphical applications, but also for machine learning and cryptography. GPU-accelerated programs have traditionally been written in C, C++ for high-performance applications such as physics simulations and graphical applications and more recently in Python to optimize machine learning algorithms. Most GPU-APIs, including Nvidia CUDA, restrict their developers to using C, C++ or Python to write programs targeting those APIs.

In this thesis a framework called *CuSharp* has been developed that allows developers to build and run GPU-executable kernels directly in C#. This is achieved by using existing toolchains complemented by a specifically developed cross-compiler. In a first step, the Roslyn compiler is used to compile the C# kernel to Microsoft Intermediate Language (MSIL). Subsequently, the CuSharp compiler cross-compiles MSIL to NVVM IR, which is a low-level but platform-independent intermediate representation. Finally, NVVM IR is translated to PTX ISA, an assembly-like language for Nvidia GPUs, using the NVVM compiler library (libNVVM). By further encapsulating all device-specific compiler settings, we allow future development efforts to add support for devices other than those manufactured by Nvidia.

CuSharp supports the compilation of static methods, written in a specific C# subset, either just-in-time or ahead-of-time. The resulting PTX ISA kernels' performance was benchmarked and compared to kernels compiled by the Nvidia CUDA Compiler (NVCC). For a kernel that computes matrix multiplications, a performance slowdown between 1.4% and 4.8% was measured for CuSharp-compiled kernels compared to NVCC-compiled kernels. The measurements refer to the execution time of the kernel on the graphics card, excluding the compilation process and the data transfers.

This thesis shows the challenges of interfacing with the LLVM compiler infrastructure and the Nvidia CUDA API. In addition, it provides an overview of the complex landscape of APIs that can be used to interface with GPU devices in general, by comparing their toolchains and languages. Furthermore, it demonstrates that GPU kernels can be written in a high-level language such as C#, which is widely used in the industry, while suffering only minor performance degradation.

Lay Summary

For data-parallel computations, graphics processing units (GPUs) offer better performance than central processing units (CPUs). To perform computations on GPUs, programs must be compiled into a specific language and computational model.

Many compilers exist for languages such as C, C++ and Python, but there is a general lack of such tools for C#, a language widely used in the industry.

By combining existing tools and complementing them with our own compiler, we allow developers to write and run GPU programs, called *kernels*, directly in C#.

Using CuSharp (the result of this thesis) C# developers can write fast, computationally intensive applications, without having to learn new programming languages. This makes development faster and easier by allowing developers to focus on the problem they are solving instead of worrying about technical details.

CuSharp, as of now, supports devices manufactured by Nvidia, one of the market leading GPU manufacturers. Although CuSharp currently only supports devices from one manufacturer, it is extensible to support devices from other manufacturers.

Acknowledgements

We would like to acknowledge the following people for their valuable contributions to this bachelor thesis.

Firstly, we would like to thank Philipp Kramer for his advice and inputs during our weekly meetings. We would also like to thank Thomas Corbat and Christian Marrocco for their insightful feedback during our interim presentation.

Finally, we express our gratitude to AnneMarie O'Neill for her thorough proofreading.

Glossary

Name	Abbr.	Description
Abstract Syntax Tree	AST	Tree representation of the syntactic structure of a program.
Ahead of Time Compilation	AOT Compilation	Process to compile code before it is executed.
Common Intermediate Language	CIL	Synonym for MSIL.
Compute Unified Device Architecture	CUDA	Platform for parallel computing and a toolkit that allows to run code directly on Nvidia graphics cards.
Control Flow Graph	CFG	Program or function representation as a directed graph which displays the basic blocks and the flow of control between them.
Graphics Processing Unit	GPU	Specialized processor to render graphics and process data simultaneously.
Heterogeneous-Compute Interface for Portability	HIP	C++ runtime API and programming model used to write GPU-accelerated code that can be executed on AMD and Nvidia GPUs.
High Level Shading Language	HLSL	C-like programming language used to write shaders using Microsoft's DirectX API.
Intermediate Representation	IR	Low-level and platform-independent representation of code used by compilers to optimize and transform code.
Just in Time Compilation	JIT Compilation	Process to compile code during runtime.
Kernel	-	A parallel method or function that performs computation on a large amount of data on GPUs, often in a SIMD style.
Low-Level Virtual Machine Compiler Infrastructure	LLVM	Collection of platform-independent compiler and tool-chain technologies.
-	LLVM IR	Low-level and platform-independent representation used as an intermediate step during compilation in the LLVM compiler infrastructure.
Microsoft Intermediate Language	MSIL	A platform-independent instruction set which is the output of compilers in .NET based languages.
Nvidia	-	Large company from the USA that develops GPU technologies and which is specialized in artificial intelligence computing.
Nvidia CUDA Compiler	NVCC	Compiler used to compile CUDA code (written in a CUDA supported language such as C, C++ etc.) to PTX.

Nvidia Virtual Machine IR	NVVM IR	Low-level and platform-independent IR used as input for NVCC to generate PTX. It is based on a subset of LLVM IR.
OpenGL Shading Language	GLSL	C-like programming language used to write shaders using the OpenCL, OpenGL or Vulkan API.
Parallel Thread Execution Instruction Set Architecture	PTX ISA	Assembly-like language used to write high-performance code for Nvidia GPUs.
Phi-Instruction	-	Instruction which chooses the right value in an SSA form depending on the CFG.
Single Instruction Multiple Data	SIMD	Parallel computing technique which allows to execute the same instruction sets on multiple data simultaneously.
Standard Portable IR	SPIR-V	Binary IR used as input language for APIs such as the OpenCL, OpenGL or Vulkan API.
Static Single Assignment Form	SSA Form	Specification of an IR that each variable must be assigned exactly once. If the value of the variable changes, a new variable is created and the new value is assigned to it.

Table 1.: Glossary.

Contents

Part I.

Report

1. Introduction

Nowadays, a lot of non-graphical tasks such as crypto mining and machine learning are executed on graphics processors. One of the most popular standards for general purpose GPU programming is called CUDA [1]. It is a platform for parallel computing and an application programming interface to run code directly on Nvidia GPUs.

CUDA is designed to be used in combination with the programming languages C/C++. Often one would like to use the computing power of the graphics card directly in other programming languages without writing and integrating individual code parts in C/C++. Unfortunately, there is currently no free and convincing open source library that makes this possible for C#. Besides CUDA there are some other APIs to run highly-parallel code on GPUs, such as the Vulkan API, OpenCL etc.

The goal of this bachelor thesis is to implement a prototype of a cross-compiler to execute C# code on GPUs. The minimum viable product (MVP) should contain at least the technical breakthrough. Simple operations should be able to be executed on the graphics card. Nvidia GPUs must be supported. Optionally, graphics processing units from other manufacturers may also be supported.

The workload will be distributed into the following parts:

- Initial research and identifying existing technologies.
- Creating a concept of a .NET to (Nvidia) GPU cross-compiler.
- Creating a prototype based on the concept.

2. Background Information

There are a lot of different approaches to compile C# code for GPUs and execute it on them. In this chapter some of the most popular alternatives to do so are evaluated. The goal is to get a rough overview of the different tool-chains.

2.1. Overview

The basic structure of the various GPU development tool chains is often very similar with varying amounts of compilation-steps. Some APIs use assembly-like input while others use C-like languages. Figure 2.1 shows a possible basic structure of a .NET to GPU cross-compiler.

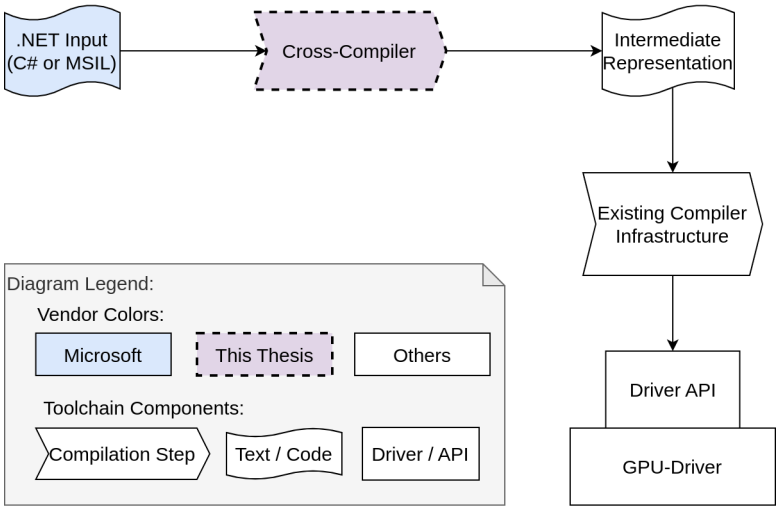


Figure 2.1.: Compilation to GPU Process Overview.

2.2. .NET Framework / C#

The .NET framework respectively C# code is the given input of the tool-chain to be developed. As a starting point, there are two possible main approaches:

- Compiling from C#, for instance by employing tools from the Roslyn compiler toolchain [2].
- Compiling the Microsoft Intermediate Language code, a byte code format stored in a DLL-file serving directly as input to the .NET virtual machine, generated by the Roslyn compiler for C#. After C# code is compiled to MSIL, the dynamic linked library (DLL) can be disassembled and the contained MSIL-code can be translated to a language accepted by the GPU-API.

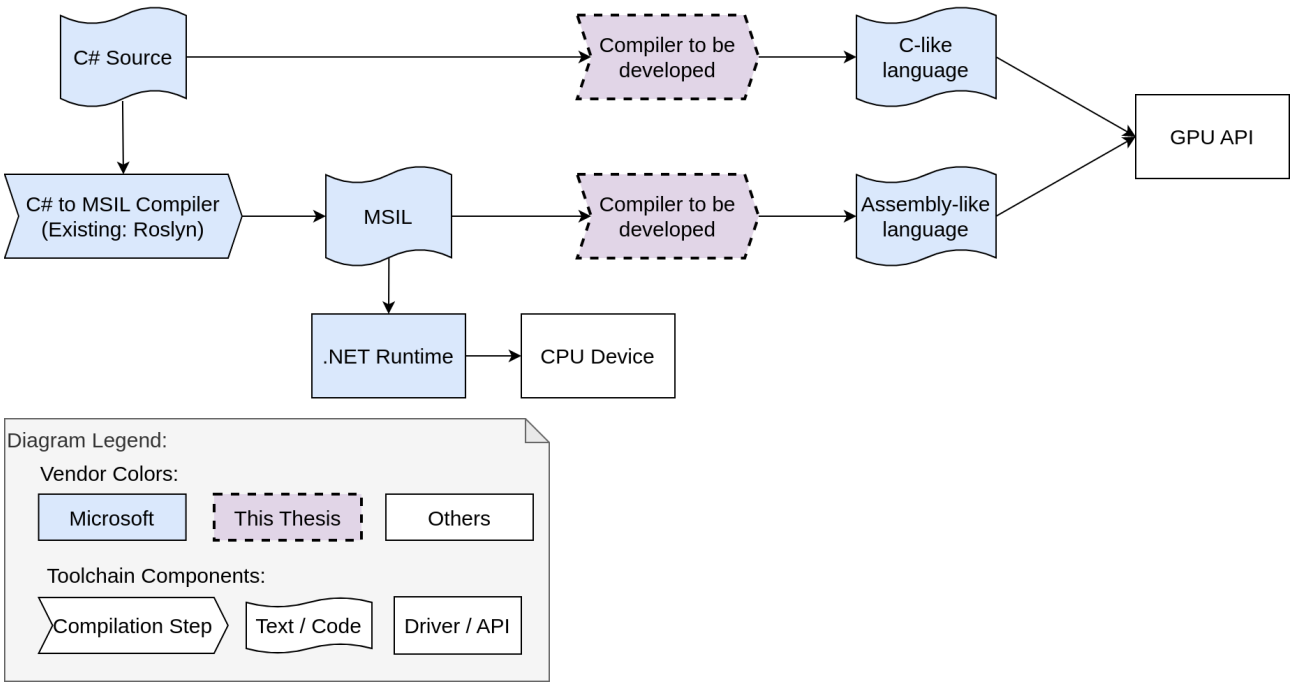


Figure 2.2.: .NET Compilation Process Overview Using the Roslyn C# Compiler [2].

2.3. Toolchains

There are various tool-chains for executing instructions on graphics cards. The best supported and most widely used input languages in this area are C, C++ and C-like languages. Often, compilers or wrappers already exist that connect the various tool-chains with each other. Figure 2.3 illustrates four of the most popular tool-chains and their interrelationships.

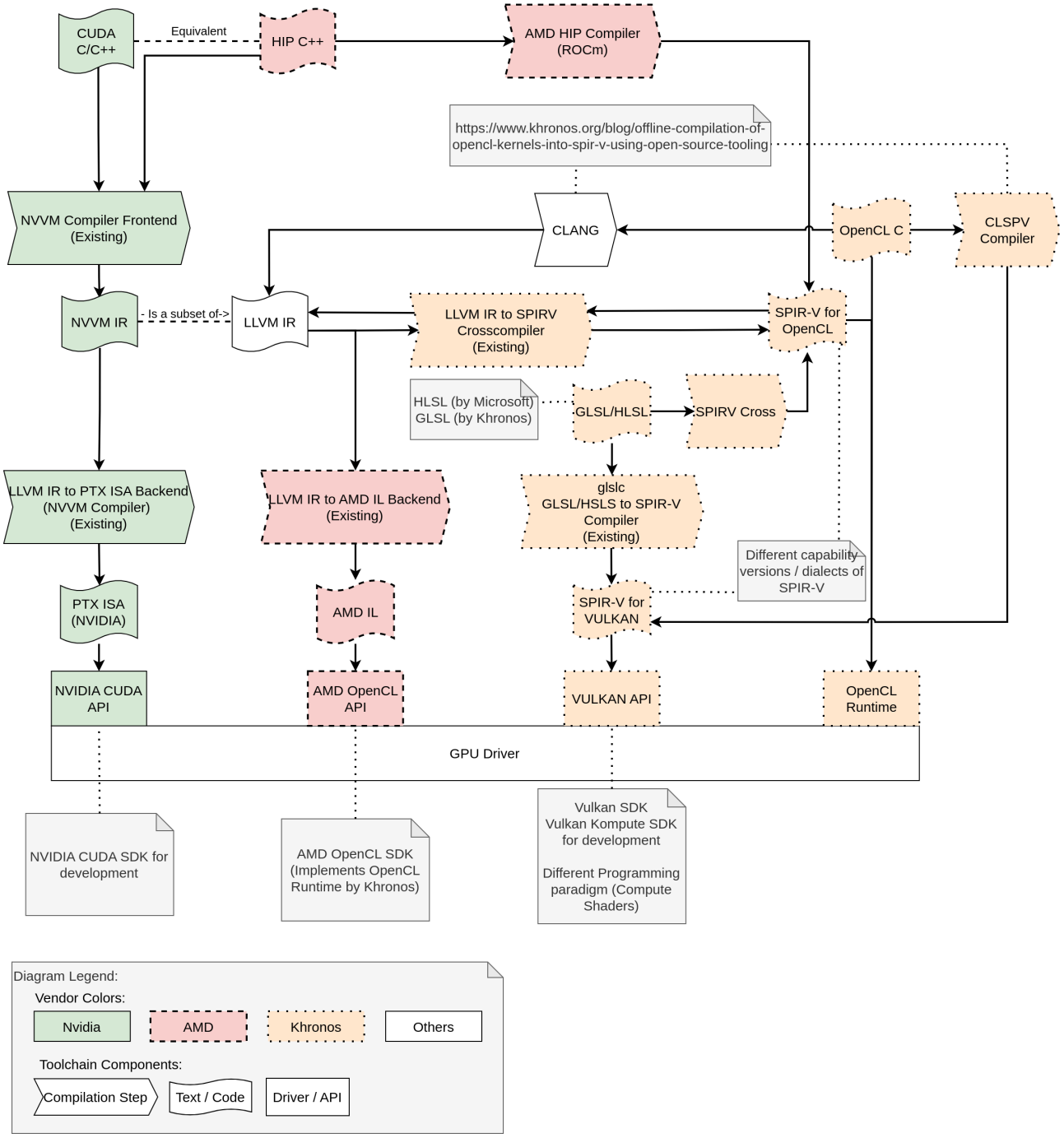


Figure 2.3.: GPU Toolchain Overview, Showing Toolchains for CUDA [1], Vulkan [3] and OpenCL [4].

2.3.1. Nvidia CUDA

Nvidia CUDA [1] consists of an API, different kernel programming languages and a toolchain. CUDA-API is a proprietary API for Nvidia GPUs. Nvidia's tool-chain supports CUDA-C, C++ and Fortran as kernel language. With the help of the NVVM compiler frontend the kernel language is translated to the intermediate language NVVM IR, a subset of LLVM IR. NVVM IR is then compiled to PTX ISA, which is afterwards executed on the GPU.

According to Nvidia's developer pages, the LLVM compiler for CUDA can also be used to compile other languages for Nvidia GPUs [5]. Implementations of this approach such as Alea GPU [6], Altimesh Hybridizer [7] and CUDAfy [8] do exist, but are either no longer maintained or closed-source.

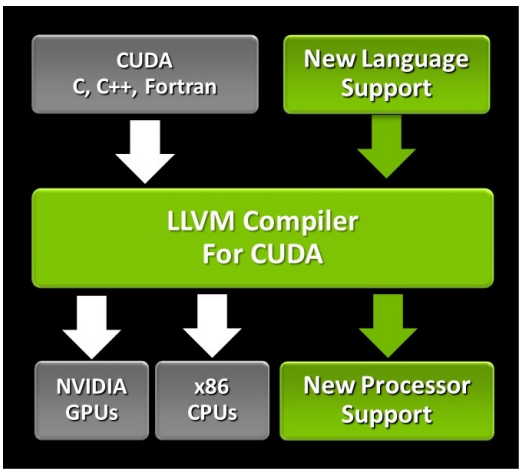


Figure 2.4.: Structure of LLVM Compiler for CUDA [5].

2.3.2. AMD HIP

AMD HIP [9] is a CUDA compatible language and compiler-toolchain by AMD. The goal of HIP is to make it possible to write the same C++ code for AMD and NVIDIA GPU-devices. The C++ code is compiled to PTX ISA for NVIDIA devices and AMD GCN ISA for AMD devices.

2.3.3. Vulkan

Vulkan [3] is an API for graphics and GPU-compute purposes. It is developed and maintained by the Khronos Group. The Vulkan API is implemented in GPUs of many vendors making it possible to write cross-platform compute-shaders.

2.3.4. OpenCL

OpenCL (Open Computing Language) [4] is an open standard that allows to write programs that can be executed on GPUs. GPUs from various manufacturers, such as Nvidia, AMD and Intel, are supported [10]. As an input language, OpenCL provides a C-like language called OpenCL C which can be executed directly on the OpenCL runtime.

The tool-chain is still maintained and also has the advantage of being platform-independent. C# wrappers for OpenCL C already exist, but they were often last modified a few years ago and do not seem to be maintained anymore. We also observed that the OpenCL C API is more cumbersome than CUDA API.

3. Requirements

3.1. Functional Requirements

No.	Level	Requirement
FR-1	Must	C# functions can be selected as a GPU-kernel.
FR-2	Must	C# kernels can be compiled to GPU-runnable kernels.
FR-3	Must	C# kernels can be launched on the GPU.
FR-4	Must	Data can be transferred to the GPU.
FR-5	Must	Data can be transferred from the GPU to main-memory.
FR-6	Must	A thread can access its thread-ID, block-ID and block-size.
FR-7	Must	Addition, multiplication and subtraction operations can be applied to data in kernels.
FR-8	Must	Any primitive numeric types of C# version 10.0 may be used as scalar or array-parameters of the kernel.
FR-9	Should	Division and modulo operations can be applied to data in kernels.
FR-10	Should	The same kernel can be launched multiple times.
FR-11	Should	Different kernels can be launched during a single program-execution.
FR-12	Should	While-loops and for-loops can be used in kernels.
FR-13	Should	If- and else-constructs can be used in kernels.
FR-14	Could	Generics can be used with kernel parameters.
FR-15	Could	Shared memory can be allocated and used in kernels.
FR-16	Could	Length-property of arrays can be used in kernels.
FR-17	Could	The framework supplies its user with tools to measure kernel-performance.
FR-18	Could	Kernels can call other methods.
FR-19	Could	Kernels can allocate new arrays.
FR-20	Could	Multidimensional arrays can be used in kernels.
FR-21	Could	Kernels can be compiled ahead of time.

Table 3.1.: Functional Requirements.

3.2. Non-Functional Requirements

No.	Level	Requirement
NFR-1	Must	Kernels can be compiled and run in less than 5 lines of code (excluding kernel-definition).
NFR-2	Must	Compilation of a simple vector-addition-kernel takes less than 3 seconds on the evaluation machines.
NFR-3	Must	Evaluation kernels runtime-performance does not deviate more than 50% from the same kernels run from the CUDA C++ library.
NFR-4	Must	The compilation steps to achieve FR-6 are unit-tested for a simple case.
NFR-5	Should	The compilation steps to achieve FR-8, FR-9, FR-12 & FR-13 are unit-tested for a simple case.
NFR-6	Could	The compilation steps to achieve FR-14 to FR-16 are unit-tested for a simple case.

Table 3.2.: Non-Functional Requirements.

3.3. Constraints

3.3.1. Source Language

The framework will be implemented in .NET/C# since the goal of this thesis is to run C# code on GPU-devices.

3.3.2. Target Devices

The code translated and deployed by the framework has to be compatible with NVIDIA devices. It may or may not be compatible with devices from vendors other than NVIDIA. In this project, only compatibility with NVIDIA devices is explicitly implemented and tested.

3.4. Scope

3.4.1. Minimum Viable Product (MVP)

The minimum viable product of this thesis is a framework that is able to deploy the C#-kernel of listing 3.1. Type T may be any numeric type. The generic type T of C# is not supported in the MVP. Operator OP represents either addition, subtraction or multiplication.

```
1 void Kernel(T[] a, T[] b, T[] c)
2 {
3     int idx = blockDim.x * blockIdx.x + threadIdx.x;
4     c[idx] = a[idx] OP b[idx];
5 }
```

Listing 3.1: MVP Kernel.

3.4.2. Limitations

The cross-compiler developed during this thesis supports a limited subset of the C# language as listed in the functional-requirements. Limitations include but are not limited to:

- Accessing fields not specified as a parameter of the kernel and not declared and defined inside the kernel except for kernel-tools like accessing block-dimensions.
- Using or declaring structured data (classes, structs, etc.), except for arrays, inside a kernel.
- Using structured data types, except for arrays, as parameter-types of a kernel.
- Calling methods except for kernel-tools like *SyncThreads*.

Declaring a kernel non-void and returning data using the *return*-keyword inside the kernel will be translated to a void-kernel returning no data.

4. Solution Strategy

This chapter documents choices made before starting with software-development. These decisions are required due to the great architectural impact imposed by the given alternatives.

To be able to run a piece of software on a GPU-device, it needs to be handed to a driver-API. Different APIs offer different capabilities and accept a kernel to be deployed in different formats. Choosing an API therefore also means choosing a target format that our C#-kernel needs to be compiled into.

4.1. Alternatives

The following alternatives were chosen for evaluation because they are established in academics and industry, well documented and still receive support and development efforts by their respective vendors.

4.1.1. Vulkan Compute-API

To run a kernel written in C# via the Vulkan compute-API it would need to be translated to a shading language such as GLSL or HLSL. Using the Vulkan libraries [3] provided by Khronos or a wrapper-framework such as Kompute [11], the shader would then be translated to SPIRV and deployed to the GPU-device. Since GLSL and HLSL are C-like languages, it would make sense to directly translate C#, for example using the Roslyn compiler-project [2], to the respective shading language.

4.1.2. NVIDIA CUDA

To run a kernel via the CUDA API one would first need to translate the kernel to either NVVM IR or PTX ISA. Since both NVVM IR and PTX ISA are assembly-like languages it is easiest to translate the kernel from its compiled MSIL form.

While both NVVM IR and PTX ISA are viable compilation-targets, NVVM IR seems like the better choice. As mentioned, NVVM IR is a subset of LLVM IR and there exist many compilers from LLVM IR to different platforms. Therefore, choosing LLVM as a compile-target will make it easier to target additional GPU-Compute-APIs in future projects without completely reimplementing everything done during this thesis.

4.2. Performance Evaluation

4.2.1. Test Setup

To test CUDA, the CUDA driver-API [12] was used, getting a pre-compiled PTX-kernel as input. The Kompute-library [11] was used to perform tests via the Vulkan API in a simplified manner. The programming models of CUDA and Vulkan differ in certain areas. The exact source-files of the tests can be found in Appendix E.

The following is a pseudocode representation of the test-kernel or test-shader respectively.

```
1 For i = 0 To 999
2     outArray[kernelIndex] = arrayA[kernelIndex]*arrayB[kernelIndex] +i
3 EndFor
```

Listing 4.1: Evaluation Kernel Pseudo-Code.

The for-loop is there to increase computational overhead and therefore make the kernel more compute-bound.

4.2.2. Results

These results show an average of 10 measurements per input-size, device-model and API. The raw data can be seen in Appendix F.

Notably, Vulkan compute-shaders not only perform worse than CUDA-kernels in general, but also start

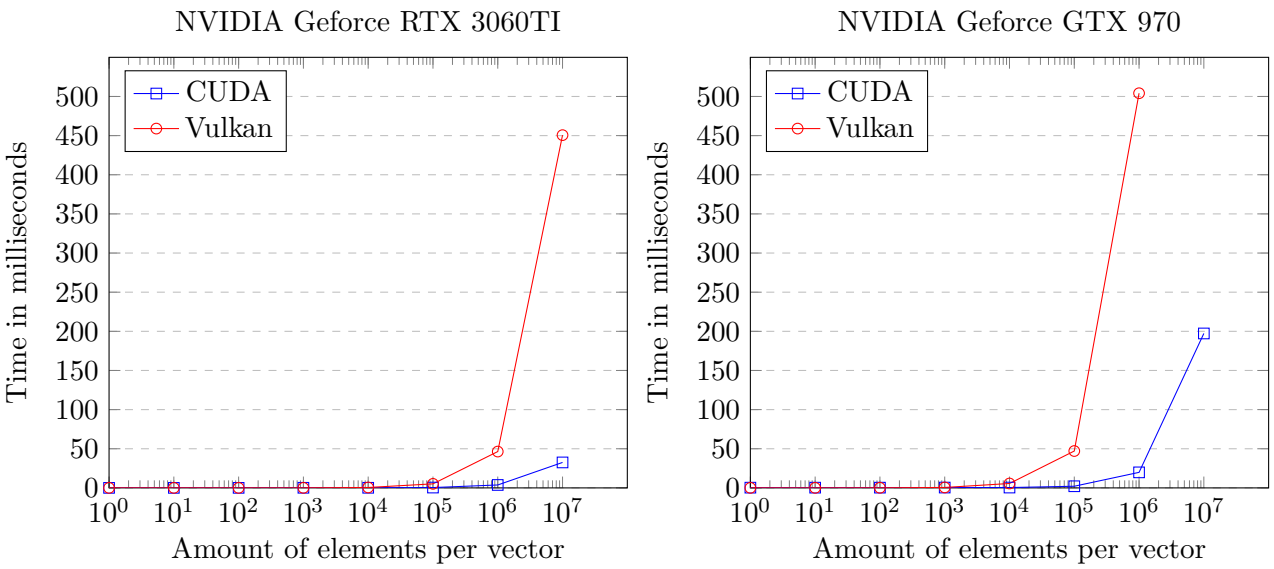


Figure 4.1.: CUDA vs Vulkan Testresults.

to scale linearly with the vector-size earlier. This is most likely due to limitations to the amount of threads started in parallel in Vulkan. Additionally, CUDA scales stronger with the efficiency of newer GPUs. At 10 million elements per vector CUDAs performance increased 506.5% from the GTX970 to the RTX3060TI, while Vulkans performance increase is only at 11.2%.

4.3. Additional Evaluation

In addition to performance indicators, the type of IR we want to target and the way in which kernels or shaders are executed are also important to the success of this thesis.

Since assembly-like languages are more simple in their grammar than C-like languages it will be less labor to translate MSIL to NVVM IR than to translate C# to GLSL.

The CUDA driver-API has similar semantics as the product of this thesis should have. The Vulkan-API itself is more cumbersome to use and differs in semantics, especially when comparing grid size and block size definition in CUDA to workgroup-size definition in Vulkan/Kompute.

Even though a Vulkan-based product could natively be used on most modern GPU-devices, similar platform independence can also be achieved by creating or using additional compilers from NVVM IR to platform-specific instruction sets in future projects and theses.

4.4. Decision

Considering the results of the evaluation, this thesis will implement a product based on CUDAs driver-API and the Nvidia NVVM IR Library (LibNVVM).

5. Architecture

This chapter describes the architecture of the software-library called *CuSharp* that enables its user to run kernels written in C# on NVIDIA GPU-devices. The architecture is documented using the C4 template [13]. The context-diagram shows the library itself as a black-box and emphasizes external dependencies. The container-diagram shows separately usable parts of the library. The component-diagram further refines the architecture by presenting software-components and dependencies between them in an abstract manner. Components are not necessarily single C# classes and not every C# class is necessarily represented by a component.

5.1. Context

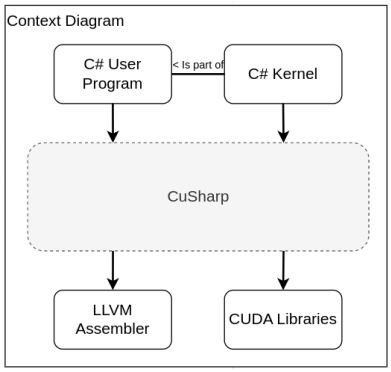


Figure 5.1.: C4 Context Diagram.

The *program written by the user* in C# uses the framework in development to configure how *kernels* are run. It also specifies which C# functions are meant to be discovered as kernels by the library.

The kernel-itself also uses the library, to receive information about its thread-id, block-id and block-size.

The *LLVM Assembler* allows the framework to emit LLVM IR.

CUDA Libraries are used to translate our kernel to CUDA specific representations and to launch kernels on a GPU-device.

5.2. Containers

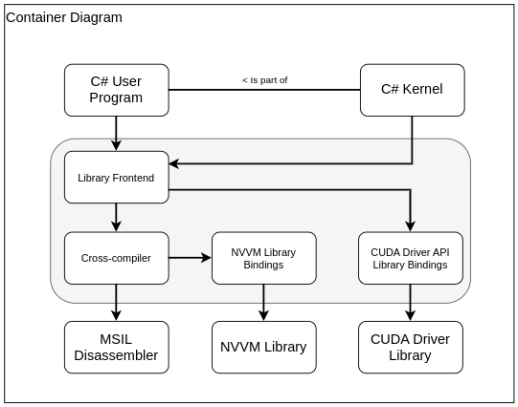


Figure 5.2.: C4 Container Diagram.

The *Framework Frontend* that gives the user the ability to configure and launch kernels allows various compilation backends to be used.

The *Cross-compiler* compiles an MSIL function to NVVM IR and finally translates it to PTX ISA.

The *NVVM Library Bindings* use the *NVVM Compiler Library* to compile NVVM IR to PTX ISA

The *CUDA Driver API Bindings* make the *CUDA-Driver API* available to the *Framework Frontend*

5.3. Components

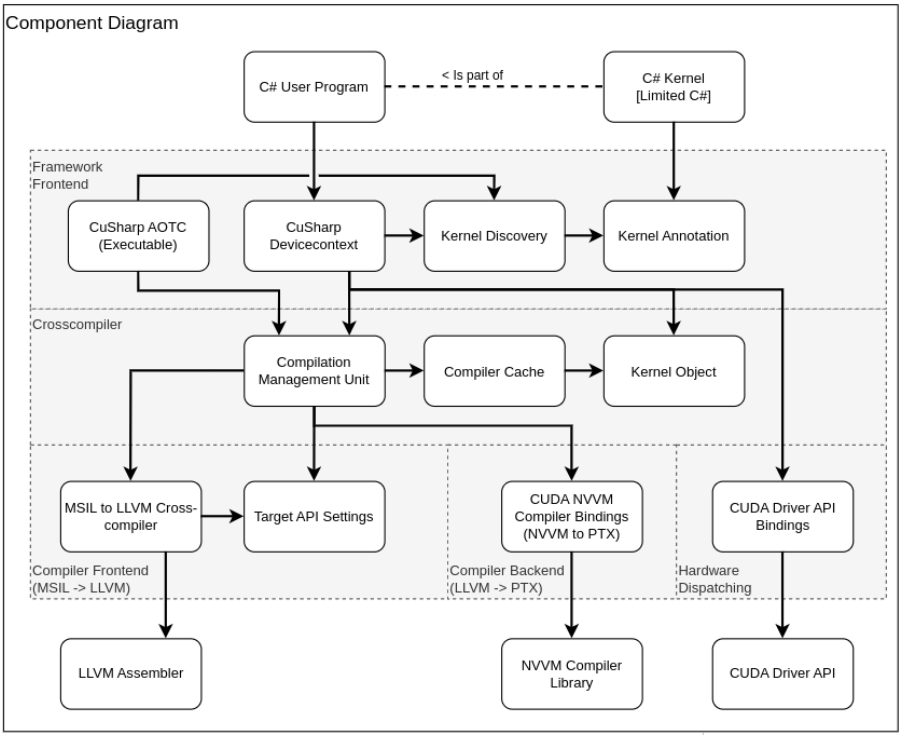


Figure 5.3.: C4 Component Diagram.

5.3.1. Framework Frontend

The framework-frontend consists of three components:

CuSharp AOTC is an executable that acts as an ahead-of-time compiler. It allows its users to compile all kernels marked with the *Kernel Annotation* before the *C# User Program* is being executed.

A *CuSharp Devicecontext* allows the user to configure and launch kernels on the GPU-device. When launching a kernel, the device context will either JIT-compile the method selected as kernel, or use an AOT-compiled kernel, if it exists.

The *Kernel Discovery* component is encapsulated in its own component that is responsible for identifying kernels marked with a *Kernel Annotation* via reflection.

5.3.2. Cross-Compiler

The Cross-compilation layer contains the business logic of the framework in development.

The *Compilation Management Unit* acts as an abstraction layer for the compilation-process and manages all stages of the tool chain. It makes compilation-results available to later stages in a *Compiler Cache* as a *Kernel Object* that is no longer meant to be analyzed in detail by the framework in development. It also tries to find any ahead-of-time compiled kernels to use instead of just-in-time compiled ones.

The *Kernel Object* contains the final PTX-code.

The *MSIL to LLVM Cross-Compiler* generates LLVM code with constraints such as NVVM as specified in the *Target API Settings* component.

The *CUDA NVVM Compiler Bindings* translate NVVM compliant IR into PTX ISA, by using the *NVVM Compiler Library*.

5.3.3. Hardware Dispatching

The *CUDA Driver API Bindings* are called upon by the *Cuda Context* and runs an *Kernel Object* by passing it to the *Cuda Driver Library*.

5.4. Architectural Decisions

5.4.1. Framework Frontend

External dependencies In the context of maintainability we decided to hide all external dependencies of our software from the user and to not allow any access to third-party libraries. This ensures that external dependencies are easily replaceable. It also allows us to guarantee full control over the frameworks interface, accepting more labor in implementation and architecture.

Kernel Launch API In the context of the API allowing the user of the framework in development to launch a kernel on a GPU device, we decided to use multiple overloads of the launch-method and against creating a single-method with generally-typed parameters or manually overloading the launch-method, to give programmers more type-safety and allowing for faster changes of the framework-API, accepting additional development time of the code-generator.

5.4.2. Compiler Frontend

Disassembling MSIL: In the context of disassembling MSIL code to cross-compile it to LLVM, we decided not to use a dedicated disassembler-library or component. Instead, we chose to manually read the binary MSIL-opcodes and operands as needed and cast them to a readable format to lessen the performance overhead. While this approach may result in a loss of type-safety and convenience, we deemed it to be an acceptable trade-off.

LLVM IR Assembling: In the context of assembling LLVM IR during cross-compilation, we decided to use *LLVMSharp* [14] instead of developing a dedicated LLVM or NVVM assembler. This decision was made to save time during development and to gain confidence in the generated code, as well as to assemble more optimized code. However, we accepted less control over the generated code and the need to learn the *LLVMSharp* API.

5.4.3. Compiler Backend

NVVM to PTX Compilation: In the context of NVVM IR to PTX ISA compilation, we faced the need to programmatically and easily compile and access the results. We decided to create our own libNVVM bindings instead of using the open-source executable compiler "llc". This choice allowed us to achieve more control over compilation, although it required more overhead for creating and maintaining the libNVVM bindings.

Cross-Compiler Device Specifics: In the context of device specifics, we decided to create a *Compiler Configuration* component that encapsulates device specific datalayouts and kernel-annotations. By doing so instead of integrating this information into the compiler, we enable future development of the framework to expand the compiler to support more devices from different device-vendors. To support this we accept a more complex interface of the cross-compiler.

5.4.4. Hardware Dispatching

GPU Kernel Execution Library: In the context of executing GPU kernels via the CUDA API, we faced the need to transfer data to and from the GPU device, as well as launching kernels. In order to achieve more control over the API, we decided to use the CUDA Driver Library instead of the CUDA Runtime Library. This better fits our requirements, despite the Driver Library having a more complex API compared to the Runtime Library.

Kernel Execution Bindings: In the context of executing GPU kernels via the CUDA Driver Library in C#, we faced the need to access the CUDA Driver Library in an object-oriented manner. After considering our options, we decided to use *managedCUDA* [15] instead of creating our own bindings. This decision was made to save time in development and licensing research, even though we accepted having to use the API of *managedCUDA* instead of having an API perfectly fitting our use-case.

6. Quality Assessment

The following chapter is a quality assessment of the product built during this thesis. The sections below list the fulfillment status of requirements as well as a performance analysis.

6.1. Testing

Three classes of tests were performed to assess the quality of CuSharp:

Automated Unit-Tests:	Contained in the *CuSharp.Tests* project. Test the behavior of single components in the architecture.
Automated Integration-Tests:	Contained in the *CuSharp.Tests* project. Test the behavior of the whole system including compilation and runtime.
Performance-Tests:	Contained in the *CuSharp.PerformanceEvaluation* project. Measure the performance of different types of kernels.

Table 6.1.: Overview of Test Types.

In total there are almost 300 automated tests. About two thirds are unit tests and one third are integration tests.

6.1.1. Test Coverage

An evaluation of the test coverage has been done, but an exact coverage number cannot be given.

The problem is that certain tests need to run in debug mode and certain tests need to run in release mode. The tests depend on the MSIL output from Roslyn. The output varies depending on the build configuration. When evaluating the test coverage this leads to different reports being generated and these would have to be manually merged to get the numbers (lines of code covered and branches covered) right.

What can be said is that the coverage is at least 77%.

6.2. Requirements Analysis

6.2.1. Functional Requirements

No.	Fulfilled?	Level of Fulfillment
FR-1	Yes	All static methods may be selected as kernels.
FR-2	Yes	Fulfilled. All methods complying with the CuSharp language subset may be compiled to GPU-runnable kernels.
FR-3	Yes	All kernels complying with the CuSharp language subset may be launched on Nvidia-GPUs.
FR-4	Yes	All types as defined in Chapter D are transferrable to the GPU.
FR-5	Yes	All types as defined in Chapter D are transferrable to main-memory.
FR-6	Yes	NVVM intrinsic fields and functions are accessible as defined in Chapter D.
FR-7	Yes	Fulfilled for all supported data types.
FR-8	Yes	Fulfilled as specified in Chapter D.
FR-9	Yes	Fulfilled for all supported data types.
FR-10	Yes	Compiled kernels are cached and can be redeployed to the GPU.
FR-11	Yes	Any number of kernels may be executed during one single program-execution.
FR-12	Yes	Fulfilled as specified in Chapter D.
FR-13	Yes	Fulfilled as specified in Chapter D.
FR-14	No	Unfulfilled because of time-constraint.
FR-15	Yes	Newly allocated arrays are put into shared memory.
FR-16	No	Unsupported because it would generate major runtime-overhead.
FR-17	Yes	*CUDA-Events* can be used to measure execution time.
FR-18	Yes	Supported as specified in Chapter D.
FR-19	Yes	Arrays allocated during kernel execution, are allocated in shared memory.
FR-20	Yes	Multidimensional arrays can be used, both as parameter and local-variable.
FR-21	Yes	Implemented by the CuSharp.AOTC executable.

Table 6.2.: Analysis of Functional Requirements.

6.2.2. Non-Functional Requirements

No.	Fulfilled?	Level of Fulfillment
NFR-1	Yes	One line for specifying the device and one line to launch a kernel
NFR-2	Yes	Fulfilled as specified in Section 6.4.
NFR-3	Yes	Fulfilled as described in Section 6.4.
NFR-4	Yes	FR-6 is extensively tested by many unit-tests and integration-tests.
NFR-5	Yes	FR-8, FR-9, FR-12 & FR-13 are unit-tested and integration-tested.
NFR-6	Partially	FR-16 is not implemented. FR-14 and FR-15 are implemented and tested.

Table 6.3.: Analysis of Non-Functional Requirements.

6.3. Technical Debt

6.3.1. Platform Independence

Due to the NVVM native bindings, CuSharp can only be used on Windows machines.

Windows library: 'nvvm\bin\nvvm64_40_0.dll'

Linux library: 'nvvm/lib64/libnvvm.so.4.0.0

6.4. Performance Analysis

The following analysis show the results of experimental performance measurements of the compiled kernels to be deployed on a CUDA enabled GPU-device. Measurements only include the time to run on the devices and do not incorporate time to compile or data-transfer times. The results show the performance of the MSIL to PTX compiler developed by this thesis in comparison to the proprietary compilers developed by Nvidia.

All measurements are averages of 10 measurements.

6.4.1. Test Setups

Test device A

Device ID	Nvidia GeForce GTX 970
No. of Cores	1664
Memory size	4 GB
Memory type	GDDR 5
CUDA API Version	5.2

Test device B

Device ID	Nvidia GeForce RTX 3060TI
No. of Cores	4864
Memory size	8 GB
Memory type	GDDR 6
CUDA API Version	8.6

6.4.2. Matrix Multiplication Performance

The following charts show the execution time of matrix multiplication operations on *double* typed data, with matrices of sizes between one and four million elements. Note that the x-Axis is not scaled linearly but polynomially since we scaled the matrix width linearly. The kernel used in this evaluation can be seen in appendix, Section E.1

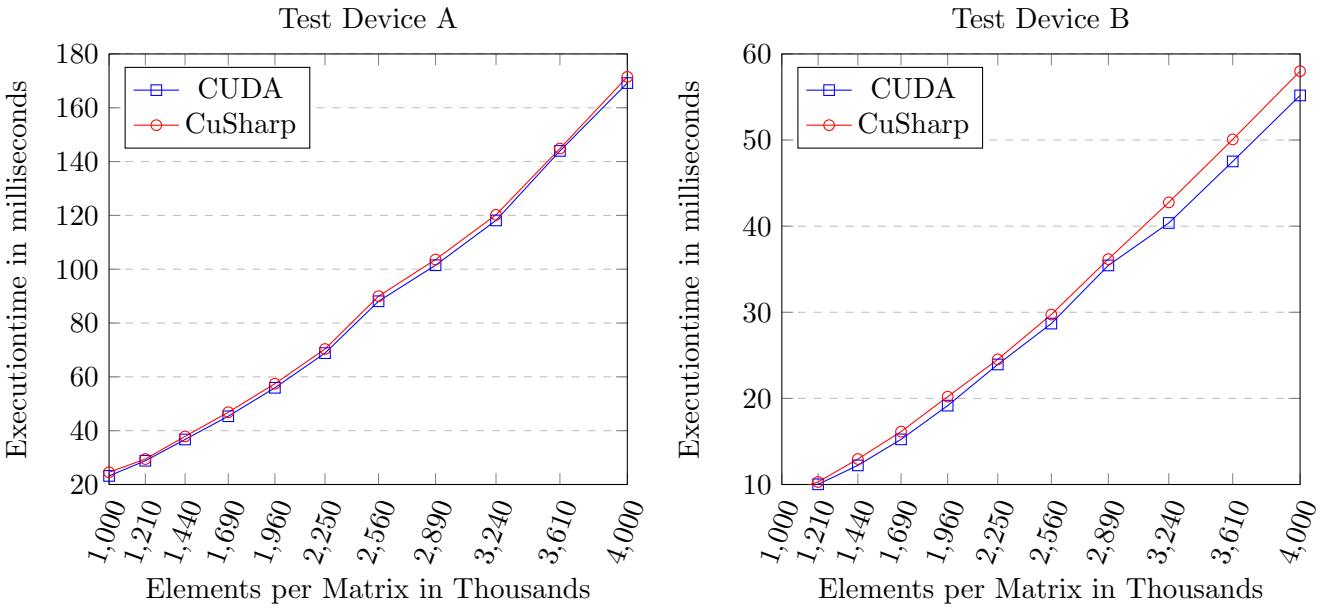


Figure 6.1.: CUDA vs CuSharp Matrix Multiplication Test Results.

The kernel-code generated by the original cuda compiler generates more efficient code in all cases.

This is likely due to more optimizations being made. The deviation between CuSharp and CUDA performance is almost constant across all tested matrix sizes.

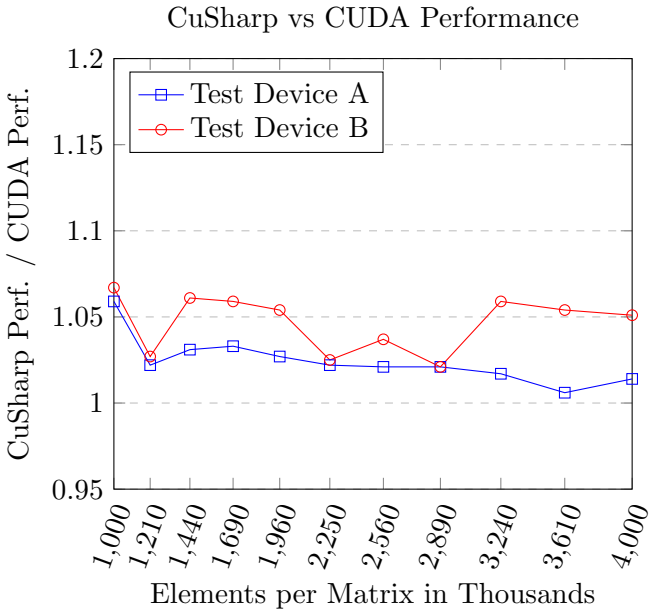


Figure 6.2.: CUDA vs CuSharp Matrix Multiplication Deviation Factor.

On average, CuSharp-compiled kernels executed 2.5% slower than CUDA-compiled kernels on test device A and 4.7% slower on test device B.

6.4.3. Tiled Matrix Multiplication Performance

The following charts show matrix multiplication operation similar to the one performed in Subsection 6.4.2, but accelerated by first copying tiles of each input matrix into the shared memory of the devices. The kernel used in this evaluation can be seen in appendix, Section E.2.

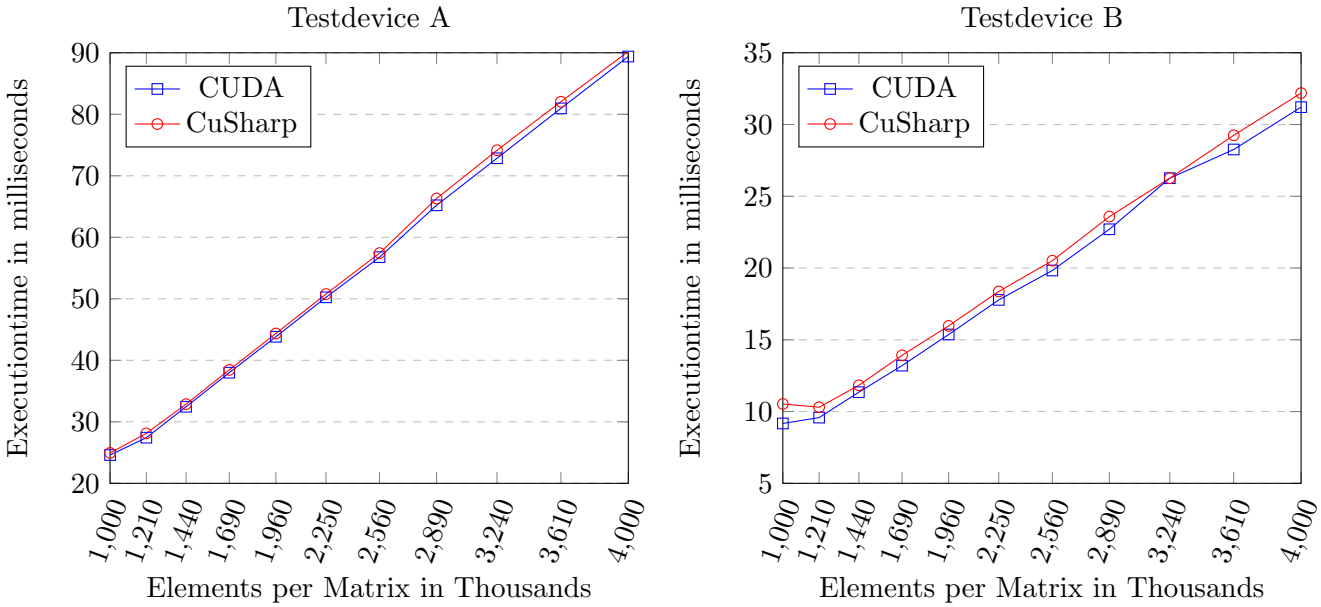


Figure 6.3.: CUDA vs CuSharp Tiled Matrix Multiplication Test Results.

The deviation between CuSharp and CUDA performance, again, has an almost constant relationship.

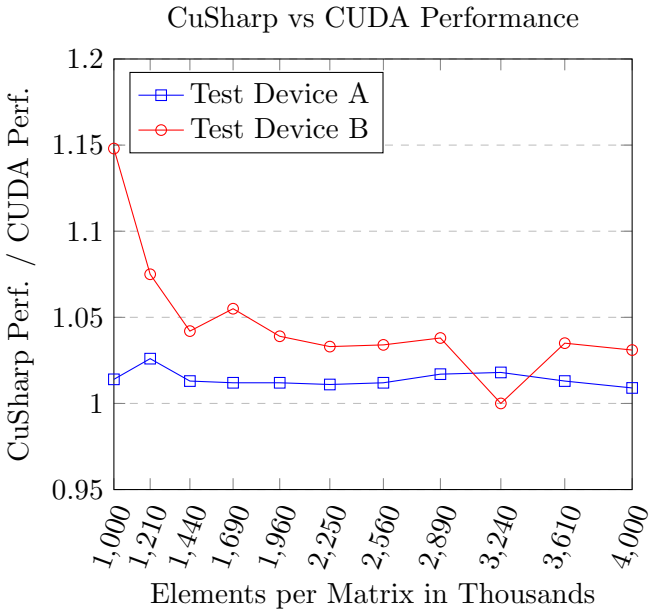


Figure 6.4.: CUDA vs CuSharp Tiled Matrix Multiplication Deviation Factor.

On average, CuSharp-compiled kernels executed 1.4% slower than CUDA-compiled kernels on test device A and 4.8% slower on test device B.

6.4.4. 2D-Array Matrix Multiplication Performance

As described in Chapter 7, two-dimensional arrays have a performance overhead over single-dimensional arrays. The following chart illustrates this overhead. Measurements were performed on double-typed matrix multiplication operations, on test device A.

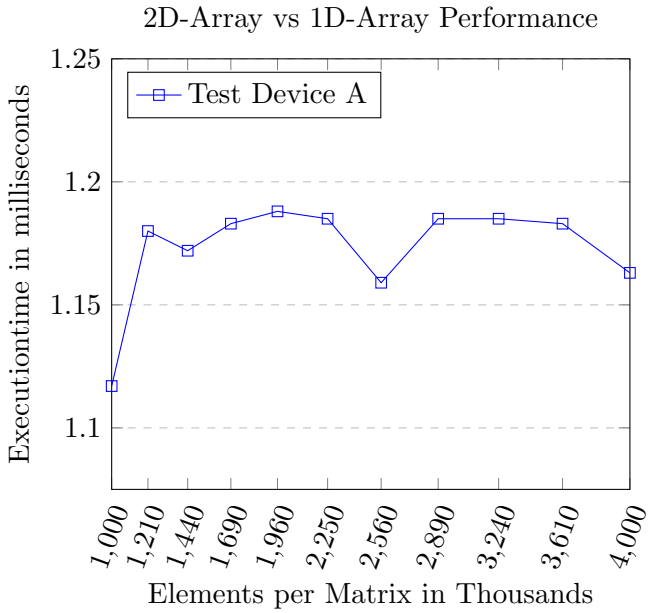


Figure 6.5.: CUDA vs CuSharp Matrix Multiplication Test Results.

The overhead scales linearly, since it stems from the fact that each array-access needs to dereference two array-references in two-dimensional arrays, instead of the single dereferencing of single-dimensional arrays.

7. Implementation Challenges

This chapter gives an overview of the hardest problems we encountered while implementing this thesis.

7.1. Language Execution Model Deviation

Our source and target-language deviate in how they handle kernel-local memory. While MSIL, the source language, targets a virtual stack, NVVM IR, our target language, uses virtual registers. To handle this deviation, we introduced a stack of registers in our cross-compiler. Any time MSIL pushes a value onto the stack, we store it in a virtual register and push a reference to said register onto our compiler stack. Any time MSIL instructs us to pop a value from the stack, we pop a register from the compiler stack. This process is well known and is thoroughly described in literature [16].

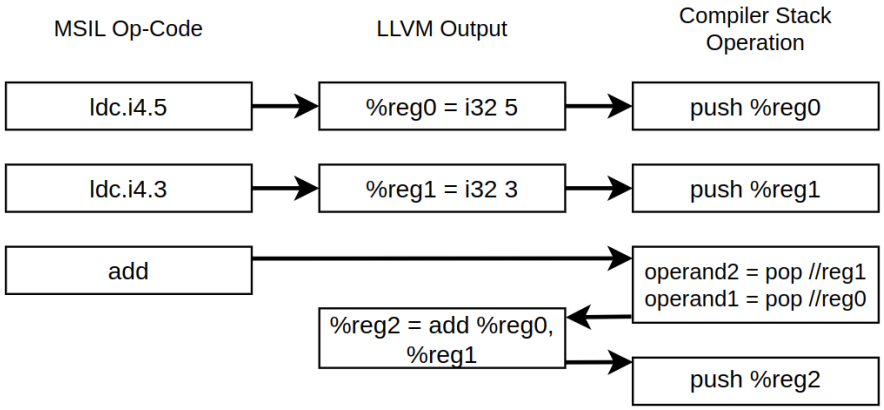


Figure 7.1.: Compiling from Stackmachine to Registermachine via Compiler Stack.

7.2. Static Single Assignment (SSA)

LLVM IR and therefore NVVM IR constrains all code to static single assign (SSA) form [17]. Compiling any non-SSA language into LLVM IR needs special care, as described in sections 7.3 and 7.4. To solve this issue, the cross-compiler built during this thesis creates a control-flow-graph (CFG) [18] and populates its nodes with the necessary information.

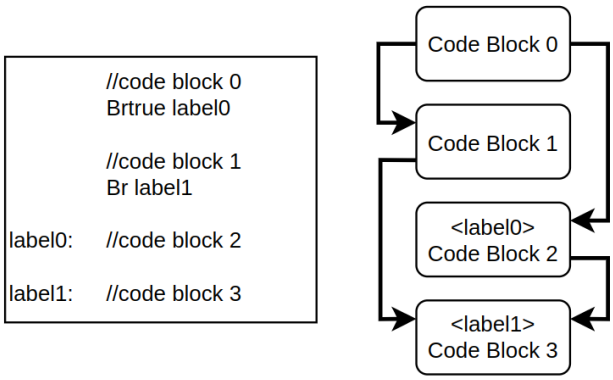


Figure 7.2.: Control-Flow-Graph Definition for an MSIL Example.

7.3. Variable Deviation after Branching

Local variables are represented by virtual registers in LLVM. The SSA-constraint may lead to conflicts between registers to use as a local-variable or parameter after multiple branches merge, by jumping into the same target-branch.

To resolve the confusion, the *phi*-function is used. The phi-function accepts parameters where the i-th parameter represents the value of the version of the variable that is valid if the i-th predecessor of the current block led to the current block being executed.

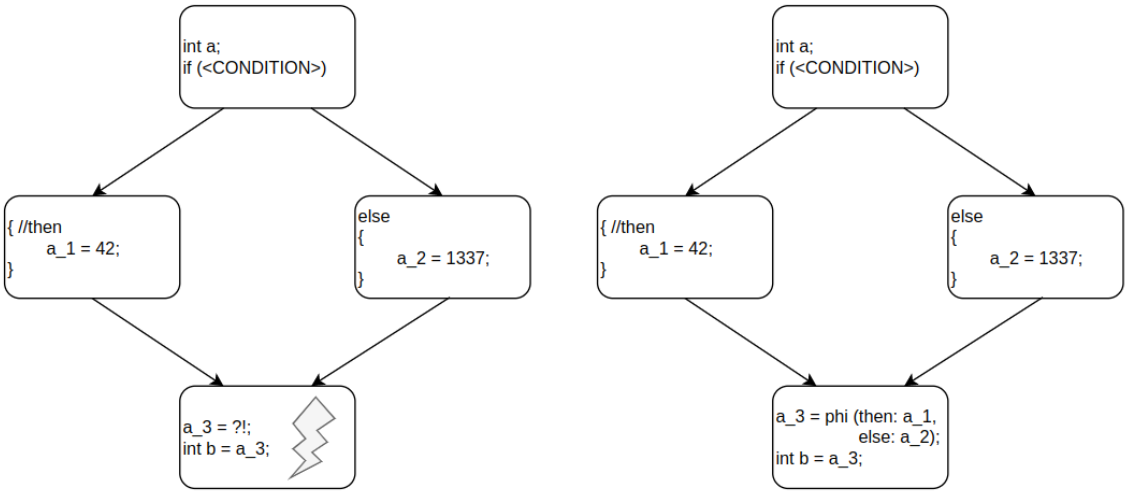


Figure 7.3.: Phi-Problem (Left) and Solution (Right) after Branching (in Pseudocode).

An additional problem occurs when the block that is the target of two preceding blocks does not access local variables of its predecessors. In this case, later blocks will not be able to refer to these variables. Therefore, the phi-function has to be introduced regardless of the current blocks usage of the respective variables.

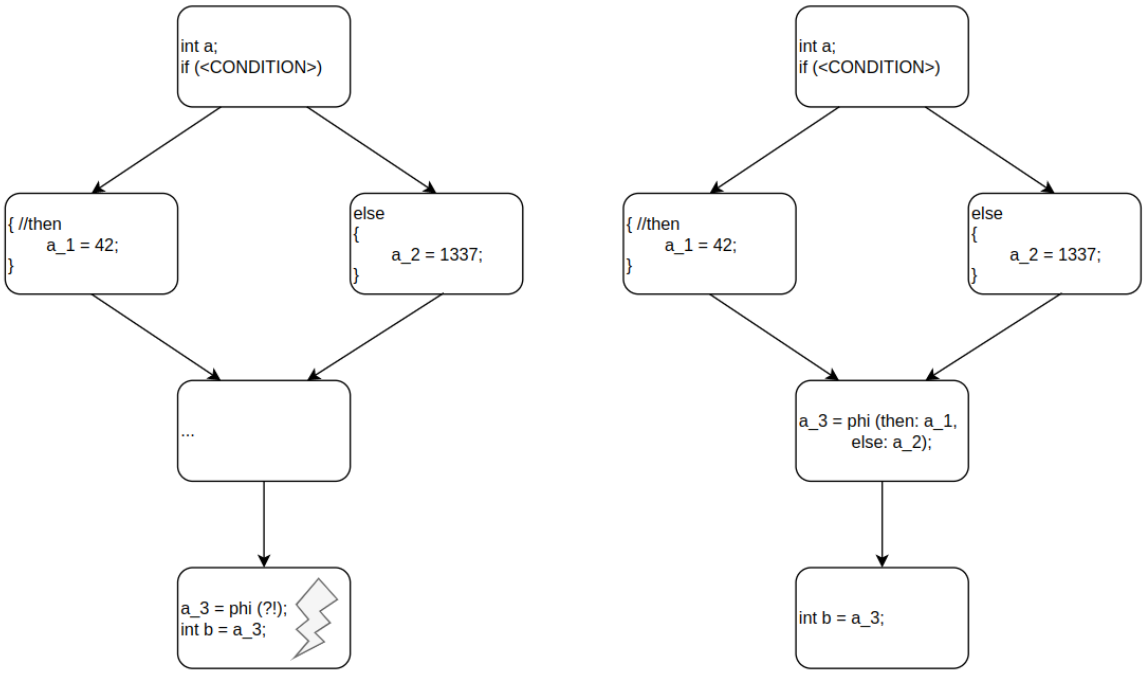


Figure 7.4.: Phi-Problem (Left) and Solution (Right), after Branching to Predecessor (in Pseudocode).

Not all predecessors of a block in the control-flow-graph may be known at the time the block is compiled. To solve this issue, compilation is separated into a main compilation phase and a post compilation phase. During the main compilation phase, all op codes are compiled to their LLVM counterparts and the control-flow-graph is being built. Phi-inststructions are built, but left empty without parameters. During the post compilation phase, the control-flow-graph is being walked again

and all phi-instructions get their incoming values.

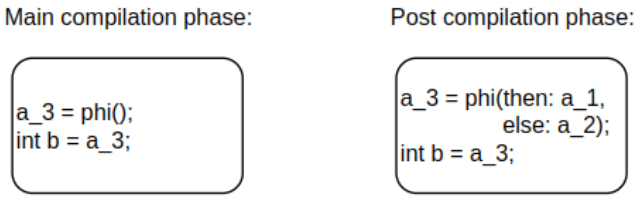


Figure 7.5.: Compilation Steps for Building Correct Phi-Functions.

7.4. Stack Deviation after Branching

Additional branching conflicts might occur with respect to the variable-stack. MSIL allows for different values to be pushed onto the stack right before branching occurs. Since we translate the stack based MSIL language to the register based LLVM language by creating a stack of registers during compilation, this means that, at compile time, it is unclear to us how the stack should look at the start of a block with multiple predecessors.

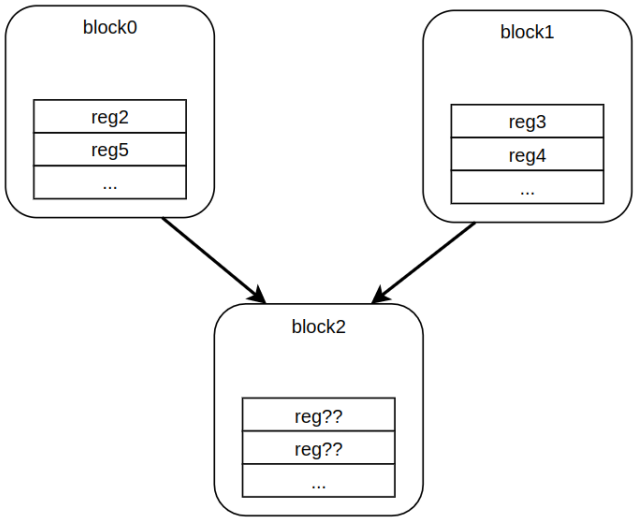


Figure 7.6.: Branch-Conflict with Compiler Stack.

To resolve this conflict, additional phi-instructions need to be built. The newly built instructions replace the conflicting ones on the stack. This ensures that the right register-values will be on the stack regardless of which predecessor jumped to the current block.

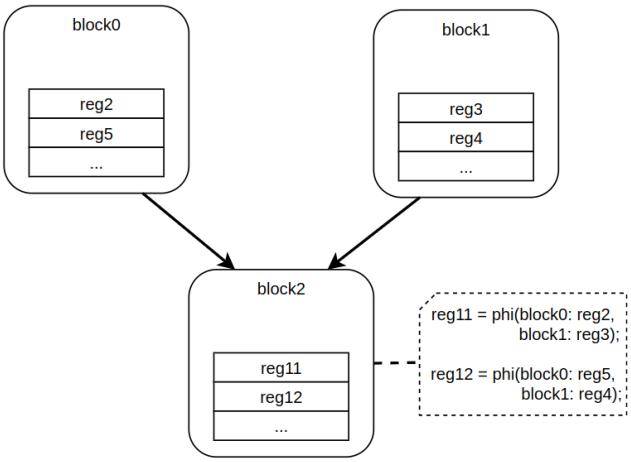


Figure 7.7.: Branch-Conflict with Compiler Stack Solution.

7.5. Array-Length Property

In C# the length of an array can be queried by using `array.Length`. This feature does not exist in LLVM. To still be able to use the *length*-property in CuSharp-kernels, array-lengths of parameters must be provided to the LLVM kernel at runtime as additional parameters.

This can either be achieved by creating a map from array-address to array-length, with an entry for each array-parameter, or alternatively, instead of allocating just the array, by allocating a struct containing a reference to the array and a field containing the length of the array.

Due to time-constraints and the performance overhead of managing the mechanisms to query the array-length, CuSharp currently does not support the array length property.

7.6. Two-Dimensional Arrays

C# supports two-dimensional Arrays that, for instance, can be used to represent matrices.

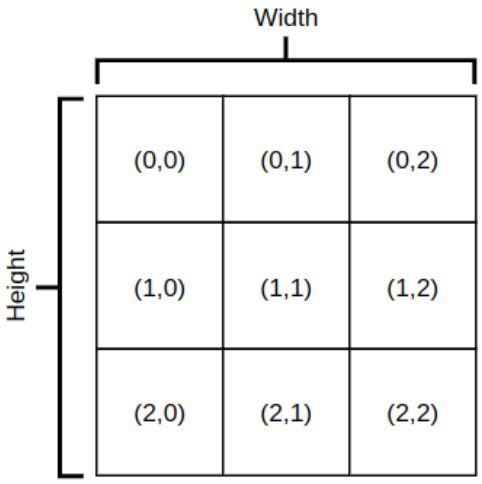


Figure 7.8.: Two-Dimensional C# Arrays.

There are two ways to support two-dimensional arrays in our compiler.

Two-dimensional arrays can be flattened-out, so that they appear as a single dimensional array inside NVVM IR.

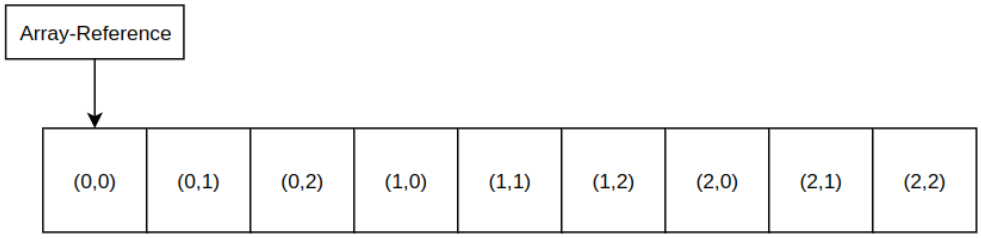


Figure 7.9.: Flattened-out 2D-Array.

Access to an element at position `(x,y)` in an array with a width of `rowLength` would look as described in Listing 7.1.

```
1 element = flattArray[x * rowLength + y]
```

Listing 7.1: Pseudo-Code of Two-Dimensional Array Access Using Array-Length.

Given that the array-length access is implemented efficiently, this way would be efficient itself, since we would only need a single dereferencing of an array-reference. However, since CuSharp currently does not support array-length access, we cannot implement the access this way.

Alternatively we can create an additional array, containing a reference to each row-start.

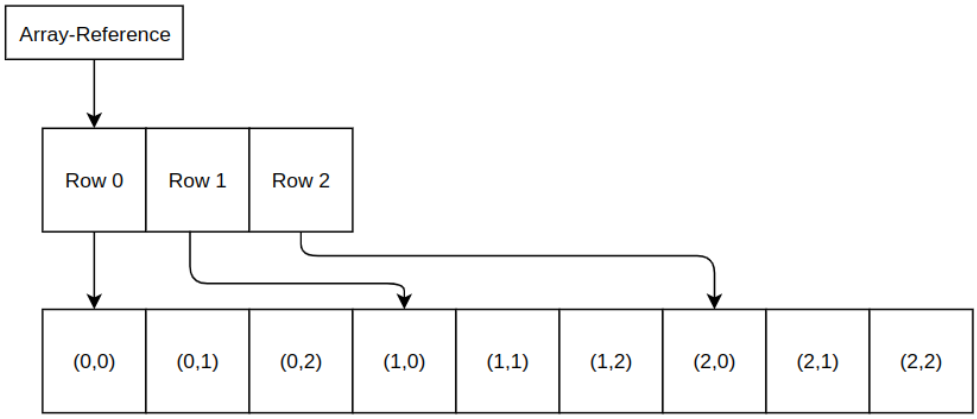


Figure 7.10.: Flatted out 2D-Array with Additional Row Array.

To access an element at position `(x,y)` we can proceed as described in Listing 7.2.

```
1 rowArray = arrayOfRows[x];
2 element = rowArray[y];
```

Listing 7.2: Pseudo-Code of Two-Dimensional Array Access Using Two Arrays.

This approach has the performance disadvantage of an additional indirection. However, it is more suitable if we do not have any information about the with of the 2D-array at runtime. As a result, CuSharp implements the later approach.

7.6.1. Initialization of Two-Dimensional Arrays

CuSharp kernel-parameters receive two-dimensional arrays as arguments of type `T**` where `T` is the type of the arrays elements. To do this, CuSharp copies the two-dimensional array of type `T[,]` to a one-dimensional array of type `T[]` and stores pointers to each row in an additional array, passing the latter typed `T**`.

A series of NVVM IR restrictions on LLVM IR lead to a more complicated process for initialization of multidimensional, locally-allocated arrays.

The first restriction is that the `alloca`-instruction is not supported except for thread-local memory allocation [19]. To allocate memory in address-spaces `global` and `shared`, the only possible way found during research is to declare an array-typed global variable.

LLVM-arrays are declared with the type specifier `[N x T]` where `N` is the statically defined number of elements in the array and `T` is the type of the elements of the array. The phi-function can only return values of a single type. In the case of the one-dimensional array, this poses no problem since a value of type `[N x T]` can simply be cast to `T*`.

In the case of two-dimensional arrays, however, allocation becomes more complicated. Type `[N x [M x T]]`, where `N` is the height of the array and `M` is the width of the array, cannot be cast to `T**`, which is the type of two-dimensional arrays used as parameters.

Because a single phi-function returns one type, using both `[N x [M x T]]` and `T**` typed registers in a single phi-call is not possible.

Therefore, using `[N x [M x T]]` would mean that parameters could not be assigned to local variables, and local variables could not be assigned to parameter variables, since these assignments require both types to be used in single phi-calls.

The solution to this is to simply declare one global variable of type `[NM x T]`, containing the elements of all rows of the array (`N` times `M` elements) and one global variable of type `[N x T*]` containing pointers to the rows of the first array. The later variable will then be used to refer to the two-dimensional array.

The array of row-pointers now needs to be initialized by a series of `N` store-operations. For arrays passed as arguments, this process is performed outside the kernel. However, for locally allocated arrays this needs to be performed inside the kernel.

8. Final Product

8.1. API

8.1.1. Static Class: Cu

Methods:

- `static IEnumerator<(int ID, string Name)> GetDeviceList()`
  Returns an enumerator to a tuple of CUDA-enabled device-IDs and device-names.
- `static CuDevice GetDeviceById(int deviceId)`
  Accepts a device-ID as argument.
  Returns a CuDevice instance.
- `static CuDevice GetDefaultDevice()`
  Returns a CuDevice instance of the system-default CUDA-device.
- `static CuEvent CreateEvent()`
  Returns a CuEvent instance.

Properties:

- `static bool EnableOptimizer`
  Is used to enable or disable NVVM IR optimization.
  Defaults to *true* in release-mode, *false* in debug-mode.
- `static string AotKernelFolder`
  Is used to set a folder of precompiled kernels to be used instead of JIT-compilation.

8.1.2. Class: CuDevice

Implementing: IDisposable

Methods:

- `string ToString()`
  Returns the device-name as string.
- `void Synchronize()`
  Blocks until the device has completed all requested tasks.
- `Tensor<T[]> Allocate<T>(int size)`
  Accepts a size to be allocated
  Returns a Tensor instance as reference to the allocated array on the GPU-device.
- `Tensor<T[,] Allocate<T>(int sizeX, int sizeY)`
  Accepts x and y sizes of the 2D-array to be allocated.
  Returns a Tensor instance as reference to the allocated array on the GPU-device.
- `Tensor<T[]> Copy<T>(T[] hostTensor) where T : struct`
  Accepts an array of value-type elements as argument, allocating it on the GPU-device.
  Returns a Tensor instance as reference to the allocated array on the GPU-device.
- `Tensor<T[,]> Copy<T>(T[,] hostTensor) where T : struct`
  Accepts a 2D-array of value-type elements as argument, allocating it on the GPU-device.
  Returns a Tensor instance as reference to the allocated 2D-array.
- `Tensor<T> CreateScalar<T>(T hostScalar) where T : struct`
  Accepts a scalar of value-type as argument, preparing it to be used as a parameter to the kernel.
  Returns a Tensor instance as reference to scalar value.

- `T[] Copy<T>(Tensor<T[]> deviceTensor) where T : struct`
  Accepts a Tensor with an array generic type-parameter copying it back to host-memory.
  Returns the array copied back from the GPU-device.
- `T[,] Copy<T>(Tensor<T[,] deviceTensor) where T : struct`
  Accepts a Tensor with a generic 2D-array type-parameter copying it back to host-memory.
  Returns the 2D-array copied back from the GPU-device.
- `void Launch<T0, ..., TN>(Action<T0, ..., TN> kernel, (uint, uint, uint) gridDimensions, (uint, uint, uint) blockDimensions, Tensor<T0> param0, ... , Tensor<TN> paramN)`
  Compiles a kernel-method and launches it with the specified grid and block dimensions as well as arguments.
  Accepts a method reference, a grid-dimensions triple, a block-dimensions triple, up to 10 parameters.
  Scalars may be directly passed as tensor-arguments. They will be implicitly converted to tensors. For this to happen, the generic type parameters must be specified explicitly.

IDisposable implementations:

- `void Dispose()`
  Deallocates resources of the device handle.

8.1.3. Interface: ICuEvent

Implementing: IDisposable
Can be used to measure runtime performance of GPU kernels.

Methods:

- `void Record()`
  Records the event at the current point in time.
- `float GetDeltaTo(CuEvent secondEvent)`
  Calculates the timespan in milliseconds between the event and the event specified as parameter.
  Accepts a CuEvent instance representing the second event.

IDisposable implementation:

- `void Dispose()`
  Deallocates resources of the event handle.

8.1.4. Attribute: Kernel

Constructors:

- `[Kernel]`
  Marks a static method as kernel. Sets the memory-location for locally allocated arrays to: `ArrayMemoryLocation.GLOBAL` (global memory).
- `[Kernel(ArrayMemoryLocation location)]`
  Marks a static method as kernel. Sets the memory-location for locally allocated arrays to `location`.
  Allowed values for `location` are: `ArrayMemoryLocation.GLOBAL` and `ArrayMemoryLocation.SHARED`.

8.1.5. Examples

```
[Kernel]
static void IntAdditionKernel(int[] a, int[] b, int[] result)
{
    uint index = KernelTools.BlockIndex.X *
                 KernelTools.BlockDimensions.X +
                 KernelTools.ThreadIndex.X;

    result[index] = a[index] + b[index];
}
```

Listing 8.1: Kernel Definition Example.

```
Cu.EnableOptimizer = true; //optional
Cu.AotKernelFolder = "./resources"; //optional
var device = Cu.GetDefaultDevice();

var arrayA = new int[] {1,2,3};
var arrayB = new int[] {4,5,6};

var deviceArrayA = device.Copy(arrayA);
var deviceArrayB = device.Copy(arrayB);
var deviceResultArray = device.Allocate<int>(3);

device.Launch(IntAdditionKernel, (1,1,1), (3,1,1),
    deviceArrayA, deviceArrayB, deviceResultArray);

var arrayResult = device.Copy(deviceResultArray);
```

Listing 8.2: Kernel Launch Example.

```
[Kernel]
static void FillArrayWithIntKernel(int[] array, int scalarValue)
{
    uint index = KernelTools.BlockIndex.X *
                 KernelTools.BlockDimensions.X +
                 KernelTools.ThreadIndex.X;

    array[index] = scalarValue;
}
```

Listing 8.3: Kernel Definition Example with Scalar.

```
var device = Cu.GetDefaultDevice();

var deviceArray = device.Allocate<int>(3);

device.Launch<int[], int>(FillArrayWithIntKernel, (1,1,1), (3,1,1),
    deviceArray, 42);

var result = device.Copy(deviceArray);
```

Listing 8.4: Kernel Launch with Scalar Example.

8.2. Ahead-of-Time Compiler

```
CuSharp.AOTC.exe <path to dll> <path to output-folder>
```

The ahead-of-time compiler is an executable taking two arguments as input:

1. Path to the DLL containing the kernel-methods.
2. Path to the output folder for the compiled PTX Kernels.

Kernels to be compiled must be marked with the `[Kernel]` attribute. The name of the output kernel-files must not be changed for the framework to use them instead of JIT compilation.

8.3. Supported Language

CuSharp supports a subset of C#. A detailed description of this subset can be found in Appendix D.

8.4. Publication of the Application and Sourcecode

All sourcecode comprising CuSharp is published on GitHub inside the CuSharp git-repository [20]. CuSharp is also deployed as a NuGet-packages [21] on the official NuGet-gallery [22].

8.4.1. Licensing

All code, developed in the scope of this thesis is licensed under the GNU General Public License v3.0 (GPL3) [23]. GPL3 was chosen to ensure that CuSharp and any derivative work will remain open-source and free of charge to the public.

8.4.2. Git Workflow

Most development of CuSharp was conducted on the main-branch of the git-repository. To keep the main-branch in a working state after the public release of CuSharp, development will only be performed on branches derived from the main-branch and later merged back into the main-branch.

Development branch-names are prefixed *feature_*, for branches intended for development of new features, and *bugfix_*, for branches intendet to fix issues that exist on the main-branch.

Part II.

Conclusion and Outlook

9. Conclusion and Outlook

The thesis results in a working prototype of a framework, to deploy C# methods to the GPU.

9.1. Production Readiness

All example use-cases described throughout this document are thoroughly tested. Despite this, we do not recommend to use CuSharp in safety-critical applications. To make CuSharp ready for development of safety-critical systems, it would need to be verified using more formal techniques than unit-testing.

Still, we think that CuSharp can be used in less than safety-critical production environments. To do so, we recommend testing CuSharp in the specific environment it is to be used in and with the specific kernels to be compiled.

9.2. Outlook

There are multiple areas where CuSharp could be extended to support more features which were not implemented due to the time constraint.

Using CuSharp on a linux system is currently unsupported. To support this, CuSharps libNVVM bindings need to be extended.

Data types other than primitives can currently not be used in kernels. To implement the use of classes, the class declarations need to be compiled to NVVM IR as well.

The initialization overhead through store-operations described in Subsection 7.6.1 is parallelizable by a GPU-kernel itself. Further research and experiments could find out in which cases this would provide a performance benefit.

For reasons specified in Section 7.5, use of the length-property of arrays is not supported. Implementing this, will take a lot of thought to find the most efficient way to do so.

As mentioned in Section 7.6, with the implementation of the length-property of arrays, multidimensional arrays could potentially be implemented more efficiently and more cleanly.

An additional feature that could be implemented in CuSharp is that CuSharp could cross-compile anonymous-functions, making it easier and more readable to deploy small kernels.

Lastly, support for LINQ queries on arrays would generate significant development comfort.

Part III.

Appendix

A. Project Plan

A.1. Important Dates

- 20.02.2023: Project kickoff
- 10.04.2023: One week of vacation
- 13.04.2023: Interim presentation
- 12.06.2023: Start of block week
- 12.06.2023: Abstract submission
- 16.06.2023: Final Submission (report and code)
- 26.06.2023: Final presentation with the bachelor exam

A.2. Working Process

Since the team consists of two people, the process framework "Ration Unified Process (RUP)" is applied to plan the workload of the bachelor thesis. RUP contains four project life-cycle phases covering the following project-specific work areas:

Inception phase, elaboration phase, construction phase and transition phase

A.3. Phases and Milestones

In addition to the phases specified by RUP, the project was more finely divided into the following phases, with a focus on the technical aspects:

- Part 1: Evaluation
- Part 2: MVP (technical breakthrough and meeting all must requirements)
- Part 3: Adding functionality (meeting all should requirements)
- Part 4: Testing and optimization (bringing the cross-compiler to a beta state)
- Part 5: Final Submission (submitting all deliverables)

No.	Milestone	Goal	Due Date	Ass. Part
M1	Evaluation Finalized	Existing technologies and tools were evaluated, and a rough concept was created. All essential decisions were made and documented.	19.03.2023	Part 1
M2	Must-Requirements	The MVP (= all must requirements are fulfilled) has been completed, and the technical breakthrough has been made.	02.04.2023	Part 2
M3	Should-Requirements	All should requirements have been implemented. Optionally, the could requirements have also been fulfilled.	28.05.2023	Part 3
M4	Beta	The cross-compiler was tested and improved. Where possible, it was also optimized. The final product is in a state where it can be released as a beta version.	11.06.2023	Part 4
M5	Final Submission	Final work has been done, and all deliverables have been submitted.	16.06.2023	Part 5

Table A.1.: Milestones.

A.4. Big Picture

The following section contains a rough project plan that was refined in the first part of the project and which was be completed during the project.

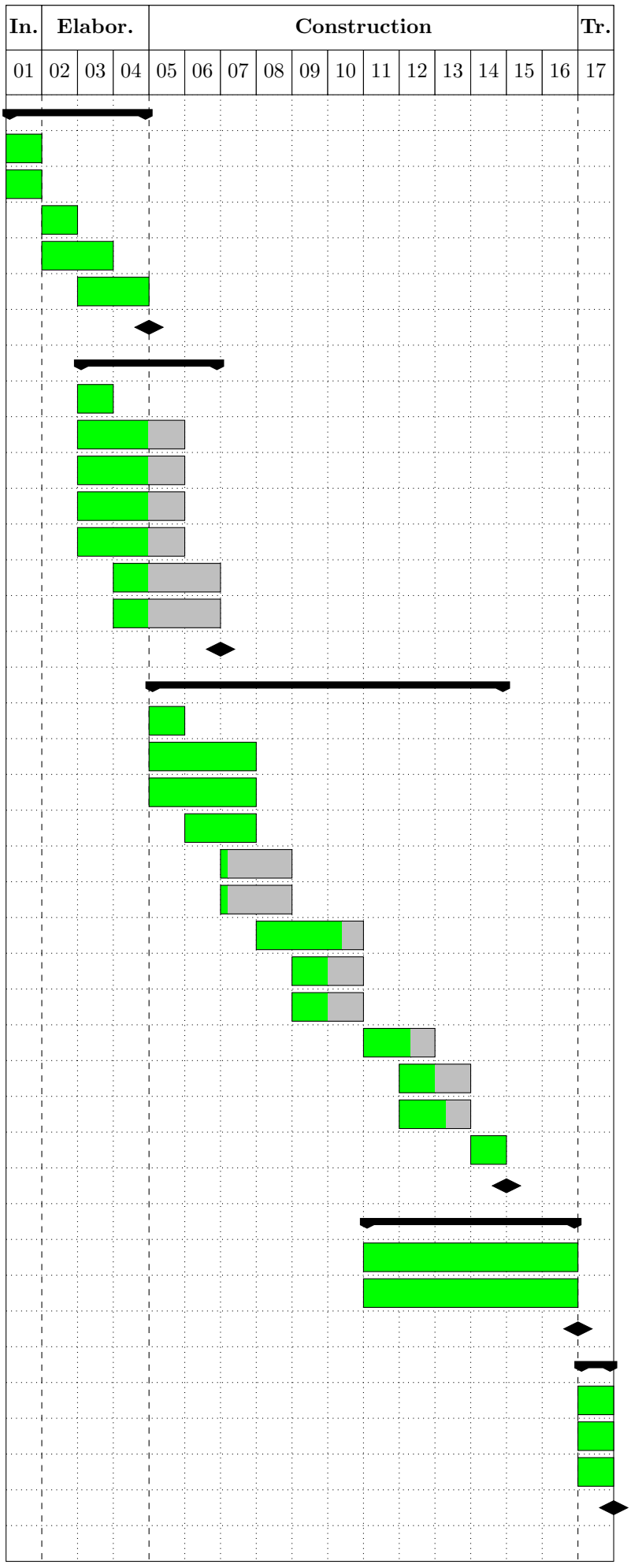
These abbreviations were used in the title bar:

- In. = Inception Phase
- Elabor. = Elaboration Phase
- Tr. = Transition Phase

The color codes used in the diagram have the meaning as follows:

- Entire bar = Total estimated duration of the task
- Green = Required time for the task
- Gray = Not required time for the task (= finished earlier than estimated)

The Gantt chart shows the different project phases and tasks. Based on experience from previous projects, we planned conservatively. In addition, there was enough time in the penultimate phase to test and optimize the product. This schedule worked very well. We often finished ahead of schedule because there were fewer negative surprises than expected.



1. Evaluation
1.1 Initial Research
1.2 Tool-Chain Evaluation
1.3 Tool Evaluation
1.4 Solution Strategy
1.5 Software Architecture
M1: Evaluation Finalized
2. MVP
2.6 Decompilation
2.7 Compilation
2.8 Code Generation
2.9 Frontend and Kernel Discovery
2.10 GPU Bindings and Dispatching
2.11 Technical Breakthrough
2.12 Testing and Improving the MVP
M2: Must-Requirements
3. Adding Functionality
3.13 Adding more Operations
3.14 While- and For-Loops
3.15 If- and Else-Constructs
3.16 Logical Operators
3.17 Launching Kernel Multiple Times
3.18 Launching Different Kernels
3.19 Call Support
3.20 Array Allocation Support
3.21 Shared Memory Allocation Support
3.22 Ahead-of-Time Compilation
3.23 Multidimensional Array Support
3.24 Generics Support
3.25 Bitwise and Shift Operators
M3: Should-Requirements
4. Testing / Optimization
4.26 Optimization and Improvement
4.27 Intensive Testing
M4: Beta
5. Finalization
5.28 Report Finalization
5.29 Code Finalization
5.30 Final Submission
M5: Final Submission

Table A.2.: Project Plan as Gantt Diagram.

B. Personal Reports

B.1. Adrian Locher

Developing compilers, to me, is the single most interesting discipline in software-engineering. I was certain from the start that this thesis would pose a challenge I could sink my teeth in. I was not disappointed. While developing the framework resulting from this thesis, I learned a lot about compiler and library design, GPUs and formal languages.

While being the challenge that I've anticipated, I'm now reassured that I'm ready for the challenges that await me as a software-engineer in the industry. Having worked on most projects of my studies together with Jason Benz, we are a well rehearsed team and development of the CuSharp framework ran as flawlessly as I had hoped for.

B.2. Jason Benz

The backend is my preferred area in software engineering and through the module 'Compiler Construction' I came into contact with a compiler for the first time from a compiler developer's point of view. This aroused my interest in this topic and therefore this thesis was exactly what I was looking for.

While I was able to apply some hard skills I have learned in my studies to this thesis, I quickly encountered implementation challenges. With a lot of research and testing I was able to solve them. I was able to learn a lot during this thesis and broaden my horizons. Not only was I able to learn technically, but also how to approach problems that are not so widespread and well documented. I think this will also help me on my further path in life.

Since we completed most of the projects together during our studies, Adrian Locher and I are a very well-practiced team. The collaboration with him was great, as usual.

C. Compiler Specification

This chapter describes the most important traits of the behavior of the CuSharp cross-compiler.

C.1. Introduction

CuSharp is an MSIL to LLVM IR cross-compiler that enables its user to execute C# code on Nvidia GPUs. The framework facilitates the compilation of annotated or manually selected methods from MSIL to PTX via the LLVM compiler backend for CUDA. CuSharp supports a subset of C#, which is described in Chapter D.

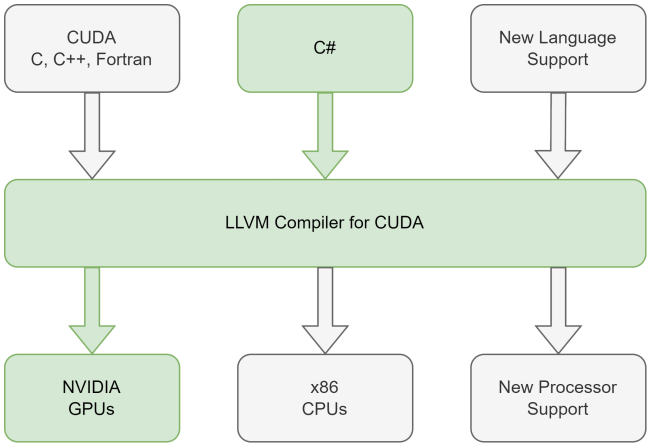


Figure C.1.: LLVM Structure with C# Language Support.

C.2. Tested Version

CuSharp is a cross-compiler that was specifically designed for the C# programming language. While other .NET languages - or subsets of them - may be supported, they have not been tested. The latest version of CuSharp has been tested for compatibility with C# 11.0, respectively .NET 7.0, compiled by the Roslyn compiler. Only the language features specified below are supported.

C.3. Configuration

The compilation process to NVVM requires a configuration to be specified. The default configuration includes:

Setting Name	Description	Default Value
DataLayout	Data layout for Nvidia GPUs	Default Nvidia data layout
Target	Target architecture for the compiled code	nvptx64-nvidia-cuda
KernelName	Name of the kernel. Needs to be overwritten.	""
DefaultArrayMemoryLocation	Default memory location of arrays allocated with 'new ...' inside of a Kernel. Can be overwritten with the kernel attribute.	GLOBAL
DeclareAnnotations	Declaration of required named metadata	Default NVVM annotations
DeclareExternalFunctions	Declaration of NVVM IR intrinsic functions	See Subsection C.3.1

Table C.1.: NVVM Configuration.

When creating the configuration, the values can also be set manually if adjustments are required or a different target device needs to be used.

C.3.1. NVVM IR Intrinsic Functions

NVVM IR (Intermediate Representation) declares a series of functions that are specific to the NVPTX (NVIDIA Parallel Thread Execution) architecture. These functions allow the program to access various system-level parameters, such as block-index, block dimension and thread-index, which is required for implementing parallel programs on NVIDIA GPUs. These functions are essential for thread and block coordination.

Since C# does not provide these intrinsic functions, the static class 'KernelTools' is introduced. It contains properties which can be used in the C# source code to execute the intrinsic functions. The mapping of the properties to the functions is defined in the configuration object.

The functions for grid dimension, block index, thread index and block dimension always have the suffixes `x()`, `y()` and `z()` to access the corresponding axis. The equivalent on the properties are the fields `X`, `Y` and `Z`:

C# Property	NVVM IR Intrinsic Function	Description
`GridDimension`	`@llvm.nvvm.read.ptx.sreg.nctaid`	Returns the total number of blocks along an axis (X, Y or Z) of a grid.
`BlockIndex`	`@llvm.nvvm.read.ptx.sreg.ctaid`	Returns the IDs of the calling block.
`ThreadIndex`	`@llvm.nvvm.read.ptx.sreg.tid`	Returns the IDs of the calling thread.
`BlockDimensions`	`@llvm.nvvm.read.ptx.sreg.ntid`	Returns the total number of threads along an axis (X, Y or Z) in a block.
`SyncThreads()`	`@llvm.nvvm.barrier0()`	Inserts a barrier for synchronizing threads in a block. Waits until all threads in the thread block have reached this point [24].
`GlobalThreadFence()`	`@llvm.nvvm.membar.gl()`	Is a memory fence at the device level [24]. Inserts a barrier for synchronizing threads across all blocks within the grid.
`SystemThreadFence()`	`@llvm.nvvm.membar.sys()`	Is a memory fence at the system level [24]. Inserts a barrier for synchronizing between the CPU and GPU.
`WarpSize`	`@llvm.nvvm.read.ptx.sreg.warpsize()`	Returns the size of the warp which is the number of threads that can be executed in parallel on a single instruction in a SIMD architecture.

Table C.2.: LLVM IR Intrinsic Functions.

The mapping from the KernelTools properties to the intrinsic functions is defined in the configuration file. The example below shows which NVVM IR output is generated when using the KernelTools:

```
// Usage in a C# method to determine the correct index
int i = (int) (KernelTools.BlockIndex.X *
               KernelTools.BlockDimensions.X +
               KernelTools.ThreadIndex.X);

// NVVM IR output with the usage of the intrinsic functions
%reg0 = call i32 @llvm.nvvm.read.ptx.sreg.ctaid.x()
%reg1 = call i32 @llvm.nvvm.read.ptx.sreg.ntid.x()
%reg2 = mul i32 %reg0, %reg1
%reg3 = call i32 @llvm.nvvm.read.ptx.sreg.tid.x()
%reg4 = add i32 %reg2, %reg3
```

Listing C.1: Usage Example and Mapping of Intrinsic Functions.

C.4. Launching Kernels

Kernels can be defined and launched as described in Section 8.1. Examples are available in Subsection 8.1.5.

C.5. Kernel Attribute

A kernel can be annotated with the `[Kernel]` attribute (see Subsection 8.1.4) to be discovered by the AOT-compiler.

C.6. Optimizer

CuSharp provides a static property *'Cu.EnableOptimizer'*. By default, the optimizer is enabled in release mode and disabled in debug mode. If enabled, the optimizations from the following sections will be performed.

C.6.1. Built-In Optimizations

There are a number of built-in optimization steps in the LLVM compiler infrastructure that significantly improve performance. It is important to execute them in the right order to achieve the maximum performance gain. [25]

1. Supporting basic alias analysis for global value numbering (GVN)
2. Promoting allocas to registers (replaces stack-allocated variables with register-allocated variables)
3. Doing simple 'peephole' optimizations and bit-twiddling optimizations
4. Reassociating expressions
5. Simplifying the control flow graph (deleting unreachable blocks, etc.)
6. Eliminating common subexpressions

D. Language Specification

This chapter is directed at developers that want to use CuSharp to write and deploy GPU-kernels. It describes the C# subset that is supported by CuSharp.

D.1. Supported C# Features

D.1.1. Supported Types

C# Type	LLVM Type	Description
`void`	void	Void, represents the absence of a value
`bool`	`i1`	Boolean value, aligned to 1 byte
`byte`	`i8`	Byte value, aligned to 1 byte
`short`	`i16`	Short value, aligned to 2 bytes
`int`	`i32`	Integer value, aligned to 4 bytes
`uint`	`i32`	Unsigned integer value, aligned to 4 bytes
`long`	`i64`	Long integer value, aligned to 8 bytes
`float`	`float`	Floating point value, aligned to 4 bytes
`double`	`double`	Double floating point value, aligned to 8 bytes
`T[]`	`T*`	Array of numeric elements
`T[,]`	`T**`	Two dimensional array of numeric elements, may be slower than single dimensional arrays

Table D.1.: Supported Types.

D.1.2. Generic Types

Generic types can be used in a kernel. For mathematical calculations the constraint `INumber<T>` can be used. Allocations of type T within the kernel are enabled with the constraint `new()`. `new()` will initialize a variable with its default value. A possible kernel method signature with generics can look like this:

```
public static void KernelName<T>(T param) where T : INumber<T>, new()
```

D.1.3. Control Flow Statements

The following control flow statements are supported. They can be nested.

Statement	Description
if-else statement	Used for conditional branching based on the evaluation of a Boolean expression.
switch statement	Used for selecting one of several possible code blocks to execute based on the value of an expression.
for loop	Used for executing a code block a specific number of times.
while loop	Used for executing a code block repeatedly as long as a Boolean expression evaluates to true.
do-while loop	Used for executing a code block repeatedly as long as a Boolean expression evaluates to true, but the block is executed at least once regardless of the initial evaluation of the expression
break statement	Used for breaking out of a loop or switch statement.
continue statement	Used for skipping the rest of the current iteration of a loop and moving on to the next iteration.
goto statement	Used for jumping to a labeled statement in the code.

Table D.2.: Supported Control Flow Statements.

D.1.4. Arithmetic Operators

All arithmetic operators are supported as expanded version (e.g. `a = a + b`) and as shorthand version (e.g. `a += b`).

Sign	Name	Description
`+`	Addition	Adds two operands together.
`-`	Subtraction	Subtracts the right operand from the left operand.
`*`	Multiplication	Multiplies two operands together.
`/`	Division	Divides the left operand by the right operand.
%	Remainder	Returns the remainder of the division of the left operand by the right operand.

Table D.3.: Supported Arithmetic Operators.

D.1.5. Logical Operators

If possible the operators are evaluated short circuit.

Sign	Name	Description
`&&`	Logical AND	Returns true if both operands are true, otherwise returns false.
`||`	Logical OR	Returns true if at least one operand is true, otherwise returns false.
`!`	Logical NOT	Reverses the logical state of its operand. If the operand is true, returns false. If the operand is false, returns true.

Table D.4.: Supported Logical Operators.

D.1.6. Bitwise Operators

Sign	Name	Description
`&`	Bitwise AND	Performs a bitwise AND operation between two operands, resulting in a new value where every bit is set to 1 if both bits of the operands are the same, otherwise it is set to 0.
`|`	Bitwise OR	Performs a bitwise OR operation between two operands, resulting in a new value where every bit is set to 1 if at least one bit is 1, otherwise it is set to 0.
`^`	Bitwise XOR	Performs a bitwise XOR operation between two operands, resulting in a new value where every bit is set to 1 if the bits of the operands are different, otherwise it is set to 0.
`~`	Bitwise NOT	Flips every bit.
`<<`	Shift Left	Shifts the bits of an operand to the left by a specified number of positions.
`>>`	Signed Right Shift	Sifts the bits of an operand to the right by a specified number of positions.
`>>`	Unsigned Right Shift	Shifts the bits of an operand to the right by a specified number of positions and fills the shifted positions with zeros.

Table D.5.: Supported Logical Operators.

D.1.7. Comparison Operators

Sign	Name	Description
`<`	Less than	Returns true if the left operand is less than the right operand.
`<=`	Less than or equal to	Returns true if the left operand is less than or equal to the right operand.
`>`	Greater than	Returns true if the left operand is greater than the right operand.
`>=`	Greater than or equal to	Returns true if the left operand is greater than or equal to the right operand.
`==`	Equal to	Returns true if the left and right operands are equal.
`!=`	Not equal to	Returns true if the left and right operands are not equal.

Table D.6.: Supported Comparison Operators.

D.1.8. Calls

Kernel Tools

The following table lists the kernel tool functions and variables with there CUDA C++ equivalent.

C# Property	CUDA C++ Equivalent
`KernelTools.GridDimension.[X|Y|Z]`	`gridDim.[x|y|z]`
`KernelTools.BlockIndex.[X|Y|Z]`	`blockIdx.[x|y|z]`
`KernelTools.ThreadIndex.[X|Y|Z]`	`threadIdx.[x|y|z]`
`KernelTools.BlockDimensions.[X|Y|Z]`	`blockDim.[x|y|z]`
`KernelTools.SyncThreads()`	`__syncthreads()`
`KernelTools.GlobalThreadFence()`	`__threadfence()`
`KernelTools.SystemThreadFence()`	`__threadfence_system()`
`KernelTools.WarpSize`	`warpSize`

Table D.7.: C# Kernel Tools with CUDA C++ Equivalent.

Nested Calls

Static methods can be called from a kernel. These methods can have a numeric return type (primitive and as array) or void. The method calls can be nested over several levels. Primitive data types or numeric arrays can be passed as parameters. The primitive data types are passed as *call by value*, the arrays are passed as *call by reference*.

D.1.9. Implicit Casts

All implicit casts in C# such as `int` to `long` work.

D.2. Unsupported Features

All features that are not described in the preceding chapters are generally unsupported. It may be that individual functionalities work anyway, but they have not been tested or checked for side effects.

There are some C# functions that one would expect to work, but that are unsupported at this time. These are, including but not limited to:

- Array length property
- Array initialization using an array initializer with values instead of parameters and variables (e.g. `var a = new int[]{1,2,3};`)
- Foreach loop (due to the missing array length property)
- Any calls to non-static methods and non-intrinsic functions
- Anonymous functions i.e. lambdas
- LINQ queries (query syntax and method syntax)
- Data types other than primitives
- Native int
- Overflow checks by using the *checked* keyword
- ...

E. Evaluation and Example Code

E.1. Double-Matrix Multiplication Kernel

```
[Kernel]
public static void MatrixMultiplication(double[] a, double[] b, double[] c,
    int matrixWidth)
{
    var row = KernelTools.BlockDimension.Y * KernelTools.BlockIndex.Y +
        KernelTools.ThreadIndex.Y;
    var col = KernelTools.BlockDimension.X * KernelTools.BlockIndex.X +
        KernelTools.ThreadIndex.X;
    double result = 0.0;
    if (row < matrixWidth && col < matrixWidth)
    {
        for (int i = 0; i < matrixWidth; i++)
        {
            result += a[matrixWidth * row + i] * b[i * matrixWidth + col];
        }

        c[row * matrixWidth + col] = result;
    }
}
```

Listing E.1: Matrix Multiplication Kernel.

E.2. Tiled Double-Matrix Multiplication Kernel

```
[Kernel(ArrayMemoryLocation.SHARED)]
public static void TiledMatrixMultiplication(double[] a, double[] b, double
    [] c, int matrixWidth, int tileWidth, int nofTiles)
{
    var tx = KernelTools.ThreadIndex.X;
    var ty = KernelTools.ThreadIndex.Y;
    var col = KernelTools.BlockIndex.X * tileWidth + tx;
    var row = KernelTools.BlockIndex.Y * tileWidth + ty;

    var aSub = new double[1024];
    var bSub = new double[1024];

    double sum = 0;
    for (int tile = 0; tile < nofTiles; tile++)
    {
        if (row < matrixWidth && tile * tileWidth + tx < matrixWidth)
        {
            aSub[ty * tileWidth + tx] = a[row * matrixWidth + tile *
                tileWidth + tx];
        }

        if (col < matrixWidth && tile * tileWidth + ty < matrixWidth)
        {
            bSub[ty * tileWidth + tx] = b[(tile * tileWidth + ty) *
                matrixWidth + col];
        }

        KernelTools.SyncThreads();

        if (row < matrixWidth && col < matrixWidth)
        {
```

```
29              for (int ksub = 0; ksub < tileWidth; ksub++)
30              {
31                  if (tile * tileWidth + ksub < matrixWidth)
32                  {
33                      sum += aSub[ty * tileWidth + ksub] * bSub[ksub *
                            tileWidth + tx];
34                  }
35              }
36          }
37          KernelTools.SyncThreads();
38      }
39      if (row < matrixWidth && col < matrixWidth)
40      {
41          c[row * matrixWidth + col] = sum;
42      }
43  }
```

Listing E.2: Tiled Matrix Multiplication Kernel.

E.3. 2D-Array Matrix Multiplication Kernel

```
1  [Kernel]
2  public static void MultiDimMatrixMultiplication(double[,] a, double[,] b,
       double[,] c, int matrixWidth)
3  {
4      var row = KernelTools.BlockDimension.Y * KernelTools.BlockIndex.Y +
           KernelTools.ThreadIndex.Y;
5      var col = KernelTools.BlockDimension.X * KernelTools.BlockIndex.X +
           KernelTools.ThreadIndex.X;
6
7      double result = 0;
8      if(row < matrixWidth && col < matrixWidth)
9      {
10         for(int i = 0; i < matrixWidth; i++)
11         {
12             result += a[row, i] * b[i, col];
13         }
14         c[row, col] = result;
15     }
16 }
```

Listing E.3: 2D-Array Matrix Multiplication Kernel.

E.4. Mandelbrot Kernel

```
1  [Kernel]
2  public static void MandelBrot(float[] light,  int maxIterations, int N,
       float zoom, float deltaX, float deltaY)
3  {
4      var row = KernelTools.BlockDimension.Y * KernelTools.BlockIndex.Y +
           KernelTools.ThreadIndex.Y;
5      var col = KernelTools.BlockDimension.X * KernelTools.BlockIndex.X +
           KernelTools.ThreadIndex.X;
6
7      if (row < N && col < N)
8      {
9          float fromX = col / zoom - deltaX;
10         float fromY = row / zoom - deltaY;
11         float x = 0.0f;
12         float y = 0.0f;
```

```
13          int iteration = 0;
14          while (x * x + y * y <= 2 * 2 && iteration < maxIterations)
15          {
16              var xtemp = x * x - y * y + fromX;
17              y = 2 * x * y + fromY;
18              x = xtemp;
19              iteration++;
20          }
21
22          light[row * N + col] = iteration;
23      }
24  }
```

Listing E.4: Mandelbrot Kernel.

F. Performance Results

F.1. Double Matrix Multiplication in Global Memory

The following numbers are the average of 10 time measurements, of the kernel specified in Section E.1, in milliseconds.

CUDA GTX 970	CUDA RTX 3060TI	CuSharp GTX 970	CuSharp RTX 3060 TI
23.19	8.42	24.55	8.98
28.82	10.02	29.46	10.29
36.73	12.22	37.87	12.96
45.37	15.24	46.85	16.14
55.94	19.16	57.46	20.2
68.82	23.94	70.34	24.53
88.09	28.69	89.97	29.74
101.48	35.44	103.6	36.17
118.13	40.37	120.21	42.76
143.98	47.52	144.85	50.07
169.17	55.18	171.46	58.0

F.2. Double Matrix Multiplication Tiled using Shared Memory

The following numbers are the average of 10 time measurements, of the kernel specified in Section E.2, in milliseconds.

CUDA GTX 970	CUDA RTX 3060TI	CuSharp GTX 970	CuSharp RTX 3060 TI
24.6	9.17	24.94	10.53
27.42	9.58	28.12	10.3
32.45	11.35	32.87	11.83
37.99	13.2	38.44	13.92
43.83	15.37	44.36	15.97
50.24	17.78	50.77	18.37
56.77	19.83	57.43	20.51
65.2	22.7	66.3	23.58
72.85	26.26	74.15	26.26
80.96	28.26	82.04	29.25
89.39	31.21	90.21	32.19

G. Bibliography

[1] "CUDA Toolkit." https://developer.nvidia.com/cuda-toolkit. Accessed: 2023-02-25.

[2] "C# Compiler Roslyn." https://github.com/dotnet/roslyn. Accessed: 2023-02-25.

[3] "Vulkan." https://www.vulkan.org/. Accessed: 2023-02-23.

[4] "OpenCL." https://www.khronos.org/opencl/. Accessed: 2023-02-22.

[5] "LLVM Compiler Structure." https://developer.nvidia.com/cuda-llvm-compiler. Accessed: 2023-02-25.

[6] "Alea GPU." https://developer.nvidia.com/blog/accelerate-net-applications-alea-gpu/. Accessed: 2023-05-16.

[7] "Altimesh Hybridizer." https://developer.nvidia.com/blog/hybridizer-csharp/. Accessed: 2023-05-16.

[8] "CUDAfy.NET." https://github.com/lepoco/CUDAfy.NET. Accessed: 2023-05-16.

[9] "AMD HIP API." https://github.com/ROCm-Developer-Tools/HIP. Accessed: 2023-02-22.

[10] "OpenCL Adopters." https://www.khronos.org/conformance/adopters/conformant-companies. Accessed: 2023-03-14.

[11] "Kompute API." https://kompute.cc/. Accessed: 2023-02-23.

[12] "Cuda Driver API." https://docs.nvidia.com/cuda/cuda-driver-api/index.html. Accessed: 2023-02-27.

[13] "The C4 model for visualising software architecture." https://c4model.com/. Accessed: 2023-02-03.

[14] "LLVMSharp Project." https://github.com/dotnet/LLVMSharp. Accessed: 2023-03-01.

[15] "managedCuda Project." https://kunzmi.github.io/managedCuda/. Accessed: 2023-03-02.

[16] L. Bläser, *Modern Runtime System and Compiler Design*, ch. 13.9. Local Register Allocation, pp. 361–363. Independent, 2019.

[17] L. Bläser, *Modern Runtime System and Compiler Design*, ch. 8.11. Static Single Assignment, pp. 182–185. Independent, 2019.

[18] L. Bläser, *Modern Runtime System and Compiler Design*, ch. 8.13.1. Control Flow Graph, pp. 191–193. Independent, 2019.

[19] Nvidia, *NVVM IR Specification v2.0*, ch. 10.6.1 alloca instruction. Nvidia, 2023.

[20] "CuSharp Git-Repository." https://github.com/CuSharp/CuSharp/, 2023.

[21] "CuSharp Nuget Packages." https://https://nuget.org/profiles/CuSharp, 2023.

[22] "NuGet Gallery." https://www.nuget.org/, 2023.

[23] "Gnu general public license v3.0." https://www.gnu.org/licenses/gpl-3.0.en.html. Accessed: 2023-05-18.

[24] Nvidia, *NVVM IR Specification v2.0*, ch. 15.2. Barrier and Memory Fence. Nvidia, 2023.

[25] "Llvmsharp optimization tutorial." https://github.com/dotnet/LLVMSharp/blob/main/samples/KaleidoscopeTutorial/Chapter4/KaleidoscopeLLVM/Program.cs. Accessed: 2023-04-15.

H. List of Figures

I. List of Tables

J. List of Listings