Karlsruhe Institute of Technology

# Automatic Loop Partitioning for Heterogeneous Systems

Bachelor's Thesis of


## Alexander Baier

At the Department of Informatics
Institut für Programmstrukturen
und Datenorganisation (IPD)


| | |
|---|---|
| Reviewer: | Prof. Dr. Walter F. Tichy |
| Second reviewer: | Prof. Dr. Ralf H. Reussner |
| Advisor: | Dipl.-Inform. Philip Pfaffe |
| Second advisor: | Dipl.-Inform. Martin Tillmann |

Duration: 15.08.2015 – 14.01.2016

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

I followed the rules for securing a good scientific practise of the Karlsruhe Institute of Technology (Regeln zur Sicherung guter wissenschaftlicher Praxis im Karlsruher Institut für Technologie (KIT)).

**Karlsruhe, 14.01.2016**

.....

(**Alexander Baier**)

**Kurzfassung**

Heutzutage ist jeder Desktopcomputer, und auch ein stets steigender Anteil von mobilen Geräten, mit Mehrkernprozessoren ausgestattet. Einige Geräte verfügen sogar über weitere Prozessoren, wie z.B. eine dedizierte Grafikkarte (GPU), welche tausende zusätzliche Kerne anbietet. Diese zusätzliche Rechenleistung kann von sequenziellen Programmen nicht ausgenutzt werden. Deshalb müssen Programme geschrieben werden, welche mit mehreren verschiedenen Prozessoren umgehen können. Allerdings ist das Erstellen von Programmen für solche heterogenen Systeme eine schwierige Aufgabe: Erstens, bieten die verschiedenen Prozessoren auch verschiedene Programmierschnittstellen an und haben verschiedene Stärken und Schwächen. Zweitens, ist eine parallele Lösung im Generellen komplexer als ihr sequentielles Gegenstück. Drittens muss der Programmierer selbst entscheiden, welcher Teil der Berechnung auf welchem Prozessor ausgeführt wird. Die richtige Verteilung der zu erledigenden Arbeit ist ein zeitintensiver und fehleranfälliger Vorgang. Es ist allerdings auch ein sehr wichtiger Vorgang, denn falsche Entscheidungen führen hier zu zeitweise ungenutzen Prozessoren und anschließend zu einem Performanzverlust. In dieser Arbeit beabsichtigen wir diese Schwierigkeiten zu beseitigen, indem wir ein Werkzeug einführen, welches automatisiert Schleifen partitioniert und diese auf einem heterogenen, CUDA fähigen System ausführt. Das automatisierte Transformieren von sequentiellen Programmen, sodass diese auf mehreren Prozessoren gleichzeitig laufen können, löst die ersten beiden Probleme: Weder Wissen über parallele Programmierung, noch über spezielle Prozessoren (wie z.B. Grafikkarten) ist vonnöten. Das dritte Problem lösen wir, indem wir einen Tuning-Algorithmus einsetzen, welcher automatisiert die optimale Verteilung der Rechenarbeit über die vorhandenen Prozessoren bestimmt. Dies erlaubt es dem Programmierer einfache sequentielle Anwendungen zu schreiben und trotzdem die zusätzliche Rechenleistung zu nutzen, die durch die moderne Hardware zu Verfügung stehen. Wir evaluieren unseren Ansatz indem wir Algorithmen aus dem Gebiet der Linearen Algebra parallelisieren. Die Ergebnise zeigen, dass wir damit einen Performanzgewinn von bis zu 1.45 erreichen, verglichen mit der sequentiellen Version dieser Algorithmen.

**Abstract**

Today every desktop computer and an increasing amount of mobile devices provide multi-core CPUs. Some devices even feature a dedicated graphics processing unit (GPU) that may provide thousands of additional cores. This extra processing power cannot be exploited by writing single threaded programs. Instead programs have to be implemented with multiple concurrent processing units in mind. Writing programs to run in such a heterogeneous environment is a difficult task: Firstly, different processing units expose different programming interfaces and exhibit different strengths and weaknesses. Secondly, a parallel solution is generally more complex than its sequential counter part. Thirdly, the programmer has to decide which part of the computation is executed by which processing unit. Finding the right distribution of work is a time-consuming and error prone process. It is, never the less, an important step as making the wrong decision leads to idle processing units and subsequently to a loss in performance. In this thesis, we aim to address these difficulties by implementing a tool that automatically partitions loops and then executes these partitions on a heterogeneous system with a CUDA enabled GPU. Automatically transforming sequential programs to run in parallel on multiple processing units solves the first two problems: Neither knowledge of parallel programming nor of programming for specific processing units would be required. We address the third problem by utilizing a tuning algorithm which automatically finds the optimal distribution of work over available processing units. This would allow the programmer to write simple sequential programs while still harnessing the additional processing power provided by modern hardware. We evaluate our approach by parallelizing algorithms from the domain of linear algebra. The results show, that we achieve a speedup of up to 1.45 compared to the sequential version of the algorithms.

# Contents

# List of Figures

# List of Tables

# 1. Introduction

In this work, we implement a tool that automatically partitions loops and then executes these partitions on heterogeneous systems. Partitioning a loop is the process of dividing a loop to form two or more new loops, each iterating over a portion of the original loops iteration space. A heterogeneous system is a system that is equipped with, and uses, different kinds of processing units. An example of this is a system that has access to a CPU and a GPU.

The remainder of this chapter is structured as follows: In section 1.1 we motivate the need for our tool, before we state the goals we want to achieve with this work in section 1.2. Finally we present the structure of this document in section 1.3.

## 1.1 Motivation

In 1975 Moore's Law predicted the number of transistors on a chip to double every two years. This prediction has correctly approximated the increase in transistors and thus the rising clock speed of processors for 30 years. The stagnation of this growth in the early 2000s can be ascribed to physical issues arising from smaller and smaller transistors: It is too hard to effectively dissipate the heat and the power consumption is too high.[Sut] In order to uphold the increasing clock speed, the hardware industry integrates multiple CPUs on a single chip. In addition to multiple cores in a single processor, the search for more computing power has lead to the utilisation of formerly special purpose processors like graphics processing units (GPU). Originally only used for graphics and image processing, newer GPUs today additionaly support general purpose computing. General purpose computing allows a programmer to harness the power of GPUs to solve problems differing from graphics computation. These include, but are not limited to, problems in fields such as machine learning, medical imaging, cryptography and data mining.[CBM+08] In comparison to CPUs, GPUs have been—and still are—massively parallel offering 100s or 1000s of cores per processing unit.

In contrast to the increasingly parallel hardware world, a large part of software solutions is still completely or to a great extent expressed as sequential programs and is thus unable to profit from advancing hardware. Moving from sequential to parallel programming, however, presents a significant paradigm shift. This means that patterns used in the one paradigm have different or detrimental implications when used in the other. An example of this is mutable state: A sequential program generally makes extensive use of mutable state

to model side effects. A parallel solution, on the other hand, is often designed to eliminate or at least minimize the amount of shared mutable state. As a programmer, learning a new paradigm can be very costly and time intensive. Additionally, a lot of software in use today was not build with concurrency in mind—it had or would have to be added as an afterthought. This leads to a further increase in complexity, because the programmer has to deal with the overhead of the existing solution, instead of being able to design a parallel one from the start. Further challenges are posed by targeting dedicated devices, like GPUs, as both—the host and the dedicated processor—only have access to their own dedicated memory. This entails, that data held in host memory has to be explicitly transferred to and from the GPU, before and after the execution of a kernel.

Simply knowing how to parallelize parts of a sequential program is not enough to positively affect the execution speed—further questions have to be answered. We need to determine which parts of the program are actually worth parallelizing. As an example, we might encounter a part whose structure lends itself nicely to parallelization, but which only represents a very small amount of the overall workload. The overhead introduced by managing the concurrency might outweigh the increases in performance achieved by the added parallelism. In the case of dedicated devices this overhead can be very high, due to the need to transfer memory back and forth.

Another decision that has to be made, is how to distribute the workload over the available processors. One way to decide this is by trial and error: Test multiple different configurations and choose the best one. This manual approach, however, is very time consuming and relies on programmer intuition rather than a deterministic algorithm. Autotuning can remedy this problem, by automating and systematizing the manual approach. The profiling results of one test run are used to determine the configuration for the next one. The best configuration is found by minimizing a cost function.

## 1.2 Goals

With the rising availablity of parallel systems and general purpose GPUs, sequential solutions need to be adapted to be able to use the full power provided to them. In the previous section, we mention three challenges that emerge when transforming a sequential program to make use of highly parallel hardware. First, parallel programming in general—and GPU programming in particular—require a deep level of additional knowledge compared to conventional sequential programming. By automatically parallelizing sequential parts, our tool helps programmers lacking the necessary skills to take advantage of parallel hardware. Second, annotations or markers in the input program are not needed for our tool to find suitable sections for parallelization – this is also done automatically. The final challenge our tool solves on behalf of the programmer, is optimizing the transformed program. It automatically and dynamically decides how the given workload has to be distributed over available processing units to minimize execution time.

## 1.3 Document Structure

The remainder of this document is structured as follows. Chapter 2 explains the concepts and technologies used in this work. These are the relevant APIs and tools in the LLVM Compiler Infrastructure project, an introduction to autotuning and the parallel computing platform CUDA. In chapter 3 we show how other approaches have been used to solve problems in the areas of code generation for GPUs, heterogeneous code execution and autotuning. In chapter 4 we continue with the concept of this work, outlining our goals in greater detail, how we detect parallelism, how we generate code for the GPU and how we integrate the autotuner into our solution. The concrete implementation of our tool is

then described in chapter 5, where we explain in detail the implementation of the concepts outlined in the previous chapter. Finally, in chapter 7 we summarize this work and give an outlook on further steps to be taken.

# 2. Foundations

This chapter describes technologies and concepts used in this work. The first section 2.1 introduces the compiler framework LLVM by explaining the parts that are relevant to gain an understanding of this work. We then move on to explain the concept of autotuning and discuss the advantages and disadvantages of it in section 2.2. Finally we describe the parallel computing platform CUDA and its virtual machine for parallel thread execution (PTX) in section 2.3.

## 2.1 LLVM

The LLVM Compiler Infrastructure is a collection of projects related to compiler technologies. These projects include among others a set of libraries called LLVM Core, a C/C++/Objective-C compiler and debugger and a back-end for the GNU Compiler Collection. We will focus on LLVM Core which is used for code analysis, optimization and generation. LLVM Core libraries do not process a high level language directly but need a compiler front-end to first compile said language to LLVM IR. LLVM IR stands for LLVM intermediate representation, which is similar to an assembly language in static single assignment (SSA) form. After the code has been processed by LLVM Core, a compiler back-end translates it to native machine code.

The remainder of this section will explain the LLVM IR in greater detail in section 2.1.1 and describe how the analyses and optimizations are organized in section 2.1.2. It will then describe how LLVM IR represents loops in section 2.1.3, how Scalar Evolution is structured in section 2.1.4 and how Polly finds parallelizable loops in section 2.1.5.

### 2.1.1 Intermediate Representation

This subsection gives a short overview over the structure and key elements of code expressed in LLVM IR.

The top-level construct is a module containing globals (global variables and functions). A module is seen as a single compilation unit to be later linked by the LLVM linker. When the linker combines multiple modules it merges their globals and the entries in their respective symbol tables and resolves any forward declarations.

A global variable holds a pointer to a region of memory. The allocation happens at compile time in contrast to local variables which are allocated at runtime.

```
x = 9
if (x < n) then x = x + 1
y = x + 2
```

Figure 2.1: A variable that has a different value in two branches of the CFG expressed as pseudo code.

```
if.cond:
 %x = 9
 %cmp = icmp ult i32 %x, %n
 br i1 %cmp, label %if.true, label %if.end
if.true:
 %x.1 = add i32 %x, 1
 br if.end
if.end:
 %y = add i32 %?, 2
```

Figure 2.2: A variable that has a different value in two branches of the CFG expressed in LLVM IR.

A function in the IR is similar to a function in higher level languages. Its signature consists of a return type, a name, an argument list and a body. The argument list contains zero or more arguments each of which has a name and a type. The body is compromised of one or more basic blocks, which in turn consist of one or more instructions. The basic blocks each end in a terminator instruction like a branch or a return instruction and thus form the nodes of the control flow graph (CFG) of the function.

An instruction has one operator operating on one or more operands. Instructions that do not manipulate the control flow produce a result which is stored in a variable. After this variable has been assigned it can never be written again. Results of subsequent computiations must be stored in new variables. This property is known as static single assignment (SSA) and simplifies or even enables certain compiler optimizations.

The SSA form poses a problem when a variable is altered in two different control flow branches that are later rejoined. The if statement in figure 2.1 demonstrates the problem. The actual value of $x$ in line 3 depends on the control flow of the program: If $n$ is smaller than or equal to 9, $x$'s value will be 9, otherwise it will be 10. After lowering this pseudo code to LLVM IR we will get code similar to that in figure 2.2. Now we do not know which of the two ($x$ or *x.1*) we should use as an operand to the add instruction in the last line. This is solved by introducing a so called phi node into the CFG. In LLVM IR this node is represented as an instruction taking a mapping of basic blocks to values as an argument. When the phi instruction is executed it will return a value based on the path the control flow took to reach the phi node. A phi instruction for our example is depicted in figure 2.3. If the control flow comes from the *if.cond*, block phi returns the value of $x$; if it comes from *if.true*, phi returns the value of *x.1*.

### 2.1.2 Pass Management

In LLVM analyses and transformations are expressed as so called passes. The pass manager schedules these passes based on certain criteria with the goal of optimizing the compile time. Some passes depend on certain analysis results to be available before they can do their work. Others can only be run after a certain transformation has taken place. Each

```
; ...
if.end:
 %phi = phi i32 [%x, %if.cond], [%x.1, %if.true]
 %y = add i32 %phi, 2
```

Figure 2.3: A phi node joining two control flow branches where each one has a different value of the same variable.

pass declares which passes have to be run before it itself can be executed. Based on this information the pass manager is then able to determine the passes' execution order.

Another criterion is the scope of a pass. The scope defines how much of the input program is processed at once. A module pass operates on the whole module and thus has the broadest scope. It is allowed to add and remove functions, manage arbitrary state and has no constraints on the order in which it processes functions. With such a broad scope a module pass has to be scheduled in sequence in relation to all other passes. In contrast to this a function pass can be scheduled in parallel to work on the same module as other function passes. This is made possible by narrowing its scope to that of a single function and by limiting the kinds of transformations it can apply. Thus a function pass is not allowed to add or remove functions and must not maintain state across different functions. It is also unable to decide the order in which it processes the functions in a particular module. This is decided by the pass manager. A basic block pass has an even narrower scope (constrained to a single basic block) which allows for even more flexibility when scheduling.

In addition to dependencies and scopes, scheduling is affected by another criterion. After a pass is finished processing a particular piece of code it tells the pass manager whether or not that piece has been modified. In case the code has been modified, the pass manager can then use the available dependencies information to work out which passes have to recompute their results.

### 2.1.3 Representation of Loops

This and the following subsection are base on a presentation [llv] given at the 2009 LLVM Developers' Meeting.

The LLVM IR does not have a direct representation of a loop. Instead, a loop is expressed by using a conditional branch instruction that branches back to a certain block as long as a certain condition is being met. This is not a strong definition, but rather the most common denominator for a piece of code to be considered a loop. Thus, a lot of different ways of expressing a loop arise. All of which would have to be considered when writing analyses and transformations dealing with loops. In order to fix this problem, LLVM offers a few mechanism that transform a loop into a canonical representation. Other parts of the compiler based on this representation only have to deal with this one loop form. Additionally this representation is optimized to make it easy to use for analyses and transformations. A loop is said to be in canonical form if it has the following properties:

The first property is a single entry point, which means that every control flow path going through the loop has to come from the same node in the CFG. This is accomplished by inserting a new block called the preheader in front of the loop's header. All blocks previously branching to the header are then changed to branch to the preheader. This makes the preheader the only block to be branching to the header.

The loop also has to have a single backedge, which is the only branch instruction with a target inside the loop. The backedge branches from the end of the loop's body to its

header. There are possibly two transformations applied to achieve a single backedge. The first one is called loop rotation and is applied to remove branches from inside the loop. This is done by moving the conditional branch to the end of the loop body and inserting an initial conditional branch just before the loop. The second one can be applied if the loop has more than one backedge. Each additional backedge is turned into its own inner loop, with the header copied from the original loop. The header contains another branch instruction, but its target is an exit block outside the loop.

Furthermore, an exit block of the loop must have exactly one other block branching to it from inside the loop, i.e. it has to have exactly one predecessor. This is achieved in a similar manner to the preheader insertion. A new block is inserted before the exit block who's only incoming edges in the CFG are from inside the loop. Its one outgoing edge is connected to the former exit block, which still has all other edges from outside the loop.

The last property describes canonicalized induction variables. This means the loop has only one induction variable starting at zero and stepping by one each iteration. All other induction variables are expressed as closed expressions taking the value of the induction variable as an argument.

### 2.1.4 Scalar Evolution

LLVM comes with an analysis pass called Scalar Evolution. Scalar stands for a LLVM value and evolution for the way this value is computed. This pass maps values it analyses to Scalar Evolution expressions (SCEVs) in its own expression language. The language includes simple arithmetic expressions, constants and add recurrences. It is defined recursively such that the operands of a SCEV are again SCEVs. The most important SCEV is the add recurrence which describes how a particular value inside a loop evolves in relation to the loop's iterations. A particular add recursion is made up of a SCEV representing the start value, an integer value representing the stride and the name of a block identifying the loop.

### 2.1.5 Polly

Polly[GGL12] is one of the projects under the general LLVM Compiler Infrastructure. It represents memory access patterns of a loop as a polyhedron, a mathematical structure. Polly can transform this structure and then generate code for OpenMP or GPUs from it. By transforming the structure, Polly can optimize and parallelize said code. One of the first steps in Polly's algorithm is finding parallelizable loops and annotating its results in the LLVM IR. A third party tool like ours can then make use of this information.

## 2.2 Autotuning

Optimizing the performance of an algorithm or a software program can be a difficult task. It is not always known or predictable how a change affects the performance of a system. It might, however, be possible to identify certain parameters that can be altered to modify in turn the way a system is executed. If such parameters exist, the optimization problem is reduced to finding a set of values for said parameters that maximize the performance. This new problem might still be impractical to solve manually. If we consider only a few parameters, each having a large value range, we might still end up with a very high number of possible combinations. In addition to this, such parameters often are not independent, further complicating the matter. The parper [WPD01] gives the problem of cache optimizations for particular hardware architectures as an example. Here, several factors like cache blocking, different caching strategies and sizes interlock in ways that are difficult to predict. This is where an empirical approach like autotuning can help find a

solution in a limited time. It uses a profiling mechanism to measure the quantity of the system we want to optimize for (e.g. program execution time). It also uses a search function which takes the above mentioned parameters as well as previously collected profiling info as input and a set of values (a configuration) as output. Each value in a configuration belongs to one of the parameters. The tuning process starts with an initial configuration of the tuning parameters. Next, the program is run with this configuration while the profiling mechanism collects measurements about the running program. Based on these measurements, the search function can now return a new configuration to be used in the next execution of the program. This is repeated until the search function finds an optimal configuration. Whether this optimum is a global or local one depends on the concrete formulation of the search function.

### 2.2.1 Offline and Online Tuning

Autotuning algorithms can be classified into two groups, offline and online tuning. An algorithm in the first group finds the optimal configuration by doing a certain amount of test runs. The program is then deployed together with said configuration and runs without the configuration changing. A tuning algorithm is said to be an online tuner, if it profiles the program after it has been deployed. This requires the tuning system to be deployed alongside the program to optimize. It collects data and changes the configuration while the program is running in order to optimize constantly.

Offline tuning has no runtime overhead, as the optimal configuration can be compiled into the final executable. In comparison, with online tuning, the calls to the tuning system, the profiling and the execution of the search function have to happen at runtime. On the other hand, while the offline approach is completely oblivious to changes in the programs environment after deployment, the online tuner can react to such changes dynamically. The input data might change in content or size over the lifetime of the program, or the overall system load might impact the available CPU time. As an example for the latter, Perpetuum[KP11] is an online autotuner that optimizes multiple applications running on the same system, at the same time. Additionally, the system that is to be used to run the program might not be available for testing. In this case the offline tuner would have to run on an approximated system leading to suboptimal results.

## 2.3 CUDA

Compute Unified Device Architecture (CUDA) is a technology developed by NVidia which was first deployed together with their Tesla mircoarchitecture. CUDA is the combination of a parallel computing platform and an API sitting on top of that. Through this API, the GPU can be used for general purpose computing, also known as GPGPU. GPUs have traditionally been used only for graphical computations. However, with the introduction of GPGPU these processors are applied to solve problems that are usually solved by CPUs. These problems include computations in fields like machine learning, medical imaging, cryptography and data mining.[CBM+08]

The remainder of this section introduces a virtual machine specifically for NVidia GPUs in section 2.3.1 and the thread hierarchy this virtual machine provides in 2.3.2.

### 2.3.1 PTX

PTX is a virtual machine for parallel thread execution accompanied by its own instruction set architecture (ISA). The PTX-ISA is specially designed for NVidia GPUs with capabilities specified by the Tesla and later microarchitectures. The ISA provides a stable programming interface for multiple generations of GPUs. Programs to be run on the

GPU can be formulated in higher level languages like C or C++ to later be translated to PTX. Before the translated code can be executed, it will be compiled a second time, now targeting specific hardware instruction set in question. This design makes it easier for developers of higher level compilers as they only have to generate code in the ISA. They do not have to consider the details of multiple different hardware instruction sets. Similarly, transformations and optimizations offered by third parties can target or operate on the ISA and can thus be plugged into the pipeline.

### 2.3.2 Thread Hierarchy in PTX

A PTX program represents one thread executing a specific task on a single data point. The concrete data point is variable and is defined by certain parameters of the program. These parameters describe the thread's position in relation to the hierarchy it is embedded in. The smallest conceptual compound structure in this hierarchy is a cooperative thread array (CTA), also known as a block. Threads in a CTA execute the program concurrently, but are able to communicate with the help of synchronization. Each thread in a block is assigned an index called the *tid*, which is one of the parameters mentioned above. Another parameter is the number of threads in each CTA, specified by *ntid*. As a single CTA can only contain a limited number of threads, multiple CTAs can be joined together to form a grid. Thus allowing the execution of a kernel on millions of threads. This, however, comes at the cost of communication between threads in different CTAs. The grid concept introduces two further parameters called *ctaid* and *nctaid*. These define the index of a CTA in a grid and the total number of CTAs in a grid respectively.

# 3. Related Work

This section outlines other works that have solved the problems stated in section 1.1 in part or in different ways. Section 3.1 looks at approaches to managing memory transfer and mapping of parallel code to the GPU architecture. In section 3.2, we describe different approaches to distributing the given input over available processing units. Finally, in section 3.3, we describe solutions to the problem of tuning the program in question at runtime.

## 3.1 GPU Code Generation

*KernelGen* [MLZB14] ports sequential CPU code to run on a CUDA enabled GPU. It produces two types of kernels (main and computational loops) each with its own purpose. The main kernel is executed in single thread and is responsible for coordinating the execution of computational loops kernels. It also offloads to the CPU unportable functions and code that is not worth being executed on the GPU—code where the overhead of GPU execution outweighs its performance improvements. A computational loops kernel contains computationally intensive loops that are parallelized to take advantage of the massively parallel GPU. *KernelGen* is implemented by extending Polly's OpenMP back-end to generate PTX code. They take advantage of up to 3 grid dimensions available on the GPU, by mapping different loops in a loop nest to different grid dimensions. This also allows them to coalesce memory transactions of threads residing in the same warp. Our tool does not take advantage of loop nests in such a way, but rather maps the iterations of the outer most loop to one grid dimension only. However, *KernelGen* lacks the ability to utilize both—the GPU and the CPU—concurrently always idles with one device, while the other is executing.

*PPCG* [VCJC+13] is a source-to-source compiler offloading data-parallel computations to CUDA enabled GPUs. It uses the polyhedral model to describe loop nests. This results in a schedule describing the execution order of contained statements. They then apply affine transformations to this schedule in order to expose parallelism and tiling opportunities. To tile a parallel loop, it is split into a tile loop, iterating over the tiles, and a point loop, which iterates inside the tiles. The tile loops are then mapped to blocks in a grid and the point loops to the threads in a block. The inner most of the parallel loops is mapped to the x dimension, to take advantage of coalescing opportunities. Memory accessed by a parallel loop has to be transferred to and from the GPU. *PPCG* accomplishes this by copying the entire content of each accessed array before starting the kernels. After the

kernels have finished, only the modified arrays are copied back to the CPU. Our tool only copies a certain part of an array, defined by the iteration bounds of the loop in question. It does, however, copy all the parts back to the CPU—not just the written ones. *PPCG* makes further efforts to allocate parts of the arrays to registers and shared memory, thus optimizing memory access times. We only allocate to global memory.

## 3.2 Heterogeneous Code Execution

*libHawaii* [RDP14] is a framework for writing and optimizing streaming applications for heterogeneous systems. These are applications that apply a set of filters to a stream of successive work items. A work item holds input data and represents this data as it flows through the system. A filter repeatedly receives a work item, processes it in some way and then pushes it further through the system. The programmer using this framework expresses the processing steps in their application as filters. These filters must then be combined into a possibly recursive structure, called a flow. A flow can employ different heterogeneous computing strategies for different types of parallelism: Data flow parallelism can be exploited via pipelining, data parallelism via partitioning and task parallelism via demand-base allocation. The framework dynamically measures the latency and throughput of every single filter. Based on the computing strategy chosen, a special heuristic then uses this data to compute the optimal mapping of filters to processors. The mapping aims for a filter to keep pace with its input while being as energy-efficient as possible. By applying these heuristics at every level of the recursive structure, a very fine-grained adaption can be achieved. *libHawaii* optionally provides specialized support for CUDA enabled GPUs by overlapping the transfer of data with the computation of results: Data needed for the next and results from the previous step can be transferred while the current step is still executing. Summarizing, *libHawaii* offers multiple different strategies to structure an application in such a way that it can be executed and dynamically optimized on heterogeneous system. It is, however, limited to a special type of application, which forces a programmer to adapt or completely rethink an existing solution, if they want to apply this framework for heterogeneous execution.

*Qilin* [LHK09] provides an automatic mapping of computations to CPU and GPU. This mapping is based on an initial training run and can be adapted to changing input problem sizes. The first run of a certain program is used to create a projection function predicting the program's runtime. This function is parameterized by the input size and produces estimates for CPU and GPU runtimes. On consequtive runs of a program, *Qilin* distributes input over CPU and GPU devices in such a way, that the maximum of both predicted runtimes is as small as possible. Basing the prediction function on only a single test run might lead to inaccuracies, when subsequent input sizes vary greatly. It also removes the ability to react to other changes affecting execution time. An example would be a significant change in load on one device compared to the other one. Online autotuning, on the other hand, is able to constantly tune the application without having to rely on training runs. It does this at runtime and is thus able to dynamically react to these changes. To take advantage of *Qilin*, programmers have to manually write code against a specific API. Not only structures linked to concurrency directly, but also datastructures and operations have to be expressed in a special API. This forces the programmer to heavily refactor existing code bases and learn a new API, even if they start from scratch. Our tool, on the other hand, automatically transforms the sequential program, saving time and resources.

## 3.3 Autotuning

*Sambamba* [SHZH13] uses a program dependence graph (PDG) as its internal structure. Each node in this graph is an instruction, a conditional branch, or a group of nodes. The

edges represent either control flow dependencies or data dependencies. The children of a group node are all reachable via the same control flow path and might exhibit data dependencies amongst themselves. An integer linear program (ILP) solver then computes an initial fork/join schedule in each group node with the goal of minimizing critical path execution time. The inputs of the ILP are data dependencies between the children, estimated execution times, branch profiles and parallelization overhead. Based on this schedule, the runtime component then profiles parallelizable function calls and combines the most promising ones into a parallel section (ParSec). This produces the ParCFG, a CFG annotated with fork and join points marking the ParSecs. The code produced from the ParCFG is further monitored for a significant change in execution time to see whether the ParCFG has to be reevaluated. This allows *Sambamba* to dynamically react to changes in its environment that impact performance. For each parallelized function, it also keeps the original sequential version. Thus it can dynamically decide which version to execute depending on different dispatch strategies. This makes it possible to limit the number of parallel tasks to prevent performance decreases by interference. *Sambamba* exclusively works on the CPU and does not make use of additional available accelerators, like a GPU.

*BAAR* [DP14] accelerates binary programs by offloading certain function calls to the Xeon Phi. The binary is first translated to LLVM IR and then executed in a just-in-time compilation environment to enable runtime analysis and transformations. To decide when to offload a certain call, it statically calculates a score for each function based on the number of integer and floating point operations. It then dynamically computes the combined sizes of the arguments passed to the function call. If the ratio of argument size over the function score is smaller than a user-defined constant, the call is executed on the accelerator. Before a function can run on the accelerator, however, it has to be parallelized, vectorized and compiled. All of these steps result in a performance hit, especially the compilation step: The IR code is translated to C code, then compiled on the CPU and finally sent to the Xeon Phi for execution. Additionally, *BAAR* either offloads the complete function or none of it - there is no nuance to utilize both devices (CPU and accelerator) simultaneously.

# 4. Concept

In this chapter we describe the concept of our tool. We begin with stating the goals we want to achieve by implementing our tool in section 4.1. In the same section, we give a short overview of the overall concept which is then explained in further detail in the following sections:

The remainder of this chapter describes key features of our tool in greater detail: Section 4.2 explains how data dependencies relate to parallelism and what loop carried dependencies are. After this, we introduce the concept of loop partitioning in section 4.3. In section 4.4, we show how we map the iterations of a loop to the cores of the GPU and how we find the memory regions the loop operates on. Finally, in section 4.5 we demonstrate how the autotuner is integrated into the program.

## 4.1 Goals

We aim to increase the performance of sequential programs by automatically transforming them to run on heterogeneous systems. The work to be done is to be optimally distributed over available processing units to maximize performance. These distributions are done at runtime, thus allowing a program to be transformed once and then be optimized for the concrete system it is executed on. In order to make the tool as universally applicable as possible, we want to do these things in an automated fashion: Firstly, a programmer is not required to know about parallel programming in general or GPU programming in particular to use our tool. Secondly, no annotations or changes to a sequential program are necessary in order to transform it. Thirdly, no programmer input is needed for the tool to make optimization decisions.

To achieve these goals, we implement a completely automatic solution whose input is a sequential program and whose output is an online tuned, parallel program. This paragraph gives an overview of this solution, which is also depicted in figure 4.1. Our tool operates on code in LLVM IR form and is thus able to process programs written in any language for which an LLVM front-end is available. We begin with analyzing the program's loops in search of areas that are parallelizable. To make analysis and later transformation easier we transform these loops into canonical form. A loop will then be marked as parallelizable, if it does not contain any loop carried dependencies. Next, we divide the loop into two loops (called partitions) and introduce a parameter `P` to vary the size of their iteration ranges. One partition will later execute on the CPU, the other one on the GPU. With

both partitions in place, we analyze which memory is accessed by the GPU partition. This information helps us to transfer the needed memory between CPU and GPU memory. We also translate the GPU partition into a kernel that can be run on a GPU. Then, the original loop is replaced by the kernel launch and the execution of the CPU partition. Additionally, measurement calls are inserted to obtain information used by the autotuner. This information is then used to adjust the parameter `P` to dynamically minimize the execution time.

## 4.2 Detection of Parallelism

The functionality described in this section is already available in `AutoCU`. Our tool starts by detecting parallelism in the input program.

It is not generally possible to reorder the instructions of a sequential program without altering their meaning. As an example, consider the instructions in figure 4.2: If `A` is executed before `B`, the variable `x` holds the value `13`. If, however, the execution order is changed and `B` is run before `A`, the value of `x` is `16` in the end. Thus, when parallelizing a program, we have to make sure to parallelize only those instructions that can be reordered without altering the result of the program.

```
x = 5
x = x * 2  // (A)
x = x + 3  // (B)
```

Figure 4.2: Two instructions that cannot be reordered without changing the meaning of the program.

Parallelizing a computation leads to an inherent overhead. In our case, the tool has to copy data to and from the GPU and has to properly start and stop kernels running on the GPU. Thus, in order to improve the overall performance of a program, the performance increases we obtain by running computations in parallel must outweigh the overhead they introduce. For this reason, we focus on parallelizing loops as these are usually the computationally intensive parts of a program. Another reason for selecting loops, is that they often exihibt a high degree of data parallelism, which allows for the efficient use of SIMD (single instruction, multiple data) processors.

We classify a loop as parallelizable, if it is in canonical form and does not contain any loop-carried dependencies (LCDs). A loop is said to be canonical, if it has a single entry point and a single backedge, if each exit block has only one predecessor and if its induction variables are canonicalized. The canonical form itself and techniques used to transform non-canonical loops are described in more detail in subsection 2.1.3. The advantage of canonical loops is that we do not have to deal with many different loop forms when implementing the later steps of our tool. This way, we have to implement the partitioning and GPU translation against just one loop form. Additionally, the transformations needed to canonicalize loops are already available in the LLVM Core. A loop contains a loop-carried dependency, if the results of one iteration of the loop depend on the results of a previous iteration. This would lead to problems, as executing the iterations in parallel could change their order and thus would potentially result in incorrect programs (see the example in figure 4.2). An example of an LCD is depicted in figure 4.3: The given loop iterates over an array `a` and assigns each element the value of the previous element plus `10`. Thus, `a[1]` depends on `a[0]`, `a[2]` depends on `a[1]` and so forth.

```
for (i from 1 to N)
  a[i] = a[i-1] + 10
```

Figure 4.3: A loop with an LCD: One iteration depends on the results of the previous iteration.

## 4.3 Partitioning

We call the process of dividing a loop into two or more loops partitioning. In our case, we only produce two partitions, as we want to run one on the CPU and one on the GPU. It is, however, possible to extend our tool to create more partitions that are then run on other processing units. A partition is essentially a clone of the original loop with its bounds altered to restrict iteration to a certain range. Our tool does this by introducing a variable `P` called the partitioning parameter. `P` is the upper bound of the first partition and the lower bound of the second partition. Figure 4.4 shows a loop (`A`) and the two partitions (`B` and `C`) produced from it by the partitioning process. The parameter can then be used to control how many iterations are executed on each device.

```
(A)                    (B)                    (C)
for (i from 0 to N)    for (i from 0 to P)    for (i from P to N)
  c[i] = a[i] + b[i]     c[i] = a[i] + b[i]     c[i] = a[i] + b[i]
```

Figure 4.4: A loop (`A`) and the corresponding two partitions (`B` and `C`).

## 4.4 GPU Transformation

The main goal we want to achieve with our tool is to increase performance through parallelizing loops. When transforming loops to GPU code, we have to think about how we map the loop's iterations to the GPU threads. This is explained in greater detail in section 4.4.1. In addition to that, we need to find a way to determine and transfer data accessed by the loop. We describe our solution to this in section 4.4.2.

### 4.4.1 Mapping Iterations to Threads

The functionality described in this section is already available in `AutoCU`.

By partially executing the loops on a GPU, we want to make use of their highly parallel nature, having thousands of cores available. In order to be able to do this, we have to translate the sequential loop in such a way that makes it feasable for it to be executed on many cores. A GPU usually operates in SIMD mode, which means that a single instruction is executed by multiple processing cores all operating on a different data point. As has been outlined in 4.2, we only parallelize loops without LCDs. Thus, we only have to deal with iterations that are independent of one another. This allows us to map each of the loop's iterations to its own thread on the GPU. We usually have a far greater number of iterations than number of available GPU cores, which necessitates the division of iterations into smaller groups. Each of these groups contains as many iterations as there are available threads. The last group may be smaller if the division has a remainder, leading to a portion of threads being idle. One thread sequentially executes one iteration out of every group, but does this in parallel to all other threads on the GPU. This reduces the execution time of the whole iterations to the time it takes to execute one iteration times the number of groups. If we take the number of iterations to be $N$, the number of available threads to

be $C$ and the time it takes to execute one iteration to be $t_0$, we get an overall execution time of $T$.

$$T = (\frac{N}{C} + 1) \cdot t_0. \tag{4.1}$$

### 4.4.2 Memory Management

The mechanisms explained in this section are taken from [BÏ5].

A loop usually depends on information outside of the loop itself, in order to compute something meaningful. If we want the loop to function correctly when running on the GPU, we have to make this information available there. This information is present in the form of local variables on the stack and as regions of memory on the heap. We do not have to worry about the values on the stack, as we can pass them as arguments to the kernel call and the runtime system then makes them available on the GPU. The regions of memory, on the other hand, have to be explicitly copied to the GPU memory. But before we are able to either pass arguments or copy memory, we have to determine which stack variables and memory regions are accessed by the loop. We find all relevant variables by looking at each instruction in the loop's body and collecting the variables referenced there. Each of these variables that is not declared inside the body itself has to be passed to the kernel call.

The process of determining the memory regions we need to copy is a bit more involved. To makes things simpler, we restricted the problem by demanding the following points:

1. The induction variable of a loop is counting up.
2. All memory accesses must be expressable as an function of the general form $a \times x + b$.
3. The amount of memory copied is a conservative guess of the actual memory that is accessed by the loop, i.e. not all of the memory copied is accessed by the loop, but all accessed memory is copied.

The first restriction makes it straight forward to determine the lower and upper bounds of the loop's range. The second one allows us to determine the address range of a certain pointer inside the loop by using Scalar Evolution analysis (see section 2.1.4 for further details). The third one protects us from identifying every memory address accessed in every iteration of the loop. Instead of finding each exact address, we compute only the minimum and maximum address pointed to by a given pointer. This makes it faster to compute memory regions, but might also copy a lot of memory that is not accessed. For example, on the one hand, a loop with a big stride relative to its total range accesses only a small part of the memory we would copy. On the other hand, a loop with the smallest possible stride (equal to the size of one element in memory) will access all copied elements.

To determine the regions of memory accessed by a particular loop, we proceed as follows. At first, we express the pointer associated with the memory access as a Scalar Evolution expression (SCEV). Thus, gaining insight into its evolution in relation to the loop's iterations. This SCEV is a function that given a value of the induction variable, produces the address the pointer points to in that specific iteration. Evaluating the function at the lower and upper bound of the induction variable's range, gives us the minimum and maximum address the pointer points to. The intervals produced by this procedure are generally not disjoint. Thus, when two or more intervals overlap it is necessary to merge them preventing redundant copying. The size of memory that has to be copied changes dynamically as the size of the GPU partition changes. Allocation of memory is a costly operation, which is why we only allocate memory once for each block of memory. We do this by allocating the largest partition possible, which is large enough to hold the data

needed by the unpartitioned loop. When the partition changes, we only copy the memory needed by that partition.

As an example, we demonstrate this process when applied to the loop in figure 4.5. Said loop scales a vector `v` by a factor `k`, adds the vector `w` and captures the result in a vector `c`. These operations are only applied to the vector elements, that fall into the range between a lower bound `lb` and an upper bound `ub`. The vectors are expressed as pointers to specific memory regions, allocated to have a size of 100 elements. Their initialisation is not shown, as the exact content is not relevant here. Starting the analysis, we first examine the loop's body to find any references to local variables. In this case, there is only one, the factor `k`. In the second part of the memory analysis, we take a look at every memory access inside the loop and express them as SCEVs. As the SCEVs in this example only differ in the pointer they reference, we focus on only one memory access, namely `v[i]`. On a system where `float` and `size_t` have a size of four bytes, the SCEV looks like this: $\{(4 \cdot lb + v), +, 4\}$. Next, we evaluate this expression at the first (`i` = `lb`) and last (`i` = `ub`) iteration, to determine the lower and upper bound of the sought memory region. This gives us $4 \cdot lb + v$ and $4 \cdot ub + v$ respectively. In this case, the memory intervals do not have to be merged as they are disjoint.

```
size_t lb, ub;
float *v, *w, *c;
v = malloc(sizeof(float) * 100);
w = malloc(sizeof(float) * 100);
c = malloc(sizeof(float) * 100);
float k = 5;
for (size_t i = lb; i < ub; ++i) {
  c[i] = k * v[i] + w[i];
}
```

Figure 4.5: A vector `v`, expressed as an array, is scaled by a constant factor `k` and added to a vector `w`. The result is stored in vector `c`.

## 4.5 Tuning

With partitioning a loop and introducing the partitioning parameter `P`, we gained the possibility to adjust the ratio between the number of iterations running on the CPU and those running on the GPU. We want to use an autotuner to find the ratio that achieves the highest occupancy rate of both devices and thus results in the highest performance increase. Using an offline autotuner for this, would not allow us to manipulate `P` at runtime, which is why we choose to use the online tuning approach instead. This gives us the possibility to react to changing inputs or other environmental factors that might have an effect on the execution of the program. The advantages of online tuning are explained in further detail in section 2.2.1.

*AtuneRT* is an online autotuner implemented in a client-server fashion. Our tool is the client in this relationship and uses the provided client-API to communicate with the actual autotuner. *AtuneRT* needs to know about the tuning parameters it tweaks and about the sections it measures in order to do its work. In our case, each loop is associated with a single parameter `P` that is introduced during partitioning and that controls the distribution of iterations over CPU and GPU. Additionally, we introduce a single section belonging to said parameter which includes the following actions:

- Copying neccessary data to the GPU's memory.

- Initiating the asynchronous execution of the first partition on the GPU.
- Executing the second partition on the CPU.
- Copying data from the GPU back to the CPU's memory.

It is important to include the memory transfer before and after kernel execution in order to obtain representative measurements. If we left them out, we would ignore a substantial part of the overhead inherent in executing on the GPU, which would in turn incorrectly favour the GPU for smaller inputs.

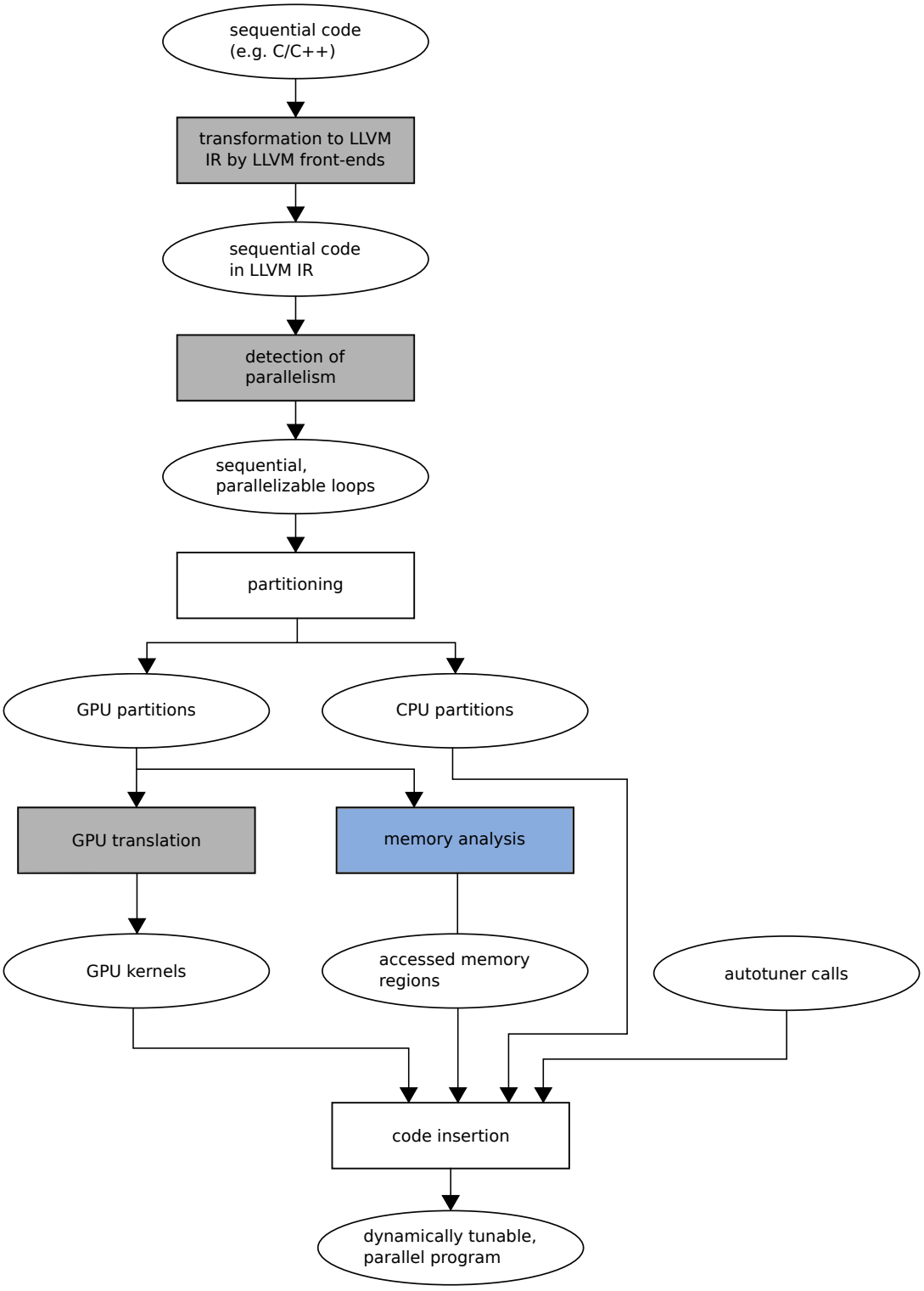


Figure 4.1: A flow chart depicting the steps (rectangles) and their input and output (ellipses) used in our solution. The grey steps were already present in `AutoCU` and the blue one was adapted from [Bï5].

# 5. Implementation

We implement our solution by extending the existing tool `AutoCU`, which offers most of the functionality we need. In the following, we first give an overview of `AutoCU` and then explain how we integrated the partitioning functionalities.

`AutoCU` makes extensive use of the pass pipeline and thus models its analyses and transformations as passes. The first step is the detection of parallelizable loops, which is implemented in the class `KernelElicitationPass`. It detects parallel loops by examining their PDG. The PDG is created by a class called `ProgramDependenceGraphPass` which depends on the class `SideEffectAnalysis`. This analysis determines which arguments and global variables a function reads from or writes to. The PDG pass depends on further analyses, which are, however, already provided by LLVM. The kernel elicitation pass works on the basis of functions: If it finds a function containing a parallelizable loop, it collects the function's references to other functions and globals. Afterwards it determines the loop's induction variables and it's lower and upper bounds. Additionally, it records certain transformations that have to be applied to the function once it is extracted into a kernel. The second step is the actual kernel creation which is done by the class `PTXTargetGenerator` that copies the function in question to the target module. It also creates a copy for the host module. The invocations of the original function are replaced by calls to a so called dispatch function. This dispatch function calls out to the autotuner, to dynamically decide which version—either host, or device function—to execute. The third step is the memory management, which is completely implemented as a runtime component. A pass named `MemoryManagementTaggerPass` inserts tracing calls after memory managing function calls like `malloc`. The tracing calls record the allocated blocks and their sizes which is then used before and after the kernel call to know which memory regions have to be transferred to and from the GPU.

A naive approach would be to implement the partitioning functionality as another transformation that is run on the function candidates before all the other transformations. The problem with this is, that the partitioning changes certain loops and thus invalidates parts of the information collected by the `KernelElicitationPass`. This then neccessitates rerunning the parts of the analysis, which meant a great amount of changes in the general design of `AutoCU`. Instead of doing this, we prepended the whole `AutoCU` pipeline with an additional pass, called `PartitioningPass`, that devides loops into partitions. For each loop, one partition is then passed on to be turned into a kernel. We still had to make changes to `AutoCU`, but these did not severly affect its overall design. We moved the insertion of tuner calls into the partitioning pass, because we want to measure the

execution time of both of our partitions, not just that of the kernel. Next, we replaced the dynamic memory management with a mostly static one, that relies on Scalar Evolution analysis to compute accessed memory. The new memory management has been adopted from [Bï15]. Due to the way the Scalar Evolution library is implemented, it was easiest to put the management into the partitioning pass. Finally, we moved the parts of the detection of parallelism out of the pass pipeline and into a separate class, allowing for it to be easily used by several passes. This allowed us to utilize the detection mechanisms in the partitioning pass to find partitionable functions in the first place. To summarize, detection of eligible loops, partitioning of those loops, analysing memory and generating memory managing and autotuning calls is done by the `PartitioningPass`. The original `AutoCU` pipeline is now in charge of transforming chosen functions into kernels.

The rest of this chapter is structured as follows: We start in section 5.1 by outlining other command line tools used before and after the invocation of our tool to create the final executable. Section 5.2 then describes how the parallelism detection in `AutoCU` (and thus also in our tool) works. Section 5.3 explains the details of creating partitions and inserting tuner calls, while section 5.4 outlines how the transformation of a certain partition to GPU code is done.

## 5.1 Toolchain Overview

The whole process operates on a sequential program as input, that is compiled with `clang` to LLVM IR. Then, several LLVM transformations are applied to that code, by running the `opt` tool with `-mem2reg`, `-instcombine`, `-loops`, and `-sroa` as flags. This code is then passed to `AutoCU` as input, which produces two LLVM IR modules: The host module and the target module. The former contains the original program with partitioned loops, calls to kernels and calls to the auto tuner. The latter holds the kernels that run on the GPU. These modules are compiled to assembly code by `llc`. Then they are linked with the runtime functions of `AutoCU` and the client code for the autotuner to produce native code.

## 5.2 Detection of Parallelism

This section describes the parallelism detection mechanisms used in `AutoCU` and thus, by extension, also in our own tool. In subsection 5.2.1 we explain how the PDG is contstructed and which information `AutoCU` uses to do so. Subsection 5.2.2 then shows how `AutoCU` selects suitable loops based on their form and their loop carried dependencies.

### 5.2.1 Construction of the PDG

The `ProgramDependenceGraphPass` makes use of multiple analyses whose results have to be available before it can run. These four analyses needed are listed below:

- `AliasAnalysis`
- `PostDominatorTree`
- `DependenceAnalysis`
- `SideEffectAnalysis`

The LLVM project already provides us with the first three of the four needed analyses, leaving the `SideEffectAnalysis` (SEA), which was already implemented in `AutoCU`.

The following paragraph describes how SEA processes a given function. This analysis finds out which global variables and which of its arguments the function accesses. It finds these

```
for each directed edge(x,y) in CFG:
    if y is not post dominated by x:
        ydom <- y in the post dominator tree
        idom <- immediate post dominator of x
        while (ydom != idom):
            create control edge (ydom,x)
            ydom <- immediate post dominator of ydom
```

Figure 5.1: Pseudo code for the control dependence algorithm presented in [FOW87].

values, by examining the load, store and call instructions of the function: If it encounters a load instruction, it recursively traverses its operands to eventually find out whether they depend on an argument or a global variable. This can be done by utilizing the def-use chain exposed by the `llvm::Value` API. Such an argument or global variable is then added to the read set of the function in question. Store instructions are processed analogously, with the difference of the values being collected in a write set instead. If a call instruction calling a function $G$ is encountered, the global values of $G$'s read and write sets are copied to $F$'s respective sets. Additionally, the arguments passed to the function call are analyzed in the same way operands of load and store instructions are analyzed.

To build a PDG, `AutoCU` needs to collect data and control dependencies existing between the instructions of a particular function. Data dependencies are divided into use, flow, anti, in and out dependencies. A dependency between two instructions is modeled as an edge in the PDG labeled with the type of the dependency. We create a use edge from an instruction $I$ to an instruction $I'$, if $I$ operates on the value produced by $I'$, according to the def-use chain. Edges of other types are created by calling `DependenceAnalysis.depends` for all pairs in the cross product of the function's basic blocks and examining the result accordingly. If one of the instructions in such a pair is a function ($F$) call, this pair has to be handled specially: First, we use `AliasAnalysis.isNoAlias` to determine, which of the arguments passed to $F$ might alias to which of the instruction's operands. Then, we use the SEA to collect the arguments that are either read or written when the instruction is written, or that are written when the instruction is read. These collected arguments together with the instruction are attached as so called contraints to the edge.

The control dependencies are found with the help of the algorithm presented in [FOW87]. Figure 5.1 shows a pseudocode version that finds the control dependencies between nodes in a control flow graph. In the LLVM IR, each basic block is a node in a CFG. Thus, the above algorithm produces dependencies between two basic blocks, which does not quite fit our purposes. For this reason, if we find a basic block $B'$ to be control dependent on a basic block $B$, we record this in the PDG as multiple edges, where the instructions in $B'$ are control dependent on the terminator instruction in $B$.

Taking all the created edges representing data and control dependencies then forms the program dependence graph of a particular function. This graph can then be used in the next step, to find loop carried dependencies.

### 5.2.2 Analyzing Loops

We outlined the criteria we apply to detect parallel loops in 4.2. The loop has to be in canonical form and must not exhibit any loop-carried dependencies. The first property is checked by verifying if the loop has a preheader, a single backedge and a single exit block. The second one is checked by looking at the strongly connected components (SCCs) of a graph. A graph is strongly connected if every vertex is reachable from every other

vertex. The SCCs of a graph are its maximally strongly connected subgraphs. Maximally strongly connected means, that no further vertices or edges can be added to the component without it loosing that property. The PDG as described in section 5.2.1 is expressed using the BOOST [boo] graph API, which allows us to use the `boost::strong_components` function to find all SCCs in our PDG. To determine if a particular SCC describes a loop carried dependency, we check if any of its nodes (instructions) is a phi instruction. If this phi can be expressed as an add recurrence and has a step width of one, it is an induction variable and the corresponding loop will be parallelized. Otherwise, it is a loop carried dependency and the loop is not touched.

## 5.3 Partitioning of Loops

Partitioning is the act of dividing the iterations of a loop into two parts in such a way, that the first part is executed by one loop and the second one by another loop. Both new loops have identical bodies and differ only in their iteration range. For a start, we focused on functions containing only one loop, because dealing with parallelizing multiple loops in the same function increases the complexity. We explain the partitioning by considering the example of a function $F$ containing a single loop $L$. We start by determining the lower and upper bound of the iteration range and by finding the variables the loop depends on. The lower bound is equal to the value the loop's phi instruction evaluates to, if the control flow comes from the loop's preheader (this is akin to the control flow entering the loop for the first time). The upper bound is equal to the number of times the backedge is taken, which is computed by the Scalar Evolution analysis. Collecting the variables is done by iterating over the operands of each instruction in the loop's body and retaining those, that are declared outside the loop.

With this information in place, we copy $L$ into a new partition function $F_h$. The function is parameterized by a lower and an upper bound, defining the new iteration range of $L$. Further parameters are introduced for every variable used inside, but defined outside of $L$. In figure 5.2 the two parameters `lower` and `upper` are added to $F_h$ and used as loop bounds.

```
void F(int *a, int *b, int *c,
          size_t N) {
  // L
  for (size_t i = 0; i < N; i++)
    c[i] = a[i] + b[i];
}
```

```
void F_h/d(size_t lower, size_t upper,
           int *a, int *b, int *c) {
  // L_h/d
  for (size_t i = lower; i < upper; i++)
    c[i] = a[i] + b[i];
}
```

```
void F(int *a, int *b, int *c, size_t N) {
  size_t P = get_partition_parameter("F", 0, N, 1);
  start_measurements("F");   // (S1)
  F_d(0, P, a, b, c);        // (S2)
  F_h(P, N, a, b, c);
  sync_with_gpu();           // (S3)
  copy_memory_to_host();     // (S4)
  stop_measurements("F");    // (S5)
}
```

Figure 5.2: A loop is extracted into a partition function and replaced by measurements, bounds calculations and calls to said partition function.

Parameters `a`, `b` and `c` are also added, as they are accessed inside the $L_h$. The original loop $L$ in $F$ is then replaced by calls to the autotuner API and calls to the $F_h$ function. The first call to the autotuner returns the next value for the partitioning parameter $P$, which can then be used to determine the bounds for the partitions. The partition function is called once for each partition, with bounds and referenced variables passed as arguments. Due to the fact, that the GPU tranformation modifies the function it transforms, we have to make a copy $F_d$ of $F_h$ which is later transformed. One of the calls to $F_h$ is then replaced by a call to $F_d$ (the loop in $F_d$ is called $L_d$). This is statement *S2* in figure 5.2. From the point when $F_d$ has copied memory and launched the kernel and $F_h$ has begun executing, both loops $L_h$ and $L_d$ run concurrently. This necessitates a synchronization call to the GPU after $F_h$ has finished, so that the GPU memory can be copied back to the host. Both of these are runtime calls that are inserted after the call to $F_h$ (statements *S3* and *S4* in figure 5.2). Finally, function calls to start and stop measurements for the autotuner are inserted before the call to $F_h$ and after the call to download the memory respectively (statements *S1* and *S5* in figure 5.2).

## 5.4 Memory Management

The memory management solution described in this section was adopted from [BÏ5].

As described in section 4.4.2, we need to determine the regions of memory accessed by the loop in question and transfer those to and from the GPU. The implementation of this algorithm has been taken from citelukas and has been adapted to our code base. It consists of two parts, a static and a dynamic one. The former is based on the Scalar Evolution analysis and compiles a list of pointer intervals and inserts calls from dynamic part into the IR. The latter deduplicates overlapping intervals, allocates and copies memory and maps between host pointers and device pointers.

The memory management is part of the `PartitioningPass`, as we have to analyze the original loop $L$ and the partitioned one $L_d$. It made sense to integrate these two, as it is not possible to analyze the same function twice (in our case containing a modified loop the second time), without loosing the results from the first run. For this reason, firstly we run the static and the dynamic part on the original loop. Then we change its bounds to reflect the new iteration range. Secondly the process is repeated on the partitioned loop.

The static part starts by analysing $L$ and creating a SCEV expression for every memory access in $L$. These SCEVs are add recursions which can be evaluated at a particular iteration to get the address the corresponding pointer points to. We evaluate each SCEV at $L$'s lower and upper bound respectively, to get an address interval containing all possible memory accesses of the corresponding pointer. During evaluation, we distinguish between induction variables that count upwards and those that count downwards—an add recursion is said to be positive or negative respectively. A positive one is evaluated at the lower bound to obtain the beginning of the address interval and at the upper bound to obtain the end. This is reversed for the negative one. The next step is to expand the evaluated SCEV expressions, which translates them into IR code. We then generate code that collects these address intervals in an array to be passed to the memory deduplication runtime call. As mentioned in the previous paragraph, this process is then repeated for the partition loop $L_d$. The results from both deduplication calls are then passed to a runtime call that allocates memory on the GPU and transfers the neccessary data. The generated code is inserted into the entry block of the device function $F_d$ (the function to be transformed by the GPU transformation part).

The first function of the dynamic part of memory management `deduplicateMemory` takes the previously mentioned intervals and combines the overlapping ones into so called memory blocks. A block begins at the smallest beginning address of its contained intervals and

ends at the largest ending address. As mentioned in 4.4.2, we allocate an amount of GPU memory that can hold all data needed by the unpartitioned loop $L$, but we only actually copy the memory needed by the current partition. This is done to prevent reallocation of memory each time the size of the GPU partition changes. For this reason, a second function `allocateAndCopyMemory` takes both sets of blocks–those needed by $L$ and those needed by $L_d$. The first set is used to determine the size of memory allocated on the GPU. It does this only once, the first time it is invoked and then stores the returned device pointer for later invocations. The second runtime function then copies each partition block to the allocated device memory and remembers which partition block was copied to which device pointer. We also store a mapping from each partition interval (the intervals that are passed to the deduplication) to its corresponding device pointer. This information is later used by the runtime function `getDevicePtr`, which maps a given host address to its corresponding device address. Memory copied to the GPU before the kernel executes has to be copied back to the host CPU, after the kernel has finished. This is accomplished by the `downloadDeviceMemory` function, which—given the name of a kernel—copies all memory used by that kernel back to the host.

## 5.5 Mapping Iterations to GPU Threads

The mapping of loop iterations to GPU threads is already implemented in `AutoCU`. The transformation of the device partition $F_d$ is implemented in the class `PTXTargetGenerator`, which is integrated into the pass pipeline via an immutable wrapper pass `PTXTargetWrapper`. Each time a loop has been divided and the partitions have been extracted into a function, one of them is selected and stored in a list. This list is then used by the generator to limit the functions it transforms. The generator has two responsibilities: It adapts the loops to be run in a parallel fashion and then replaces them with the kernel invocation.

As the kernels are compiled by a different LLVM back-end as the host code is, we create a new module (called the target module) to hold all the kernel functions. After copying such a function to the target module, its backedge is remove and replaced by an unconditional branch to the loop's exit block. This is done, because each thread only executes one iteration, as outlined in section 4.4.1. Also, the induction variable is now calculated from the thread's index instead of the induction variable:

$$\text{<thread index>} + \text{<block index>} * \text{<block dimension>} + \text{<lower bound>} \tag{5.1}$$

The next transformation creates a new function that prepares the kernel arguments and calls the runtime function `launchKernel` to start the kernel. The arguments have to be passed as an array of `void*` pointers to the CUDA API, which is why we allocate memory on the stack for each argument passed to the kernel and then collect the pointers to these arguments in an array. Arguments that are pointers themselves have to be mapped to the corresponding address they point to on the device. The `getDevicePtr` runtime function does this by first looking up the interval this pointer belongs to, which can then be used to look up the interval's position relative to the beginnig of its block. This offset is then added to the device address of the beginning of the block to obtain the device address this specific pointer points to.

The runtime function `launchKernel` first loads the required kernel from the ptx module that has been produced by the GPU transformations. It then calculates the block width and the grid width by this formular:

$$\text{block width} = \begin{cases} \text{block size} & \text{, if upper bound} > \text{block size} \\ \text{upper bound} & \text{, otherwise} \end{cases} \tag{5.2}$$

$$\text{grid width} = \frac{\text{upper bound} + \text{block width} - 1}{\text{block width}} \tag{5.3}$$

Finally, `launchKernel` starts the kernel via the `cuLaunchKernel` call, which is part of the CUDA Driver API. Passed to this call are the computed block and grid widths, and the `void*` array.

# 6. Evaluation

We evaluate our tool by transforming three algorithms implemented in the polybench benchmark suit [pol]. These algorithms are `gemm`, `gesummv` and `syrk`.

We evaluate the algorithms in different configurations. A configuration is the triple of *input size*, *input form* and *ratio*. The first one dictates the overall size of the input, e.g. the size of an $n \times n$ matrix is proportional to $n^2$. The input form describes the input dimensions and their individual sizes. As an example, an $n \times m$ matrix has two dimensions of size $n$ and $m$ respectively. The third element ratio indicates the amount of work that is executed on the GPU. A configuration with a ratio of 0.8 runs 80% of iterations on the GPU.

We do not use the autotuner during evaluation, but rather set the partitioning ratio ($r$) manually. This is done via a command line flag `-r` passed to our tool. The autotuner is excluded to make it easier to obtain deterministic results during testing. By looking at the measuring data, we will still be able to determine whether it would be beneficial to use an autotuner. If the curve of the speedup over the different values of $r$ lacks any local maxima, we can assume an autotuner would find the optimal partitioning ratio.

Each algorithm is tested in 24 different configurations. We use two different input datasets and twelve different values of ratio. We think, one of the datasets is better suited for execution on the CPU, the other for execution on the GPU. The first eleven ratio values start from 0 and go up to 1 stepping by 0.1. The twelfth ratio value is also 0, but the program is compiled only by `clang` and not processed by our tool in any way. The execution time of this configuration is used as a base line to later compute the achieved speedup. Figure 6.1 describes how said speedup $S$ is computed from the execution time of the sequential version $T_s$ and the partitioned version $T_p$ respectively.

$$S = \frac{T_s}{T_p} \tag{6.1}$$

Each algorithm is contained in a single file which is compiled with the following polybench specific compiler flags:

- `-DPOLYBENCH_TIME`
- `-DPOLYBENCH_USE_RESTRICT`
- `-DPOLYBENCH_USE_SCALAR_LB`

The first flag controls the insertion of time measuring calls into the polybench files to obtain the execution time of the program. The second flag adds the `restrict` keyword to the algorithms' input data. This is necessary, because the Alias Analysis is not able to assess that the given pointers point to destinct memory regions. The third flag tells polybench to use constant loop bounds instead of variables. The partitioning algorithm is only able to statically detect the loop bounds, if the induction variable is unsigned. As polybench uses `int` as the data type for the loop bounds, we replace the variable bounds by constants.

After compilation, the algorithm under evaluation is analyzed and transformed by our tool to produce the program used for measurements. We then execute the program ten times, discarding the fastest and the slowest run. The remaining data points $d_i$ are averaged by summing them up and dividing the sum by the number of remaining data points $n$. This average is the execution time $t$ for one specific configuration of problem size and partitioning ratio. Figure 6.2 describes this in mathematical notation.

$$\mathrm{t} = \sum_{i=0}^{n} \frac{d_i}{n} \tag{6.2}$$

We also compute the maximum deviation of the individual data points from the computed average. If a data point from a specific configuration deviates more than 5%, we repeat the evaluation of this configuration. The maximum deviation $\delta_{max}$ is computed as follows:

$$\delta_{max} = \max\{|d - t|, \forall d \in M\} \tag{6.3}$$

We evaluate on a machine with the Intel Xeon Processor containing 4 cores (8 hardware threads) clocked at 3.7 GHz. The GPU device is a GeForce GTX TITAN Black with 2880 *CUDA* cores (distributed over 15 Multiprocessors) with 889 MHz base and 980 MHz maximum clock rate and a memory bandwidth of 336 GB/s.

## 6.1 gemm

This algorithm implements general matrix multiplication taking the following parameters as input:

- $\alpha, \beta \in \mathbb{R}$
- $A \in \mathbb{R}^{ni \times nk}$
- $B \in \mathbb{R}^{nk \times nj}$
- $C \in \mathbb{R}^{ni \times nj}$

It then computes the output $C_{out} \in \mathbb{R}^{ni \times nj}$ using the following formular:

$$C = \alpha AB + \beta C \tag{6.4}$$

The source code of the function and loops to be transformed is shown in figure 6.1. We postulate that the sizes of the three matrices ($A$, $B$ and $C$) will have a significant effect on the execution time of the transformed function. As our tool partitions the outer loop and leaves the inner ones unchanged, altering the loop bound $ni$ will probably have different effects than altering $nk$ and $nj$. Iterations in the GPU partition will be executed concurrently, thus a greater $ni$ should increase the number of parallel computations, while a smaller one should decrease it. The inner loops, on the other hand, are executed sequentially. This means that their bounds $nj$ and $nk$ define the amount of work for a single

```
void kernel_gemm(
  int ni, int nj, int nk,
  double alpha, double beta,
  double C[ni][nj], double A[ni][nk], double B[nk][nj]) {
  for (int i = 0; i < ni; i++) { // (1)
    for (int j = 0; j < nj; j++)
      C[i][j] *= beta;
    for (int k = 0; k < nk; k++) {
      for (int j = 0; j < nj; j++)
        C[i][j] += alpha * A[i][k] * B[k][j];
    }
  }
}
```

Figure 6.1: Source code of the *gemm* algorithm.

thread on the GPU. Thus, to take advantage of the GPU's massively parallel structure, we believe to see the best results with a high value for $ni$ in comparison to the product of $nj$ and $nk$. For this reason we choose two problem sizes: The first one is the extra large configuration of the polybench suite, with values of $ni = 2 \cdot 10^3, nj = 2.3 \cdot 10^3$ and $nk = 2.6 \cdot 10^3$. The second one was added by us, allowing for significantly more iterations in the outer loop, with values of $ni = 10^6, nj = 100$ and $nk = 100$.

### 6.1.1 Description

The results of evaluating this algorithm are shown in graph form in figure 6.2 and in tabular form in table 6.1.1. The purple graph ($A$) starts on a plateau, where we see the global maximum speedup at $r = 0.1$ of $S \approx 0.4656$ and two slightly smaller values for $r = 0$ and $r = 0.2$. It then falls exponentially between $r = 0.2$ and $r = 0.5$ to a value of $S \approx 0.2024$. From there on its maximum deviation from the value at $r = 0.5$ amounts to approximately 1.5%, thus it basically stays constant. The second graph ($B$) in green has its maximum of $S \approx 0.3760$ at $r = 0.4$ and a slightly smaller value for $r = 0.5$. From $r = 0$ to $r = 0.4$ the graph shows a small exponential increase and from $r = 0.5$ to $r = 1$ a small exponential decrease.

| ratio | $ni = 2 \cdot 10^3, nj = 2.3 \cdot 10^3, nk = 2.6 \cdot 10^3$ | $ni = 10^6, nj = 100, nk = 100$ |
|---|---|---|
| 0.0 | 1.00000000 | 1.00000000 |
| 0.0 | .43670546 | .24464347 |
| 0.1 | .46556106 | .25883840 |
| 0.2 | .44933685 | .28877814 |
| 0.3 | .32420609 | .32656198 |
| 0.4 | .23301539 | .37598232 |
| 0.5 | .20237398 | .36451729 |
| 0.6 | .20541951 | .30645426 |
| 0.7 | .20329812 | .26448068 |
| 0.8 | .20316972 | .23222210 |
| 0.9 | .20425854 | .20808944 |
| 0.9999 | .20123451 | .18784057 |

Table 6.1: The data points for the *gemm* algorithm.

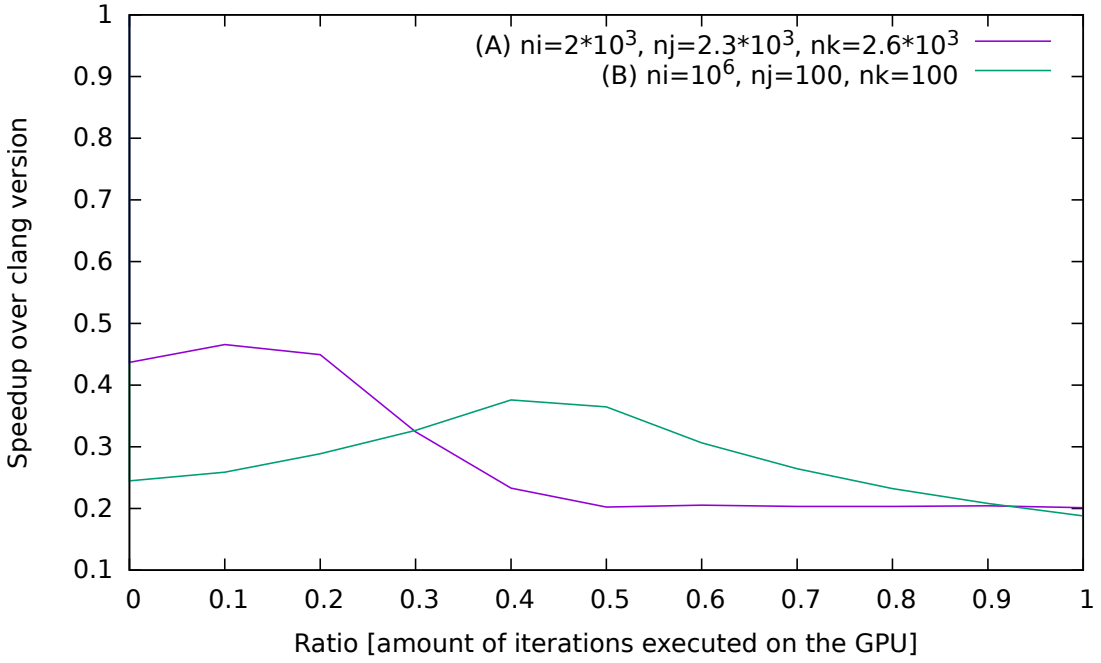


Figure 6.2: The speedups of the *gemm* algorithm for two differently shaped inputs over the ratios from 0 to 1.

### 6.1.2 Interpretation

The first thing we notice is that the maximum speedups are still smaller than one, which means that our tool slows this program down by at least a factor of 2 when compared to an executable generated by `clang`. It is nevertheless interesting to inspect the curve shapes and discuss potential explanations.

The purple graph ($A$) shows the best performance at around $r = 0.1$, which means that only 10% of the loop's iterations are executed on the GPU, the rest is executed on the CPU. This result can be explained by looking at the differing clock speeds of CPU and GPU and the form of the input data. A thread on the CPU has a much higher (by a factor of 3.78) clock speed than an individual thread on the GPU. Additionally, as our tool only partitions the outer loop, each thread on the GPU sequentially executes $O(nk \cdot nj)$ operations. Thus the GPU operates highly parallel, but each thread is slower than the CPU. Another aspect to consider is the amount of data that has to be copied to and from GPU memory before and after execution. The more data we copy the higher the additional speedup contributed by the GPU has to be in order to achieve an overall performance increase. We see how at $r = 0.1$ the speedup is high enough to outweigh the copying cost, but as we increase $r$, the copying cost increases and outweighs the speedup from $r = 0.3$ onwards.

Looking at the green graph ($B$), we can see that configurations with a different input shape allow for more GPU participation. The work to be done by any particular thread has decreased by a factor of 598, compared to earlier configurations. This, in combination with a high number of outer loop iterations ($ni = 10^6$), leads to a 40% GPU participation to achieve the best speedup.

## 6.2 gesummv

The *gesummv* algorithm implements summed matrix-vector multiplications taking the following parameters as input:

```
void kernel_gesummv(int n,
  double alpha, double beta,
  double A[n][n], double B[n][n],
  double tmp[n], double x[n], double y[n]) {
  for (int i = 0; i < n; i++) {
    tmp[i] = 0.0;
    y[i] = 0.0;
    for (int j = 0; j < n; j++) {
      tmp[i] = A[i][j] * x[j] + tmp[i];
      y[i] = B[i][j] * x[j] + y[i];
    }
    y[i] = alpha * tmp[i] + beta * y[i];
  }
}
```

Figure 6.3: Source code of the *gesummv* algorithm.

- $\alpha, \beta \in \mathbb{R}$
- $A, B \in \mathbb{R}^{n \times n}$
- $x \in \mathbb{R}^n$

It then computes the output $y \in \mathbb{R}^n$ using the following formular:

$$y = \alpha Ax + \beta Bx \tag{6.5}$$

The source code of the function and loops to be transformed is shown in figure 6.2. For the same reasons, as layed out in section 6.1, we think our tool will perform better the greater the outer loop bound $n$ is. However, given that the only inner loop is also bounded by $n$, the work each individual thread has to do will increase proportionally to the increase of available threads. We choose two problem sizes: The first one is again the extra large configuration of the polybench suite with a value of $n = 2 \cdot 10^3$, while the second one was added by us and uses a value of $n = 10^4$. Significantly larger values for $n$ are not possible, as we run out of memory when handling data of size $O(n^2)$.

### 6.2.1 Description

The results of evaluating this algorithm are shown in graph form in figure 6.4 and in tabular form in table 6.2.1. Both graphs very similar: The maximum speedup of approximately 0.390 is achieved at $r = 0$, after which the curve drops to approximately 0.014 for the purple graph $(A)$ and to approximately 0.121 for the green graph $(B)$. From $r = 0.1$ to $r = 0.999$ $(B)$ falls with a slope of 0.9521. $(A)$ falls with a slope of 0.8884 between $r = 0.1$ and $r = 0.7$ and then the slope increases to 0.7531 from there to $r = 0.999$.

### 6.2.2 Interpretation

The transformed versions of the *gesummv* algorithm are ten times (with $n = 10^4$) and nearly 100 times (with $n = 2 \cdot 10^3$) slower than their equivalent compiled by `clang`. This indicates that the performance improvements gained by parallel computing on the GPU are mitigated by the costs of copying memory between the two devices. The reason for this is probably the amount of work per GPU thread in comparison to the amount of data that has to be copied. The inner loop has a complexity of $O(n)$, while the amount of data to transfer is $O(n^2)$. However, following this reasoning, curve $(A)$ should show a greater speedup than curve $(B)$ because the former displays configurations with a smaller $n$. As we can see, this is clearly not the case. The reason for this is probably that the GPU can only utilize $2 \cdot 10^3$ threads in the case of $(A)$ versus the $10^4$ threads in the case of $(B)$.

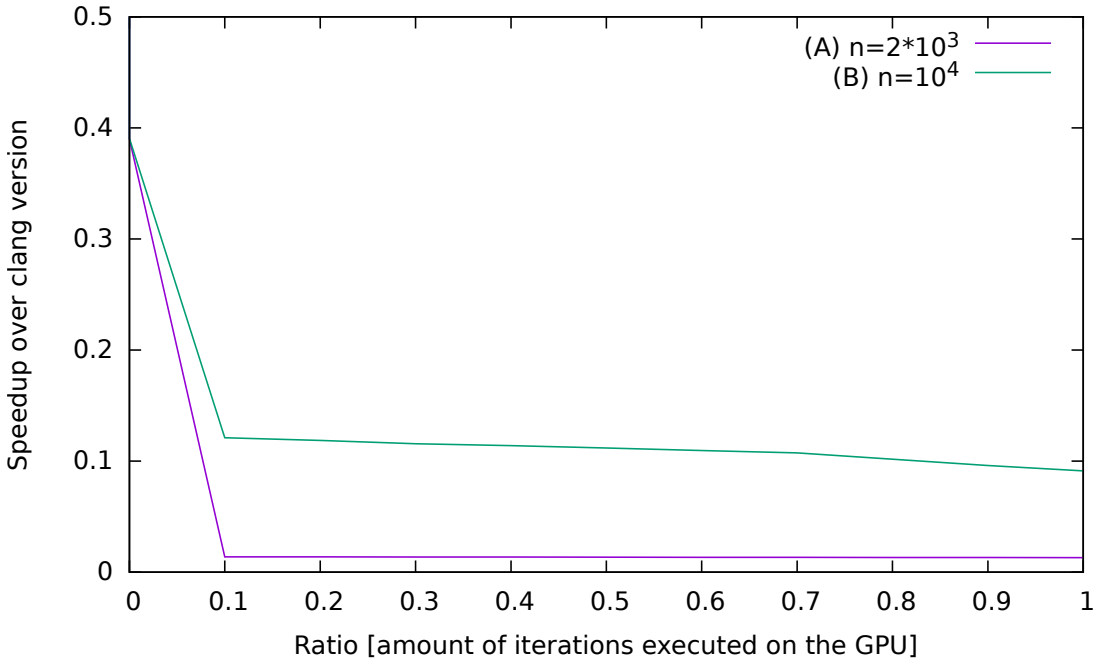


Figure 6.4: The speedups of the *gesummv* algorithm for two differently shaped inputs over the ratios from 0 to 1.

## 6.3 syrk

The *syrk* algorithm implements a symmetric rank-k update taking the following parameters as input:

- $\alpha, \beta \in \mathbb{R}$
- $A \in \mathbb{R}^{n \times m}$
- $B \in \mathbb{R}^{n \times n}$ (symmetrix matrix)

It then computes the symmetric matrix $Cout \in \mathbb{R}^{n \times n}$ as output by using the following formular:

$$Cout = \alpha A A^T + \beta C \tag{6.6}$$

| ratio | $N = 2 \cdot 10^3$ | $N = 10^4$ |
|---|---|---|
| 0.0 | .39007105 | .39052280 |
| 0.1 | .01377109 | .12101976 |
| 0.2 | .01373680 | .11858699 |
| 0.3 | .01370285 | .11566120 |
| 0.4 | .01356957 | .11396686 |
| 0.5 | .01349990 | .11179276 |
| 0.6 | .01331785 | .10945476 |
| 0.7 | .01333627 | .10745027 |
| 0.8 | .01325449 | .10164490 |
| 0.9 | .01318657 | .09608380 |
| 0.9999 | .01311388 | .09118687 |

Table 6.2: The data points for the *gesummv* algorithm.

```
void kernel_syrk(int n, int m,
  double alpha, double beta,
  double C[n][n], double A[n][m]) {
  for (int i = 0; i < n; i++) {
    for (int j = 0; j <= i; j++)
      C[i][j] *= beta;
    for (int k = 0; k < m; k++) {
      for (int j = 0; j <= i; j++)
        C[i][j] += alpha * A[i][k] * A[j][k];
    }
  }
}
```

Figure 6.5: Source code of the *syrk* algorithm.

| ratio | $n = 2.6 \cdot 10^3$, $m = 2 \cdot 10^3$ | $n = 10^4$, $m = 100$ |
|---|---|---|
| 0.0 | .88734477 | .88287985 |
| 0.1 | .85796596 | .83641286 |
| 0.2 | .86920821 | .85634764 |
| 0.3 | .90889122 | .88759596 |
| 0.4 | .97105412 | .94637566 |
| 0.5 | 1.07440160 | 1.04047315 |
| 0.6 | 1.22742361 | 1.19415669 |
| 0.7 | 1.03420614 | 1.45157061 |
| 0.8 | .78939777 | 1.57920408 |
| 0.9 | .78876578 | 1.26916038 |
| 0.9999 | .78707533 | 1.03414331 |

Table 6.3: The data points for the *syrk* algorithm.

The source code of this algorithm is shown in figure 6.3. As with the previous evaluations, a big value of $n$ and a comparatively small value of $m$ will help us to better utilize the GPU. For this reason, we set $n = 10^4$ and $n = 10^2$ for one set of configurations. The other set is the extra large dataset from the polybench suite with values of $n = 2.6 \cdot 10^3$ and $m = 2 \cdot 10^3$.

### 6.3.1 Description

The purple curve $(A)$ and the green curve $(B)$ have a very similar shape in the interval $r = [0, 0.6]$, with $(A)$ approximately 1.5 to 3.0 percent points above $(B)$. When running without the GPU, their performance lies slightly below 90% of the perfomance of their `clang` compiled counterparts. They drop at $r = 0.1$ to reach a local minimum for $(A)$ and a global one for $(B)$. At approximately 30% of GPU participation they reach the same performance as without the GPU. Another interesting point is at $r \approx 0.45$ where they are as fast as their `clang` counterparts. $(A)$ increases exponentially for $r \in [0.1, 0.6]$ with a speedup of 1.23 at its maximum. It then drops off dramatically to reach its minimum of 0.78 at $r = 0.8$ and stays constant for the remaining ratios. $(B)$, on the other hand, reaches its maximum of 1.45 at $r = 0.8$ after increasing exponentially as well over the interval $r = [0.1, 0.7]$. It then drops off as dramatically as $(A)$ to reach 1.03 at $r = 0.999$.

### 6.3.2 Interpretation

After being transformed by our tool, this algorithm runs up to 1.45 times faster than its `clang` compiled counterpart. The maximum of 1.45 is reached with a dataset that requires

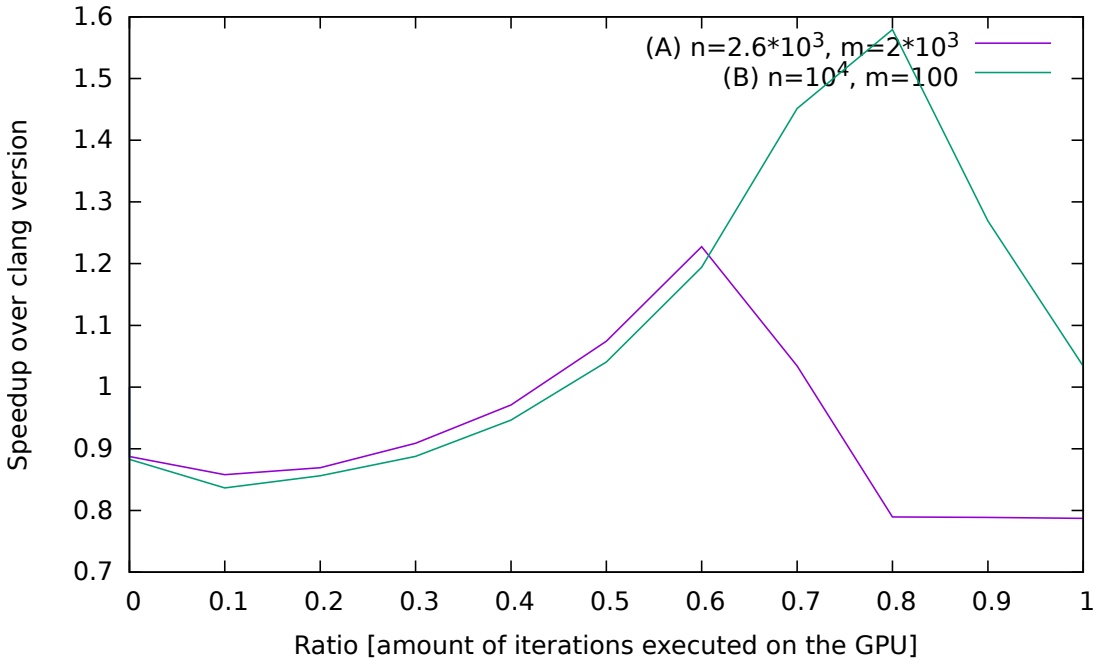


Figure 6.6: The speedups of the *syrk* algorithm for two differently shaped inputs over the ratios from 0 to 1.

a lot of iterations of the outer loop and few of the inner loops. As previously discussed, this shape of the input data is a better suited for the GPU than a data set with a smaller ratio of $n$ to $m$. Although the dataset $B$ has such a smaller ratio of $n$ to $m$, we still reach a speedup of 23%. The *gemm* algorithm reached a speedup of only 0.47 with a similarly shaped input dataset. The explanation for this might be, that the *syrk* algorithm is able to exploit the fact that A is a symmetric matrix. It thus only needs half as many iterations for matrix multiplication as the matrix multiplication in *gemm*.

## 6.4 Summary

Evaluating the three algorithms has shown, that the use of an autotuner is possible. An autotuner will be able to find the optimal ratio between CPU and GPU, for each of the tested algorithms, because all of them have a global maximum. In the case of *gemm* and *syrk* this maximum is achieved with a GPU participation of 10% or 40% and 60% or 80% respectively. The former only reaches a speedup of 0.45 because a single GPU thread has to much work to do and runs at a significantly slower clock speed than the CPU thread. The latter reaches a speedup of up to 1.45 compared to the *clang* compiled counterpart. The *gesummv* algorithm has the greatest speedup without the GPU. This can be attributed to the fact that the amount of work to be done is to small to warrant the cost of transferring data and starting a kernel.

# 7. Conclusion

This final chapter summarizes this thesis in 7.1 and then proposes further approaches to solve remaining problems and questions in 7.2.

## 7.1 Summary

In this thesis we described the conception and implementation of a tool that automatically partitions loops for heterogeneous systems. This was achieved by extending the `AutoCU`. We added the functionality of partitioning a loop into two loops. While doing this we replaced the memory management system found in [Bï5]. The new system is based on Scalar Evolution analysis and able to statically analyze the memory accessed by a given loop. The previous approach did this at runtime.

We evaluated our implementation by using it to transform three algorithms taken from the polybench suite [pol]. This showed us, that it is feasible to use an autotuner to find the optimal partitioning ratio. We also learned, that our tool can achieve a speedup of up to 45% under circumstances which allow for efficient GPU usage. However, we also saw that our tool can slow a program down to only 40% of its original performance, even if the GPU is not utilized at all.

## 7.2 Future Works

The evaluation has shown, that in some cases the usage of the GPU does not yet increase the performance of the transformed program. One reason for this is the cost of transferring data between host memory and GPU memory. An attempt to remedy this, would be the usage of asynchronous memory transfer. The tool currently uses the synchronous API to copy memory between devices leading to time slots during which neither device does any computational work. The application of asynchronous copying APIs will probably shrink these idle time slots, as the CPU will be able already run computational tasks, while the GPU is still waiting for its data to arrive. Another solution would be to spawn a separate thread on the CPU tasked with transferring memory, while the main thread can immediately execute computational tasks.

Our tool currently executes the CPU partition in a sequential fashion. As CPU are usually multi-core processors, we could increase performance by parallelizing these partitions as well. A possible implementation could utilize Polly [GGL12] and its OpenMP back-end.

Another result of the evaluation was the fact that our tool can be detrimental to a program's performance, even if it does not use the GPU and thus should be equivalent to the original. The reason for this might be the extraction of a partitioned loop into two separate functions, which then inhibits certain optimizations that would otherwise be possible. A future work could examine the partitioning process investigate if this is the reason. If this is the case, one could go on to try and find a solution for this. If this is not the case, investigation of different aspects of the partitioning process might be appropriate.

# Bibliography


[BÏ5] L. Böhm, "Compile-time code parallelization for self-adapting acceleration on intel xeon phi," 2015.

[boo] http://www.boost.org/ [Last accessed on 2015-11-28].

[CBM+08] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron, "A performance study of general-purpose applications on graphics processors using cuda," *Journal of parallel and distributed computing*, vol. 68, no. 10, pp. 1370–1380, 2008.

[DP14] M. Damschen and C. Plessl, "Easy-to-use on-the-fly binary program acceleration on many-cores," *arXiv preprint arXiv:1412.3906*, 2014.

[FOW87] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 3, pp. 319–349, Jul. 1987. [Online]. Available: http://doi.acm.org/10.1145/24039.24041

[GGL12] T. Grosser, A. Groesslinger, and C. Lengauer, "Polly–performing polyhedral optimizations on a low-level intermediate representation," *Parallel Processing Letters*, vol. 22, no. 04, 2012.

[KP11] T. Karcher and V. Pankratius, "Run-time automatic performance tuning for multicore applications," in *Euro-Par 2011 Parallel Processing*. Springer, 2011, pp. 3–14.

[LHK09] C.-K. Luk, S. Hong, and H. Kim, "Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping," in *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*. IEEE, 2009, pp. 45–55.

[llv] http://llvm.org/devmtg/2009-10/ScalarEvolutionAndLoopOptimization.pdf [Last accessed on 2016-01-13].

[MLZB14] D. Mikushin, N. Likhogrud, E. Z. Zhang, and C. Bergstrom, "Kernelgen–the design and implementation of a next generation compiler platform for accelerating numerical models on gpus," in *Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*. IEEE, 2014, pp. 1011–1020.

[pol] http://web.cse.ohio-state.edu/~pouchet/software/polybench/ [Last accessed on 2016-01-09].

[RDP14] B. Ranft, O. Denninger, and P. Pfaffe, "A stream processing framework for on-line optimization of performance and energy efficiency on heterogeneous systems," in *Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*. IEEE, 2014, pp. 1039–1048.

[SHZH13] K. Streit, C. Hammacher, A. Zeller, and S. Hack, "Sambamba: runtime adaptive parallel execution," in *Proceedings of the 3rd International Workshop on Adaptive Self-Tuning Computing Systems*. ACM, 2013, p. 7.

[Sut] H. Sutter, "The free lunch is over: A fundamental turn toward concurrency in software."

[VCJC+13] S. Verdoolaege, J. Carlos Juega, A. Cohen, J. Ignacio Gómez, C. Tenllado, and F. Catthoor, "Polyhedral parallel code generation for cuda," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 9, no. 4, p. 54, 2013.

[WPD01] R. C. Whaley, A. Petitet, and J. J. Dongarra, "Automated empirical optimizations of software and the atlas project," *Parallel Computing*, vol. 27, no. 1, pp. 3–35, 2001.