FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

# Towards Interactive Visual Exploration of Parallel Programs using a Domain-specific Language

MASTERARBEIT

zur Erlangung des akademischen Grades

**Master of Science**

im Rahmen des Studiums

**Medieninformatik und Visual Computing**

eingereicht von

**B.Sc. Tobias Klein**
Matrikelnummer 1129861

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. M. Eduard Gröller
Mitwirkung: Dipl.-Ing. Dr.techn. Peter Rautek

Wien, 24. November 2015

| Tobias Klein | M. Eduard Gröller |
|---|---|

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

# Towards Interactive Visual Exploration of Parallel Programs using a Domain-specific Language

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

**Master of Science**

in

**Media Informatics and Visual Computing**

by

**B.Sc. Tobias Klein**
Registration Number 1129861

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. M. Eduard Gröller
Assistance: Dipl.-Ing. Dr.techn. Peter Rautek

Vienna, 24th November, 2015

Tobias Klein

M. Eduard Gröller

# Erklärung zur Verfassung der Arbeit

B.Sc. Tobias Klein
Trappelgasse 3/2/16, 1040 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 24. November 2015

Tobias Klein

# Danksagung

Zuerst möchte ich meine allgemeine Dankbarkeit an alle richten, die mich während meines Studiums an der Technischen Universtät Wien, sowie bei dieser Arbeit unterstützt und immer wieder ermutigt haben.

Ich möchte ebenso meine Dankbarkeit an Peter Rautek verzeichnen für seine umfassende Hilfe während der Erstellung dieser Arbeit. Außerdem, danke ich dem gesamten Team des Visual Computing Centers der King Abdullah University of Science & Technology. Sie sind jederzeit für Hilfe bereit gewesen und versorgten mich stets mit nützlichen Hinweisen und Einsichten.

Desweiteren danke ich Eduard Gröller im Namen der Technischen Universität Wien und Stefan Bruckner für ihren konstruktiven Rat und ihre Empfehlungen.

Außerdem möchte ich meine Dankbarkeit an das International Office der Technischen Universität Wien senden für die finanzielle Unterstützung während meines Aufenthaltes an der King Abdullah University of Science & Technology.

Zuletzt möchte ich die Gelegenheit nutzen, um mich bei meiner Familie und meiner Freundin zu bedanken, für ihre kontinuierliche Unterstützung, sowie für die Ermöglichung meines Studiums.

# Acknowledgements

First, I would like to express my deep sense of gratitude to everyone who supported and encouraged me during my studies at the Technical University of Vienna and helped me to develop this work.

I also want to record my thankfulness to my supervisor Peter Rautek for his profound help during the creation of this thesis and to the whole team of the Visual Computing Center at King Abdullah University of Science & Technology. They were always willing to help and provided useful insights during the whole work-process.

Furthermore, I would like to thank Eduard Gröller on behalf of the Technical University of Vienna and Stefan Bruckner for their constructive advice and guidance.

I want to extend my gratitude to the International Office of the Technical University of Vienna, which financially supported my visit at King Abdullah University of Science & Technology.

Finally, I take this opportunity to thank my family for their continuous support and for the possibility to accomplish my studies.

# Kurzfassung

Der Einsatz von Grafikkarten und das Paradigma von massiven parallelen Berechnungen ist in vielen Forschungsgebieten immer relevanter geworden. Aktuelle Entwicklungen von Plattformen, wie OpenCL und CUDA, ermöglichen den Einsatz von heterogenen parallelen Berechnungen in diversen Gebieten. Die effiziente Nutzung von paralleler Hardware erfordert jedoch sowohl tiefgründige Kenntnisse im Bereich der parallelen Programmierung als auch der Hardware selber.

Unser Ansatz beschreibt eine domänen-spezifische Sprache, die eine schnelle Entwicklung von parallelen Programmen ermöglicht, sowie Visualisierungen, die das Laufzeitverhalten von parallelen Programmen darstellt. Unsere Visualisierungen zeigen die konkreten Interaktionen mit der Hardware, was das Verständnis des Programms unterstützt und die Ausnutzung der einzelnen Hardwarekomponenten aufzeigt. Des weiteren aggregieren wir Verhalten, welches zu bekannten Problemen in parallelen Programmen führt und präsentieren anhand dieser Daten strukturierte visuelle Ansichten.

Wir erweitern Standardmethoden wie Debugging und Profiling mit einem visuellen Ansatz, der eine problemspezifischere und detailliertere Analyse des Programms bietet. Unser System liest und interpretiert Programmcode zusammen mit benutzerspezifischen Annotierungen und ermöglicht damit eine automatische Instrumentierung des Programmes. Die daraus entstehenden Daten werden direkt mit der bekannten D3 (data-driven documents) Bibliothek visualisiert.

Eine Demonstration unseres Ansatzes zeigen wir in zwei Fallstudien. Dabei nehmen wir eine visuelle Analyse von Bankkonflikten vor und zeigen Divergenzen im Kontrollfluss des Programmes auf. Die Fallstudien untersuchen zum einen verschiedene Implementierungen von paralleler Reduktion und zum anderen ein übliches Beispiel aus der Bildverarbeitung (alle Beispiele sind dem OpenCL SDK von NVIDIA entnommen). Wir zeigen anhand dieser Beispiele, dass unsere Visualisierung ein direktes Verständnis über das Verhalten des Programms erleichtert. Des Weiteren wird ersichtlich, dass sich die Effizienz von den verschiedenen Implementierungen der parallelen Reduktion direkt in unseren Visualisierungen widerspiegelt.

# Abstract

The utilization of GPUs and the massively parallel computing paradigm have become increasingly prominent in many research domains. Recent developments of platforms, such as OpenCL and CUDA, enable the usage of heterogeneous parallel computing in a wide-spread field. However, the efficient utilization of parallel hardware requires profound knowledge of parallel programming and the hardware itself.

Our approach presents a domain-specific language that facilitates fast prototyping of parallel programs, and a visual explorer which reveals their execution behavior. With the aid of our visualizations, interactions with the hardware become visible, supporting the comprehensibility of the program and its utilization of the hardware components. Furthermore, we aggregate behavior that leads to common issues in parallel programming and present it in a clearly structured view to the user.

We augment the standard methods for debugging and profiling by a visual approach that enables a more problem-specific, fine-grained way of analyzing parallel code. Our framework parses all program code and user-specified annotations in order to enable automatic, yet configurable code instrumentation. The resulting recordings are directly linked to interactive visualizations created with the well-known D3 (data-driven documents) framework.

To demonstrate our approach, we present two case studies about the visual analysis of memory bank conflicts and branch divergence. They investigate different parallel reduction implementations and a common image processing example (all from the NVIDIA OpenCL SDK). We show that our visualizations provide immediate visual insight in the execution behavior of the program and that the performance influence of the implementations is directly reflected visually.

# Contents

CHAPTER 1

# Introduction

”The free lunch is over“ [Sut05]. Sutter reveals that the performance boost of central processing units (CPUs) seems to plateau, contrary to the exponential growth during the previous decades as predicted by Moore’s law. It is a well-known claim that ”parallelism is the future of computing“ [OHL+08]. Computational requirements increase continuously and parallel computing provides the means to tackle this issue. Especially the utilization of the graphics processor unit (GPU) for general parallel computations is common practice nowadays [ND10]. The GPU has evolved from a pure graphics processor into a general programmable parallel processor. The ubiquitous access of the GPU, which is present on desktop computers, hybrid compute clusters and even in laptops and smartphones facilitates the wide-spread usage of parallel computing. The parallel computing paradigm has also become increasingly prominent in many research domains, being used to solve the demands of various scientific challenges.

A brief look into history reveals the processing capabilities of the GPU. Until the year 2000, Intel’s ASCI Red [SDS+] supercomputer was the world’s most powerful computer with 1,34 teraFLOPS (floating-point operations per second). It occupied 1,600 square feet [MH98] of floor-space and required 850kW. In 2013, NVIDIA exceeded the computational capabilities of this supercomputer with a single video card. The GTX Titan, equipped with a Kepler GPU, has a computational power of 4,5 teraFLOPS [JLP+13] and only consumes 250W. This is more that three times the computational power of the world’s fastest supercomputer in the year 2000 and it only requires a fraction of its space and energy.

However, the fundamental principle of parallel computing is based on the fact that tasks are solved in a parallel manner. These tasks are broken into discrete parts, which are then carried out simultaneously in order to improve the performance. The real world consists of countless events that are massively parallel, hence parallel computing has applications in diverse areas, such as medical imaging and diagnosis, economic modeling, climate simulation and data mining. Nevertheless, not every task is parallelizable because sequential constraints can occur. An instance is illustrated by the women-child example, which reads as follows: One woman can carry a child in nine months, however, nine women are not able to carry one

child in one month. ”The bearing of a child takes nine months, no matter how many women are assigned“ [Bro75]. Despite the sequential constraints, even this task can be parallelized through replication. Hundred women can carry out hundred children in nine months.

The progression of parallel computing has entailed a number of new developments. Besides the evolution of the GPU, platforms like OpenCL [SGS10] and CUDA [NBGS08] established the utilization of parallel computing through the definition of general interfaces for the usage of parallel hardware. These tools offer direct access to parallel computation elements of the processing units. Programmers of parallel applications require a deep knowledge of the underlying hardware architecture in order to implement programs that execute efficiently. Furthermore, parallel computing differs fundamentally from sequential computing. Traditional approaches are not applicable anymore and the implementation, debugging and profiling of parallel applications is comparably tedious. For instance, the concept of a breakpoint is not directly applicable to parallel programs, since these programs are executed on multiple execution units. To apply breakpoints in parallel programs, a compatible break condition must be formulated. For instance, the break condition could define a stop when the first execution unit encounters the breakpoint. However, at the time the execution stops, only the specified unit is guaranteed to be in the execution state of where the breakpoint is located in the program's structure

Furthermore, achieving efficient parallel code is a complex optimization problem, and a time-consuming, error prone task [JTD+12]. The problem is multi-objective since different constraints influence the performance of the program. Optimizing for instruction-throughput may decrease memory-throughput and thereby not benefit the overall performance, and vice-versa. This is the reason for programmers having to deal with complex code, errors, and performance guess work.

Parallel program visualization addresses these issues by presenting visual representations of the program's intrinsic behavior. It offers techniques, which aid developers in improving the performance of their software. Performance visualization deals with the collection of performance data as well as their presentation to the user. In addition, the visualizations do not only support performance improvements, but also the general understanding of the execution behavior, and the detection of erroneous code.

## 1.1 Motivation

Researchers frequently encounter programming challenges that could be solved efficiently in a parallel manner but they often possess only limited experience in parallel programming. A user study of parallel libraries reveals that developers misuse them so that their code runs sequentially instead of in parallel and that this yields unnecessarily complex code [OD12]. This illustrates the complexity and importance behind the understanding of parallel programming. The execution of parallel programs is often seen as a black box, where input and output are known, but only little is displayed about the execution of the code. However, this is crucial for the understanding, correctness and performance. Tools like NVIDIA NSight [NVIc], Visual Profiler [NVIf] and VampirTrace [JBK+07] tackle this issue and support debugging, profiling

and analysis of parallel programs. These tools provide facts on the application-level such as idle times of the processors, timings of memory transactions, as well as statistics on the kernel-level, such as branch divergence, memory bank conflicts and occupancy.

In essence, the above mentioned tools are devised to show "*What* is going wrong". In contrast to these tools, the aim of this work is to allow programmers to quickly test their hypothesis on "*Why* is something going wrong". For parallel program visualization, we collect execution data and map it onto visual representations that follow the semantics of the underlying hardware architecture. We provide rapid insight into the program behavior and facilitate interactive exploration. Furthermore, rather than offering a static analysis and a fixed set of statistics, we enable the user to express fine-grained analyses of a specific problem through code annotations and dynamic instrumentation of GPU kernels.

## 1.2 Contribution

Our framework comprises three major components:

(1) the definition of a domain-specific language (DSL) for writing kernel code,

(2) a source-to-source compiler that performs just-in-time compilation of the DSL to fully instrumented OpenCL code, and

(3) a powerful interactive visualization interface based on the D3 (data-driven documents) visualization framework [BOH11].

Our approach enables a very specific, user-centered analysis, both in terms of the code instrumentation and the visualization itself. However, instead of having to manually write code that constructs the visualization, simple code annotations are introduced. They determine configurations for the source-to-source compiler to automatically generate instrumented code, which gathers additional data during the execution. The visualization part of our framework enables the interactive analysis of kernel run-time behavior in a way that is specific to a particular problem or optimization goal. Examples are: the analysis for causes of memory bank conflicts, the investigation of branch divergence, and the understanding of different implementations of a parallel algorithm. By using a clearly defined interface between data generation and visualization, the users can extend our visualizations or write their own. Our developed prototype targets OpenCL kernels. However, the same approach is applicable to the CUDA platform [NVIa] since the introduction of its run-time compilation feature.

The framework provides capabilities for visual program analysis that go beyond existing applications. We augment the standard methods for debugging and profiling by a visual approach that enables a more problem-specific, fine-grained way of analyzing code. Our framework parses all kernel code and user-specified annotations, which enables automatic code instrumentation. The recorded program behavior is displayed and the interactive visualizations are directly linked to the kernel code.

Furthermore, we provide an evaluation of our visual exploration approach with the aid two case studies. For this purpose, two algorithms are investigated, first, an image filtering example, which shows the impact of branch divergence on our visualizations. Second, we present a case study of multiple implementations of parallel reductions that lead to different performance characteristics.

## 1.3 Structure

This thesis is structured in the following way: In Chapter 2, a contextual overview of the fundamentals of this work is presented, serving as a more detailed introduction to parallel programs and the execution hardware. Related work in the realm of parallel program visualization, as well as domain-specific languages is reviewed in Chapter 3. The main methodology of this work is presented in Chapter 4, which describes the utilization of visual exploration for parallel programs. Chapter 5 explains the design of the developed DSL. The implementation of our work is outlined in Chapter 6. Our evaluation of this work is described in Chapter 7, which demonstrates the visual exploration of bank conflicts and branch divergence. Finally, in Chapter 8 conclusions are drawn and the thesis completes with an outlook of possible future work.

CHAPTER 2

# Fundamentals

In order to provide a contextual overview for the subsequent chapters, several core concepts of parallel computing are described in the following. Starting with a brief explanation of OpenCL basics, important considerations regarding the performance of parallel programs are following.

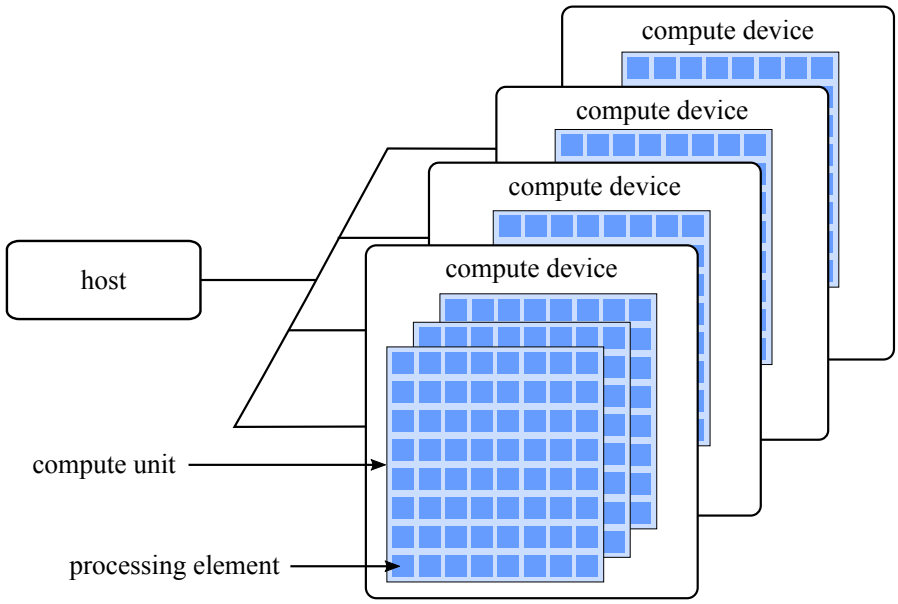



*Figure 2.1: The OpenCL platform model consists of a host application that controls various compute devices. A compute device consists of numerous compute units that are divided into processing elements [Gro].*

## 2.1 Parallel Computing and the OpenCL Terminology

In contrast to traditional sequential computing, where every computation is executed consecutively, parallel computing provides vast performance boosts through parallel execution. It is extensively used in the area of visual computing and simulation, because many tasks naturally lend themselves to parallel execution, e.g. molecular dynamics, image processing, climate modeling and traffic simulation. Additionally, due to the breakthrough of the GPU, driven by the video game industry [NHKM14], and the breakdown of the graphics hardware pipeline into programmable components, which restricts the GPU no longer to graphical tasks, parallel computing has become a wide-spread paradigm for many scientific applications. Furthermore, as a result of the recent progress of platforms like CUDA and OpenCL, easy access to the power of the GPU for parallel computation is facilitated.

Within this work, the OpenCL standard is used to execute parallel programs, due to its platform independency and just-in-time compilation capabilities. OpenCL is an open, royalty-free, cross-platform standard. OpenCL exposes its core API in C, yet there are various wrappers that also enable access in different languages. Parallel programming is realized with the OpenCL C language, which utilizes a subset of ISO C99 with additional extensions for parallelism.

We start with an outline of the different concepts and models of OpenCL. Note, that due to the complexity of OpenCL, we only explain context that is directly related to this work. For a comprehensive description of OpenCL we refer to the OpenCL specification [Gro]. A broader introduction to parallel computing is found in the book ”Programming Massively Parallel Processors [KWm12]. An introduction to OpenCL is provided by the OpenCL Programming Guide [MGMG11].

### 2.1.1 Platform Model

A platform model describes how the hardware is abstracted to be comprehensible by the user. The OpenCL platform model consists of a host, which is connected to one or more compute devices, as depicted in Figure 2.1. The host is responsible for the management of data and the execution of parallel programs on those devices. Host and devices have separated address spaces, and memory transfer between the two is explicitly invoked by the host. A compute device consists of multiple compute units. Within each compute unit there are several processing elements.

The specific mapping of this model depends on the deployed hardware. Various devices support OpenCL [Khr], the majority consists of CPUs and GPUs. For instance, CPUs of common workstations are frequently used as host, whereas a compute device optimally has a large number of parallel processors. Due to this reason, the GPU is often used as a target device. In the case of NVIDIA GPUs, the compute unit is realized as so-called ”*streaming multiprocessors*“, where each multiprocessor is of a SIMD (single instruction, multiple data) architecture.

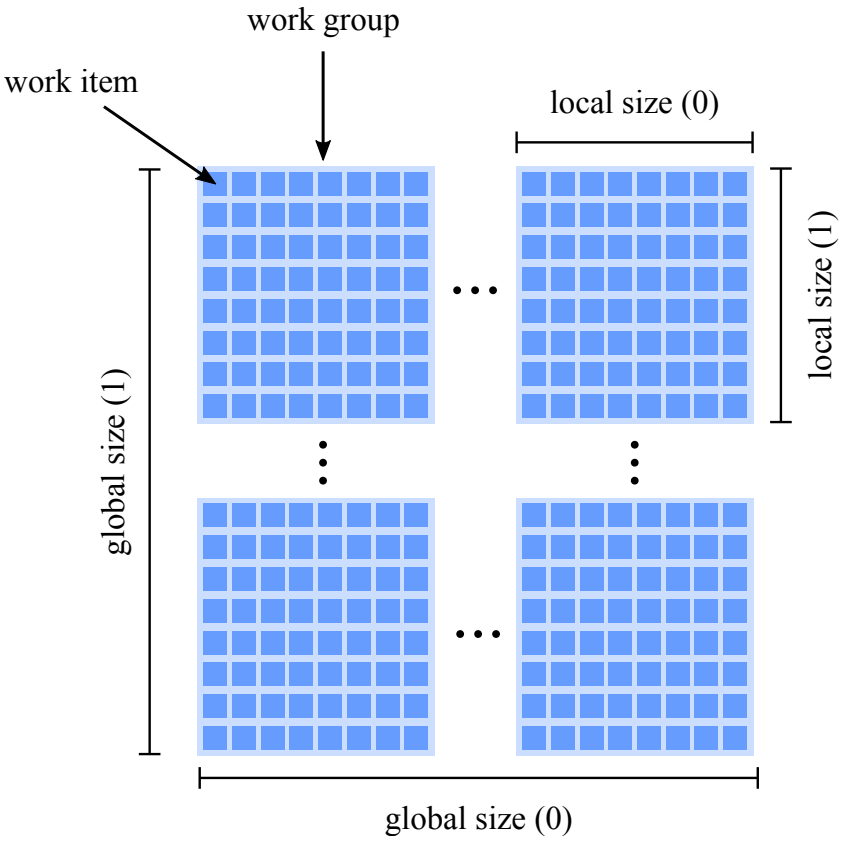



*Figure 2.2: OpenCL execution model [Gro].*

### 2.1.2 Execution Model

OpenCL applications are mainly divided into two parts, namely kernels, which are programs that are executed in parallel on OpenCL compute devices, and the host program. The host program is responsible for starting and managing kernels as well as their scheduling and memory management. It also creates the connection to the devices and commands execution of kernels. In OpenCL, an instance of a kernel program is called a work item. Work items are organized into work groups, as depicted in Figure 2.2. A work group consists of several work items, which are uniquely identified by either the local id and the work group id, or by the global id. The local id is unique within a work group.

This execution model supports data-parallel as well as task-parallel programming models. Data parallelism is typically mapped onto compute units, while task parallelism can be mapped onto compute units or compute devices, as well as be split between host an devices. In this work, we are solely focused on data parallelism, since this is the more important paradigm in GPU compute systems.

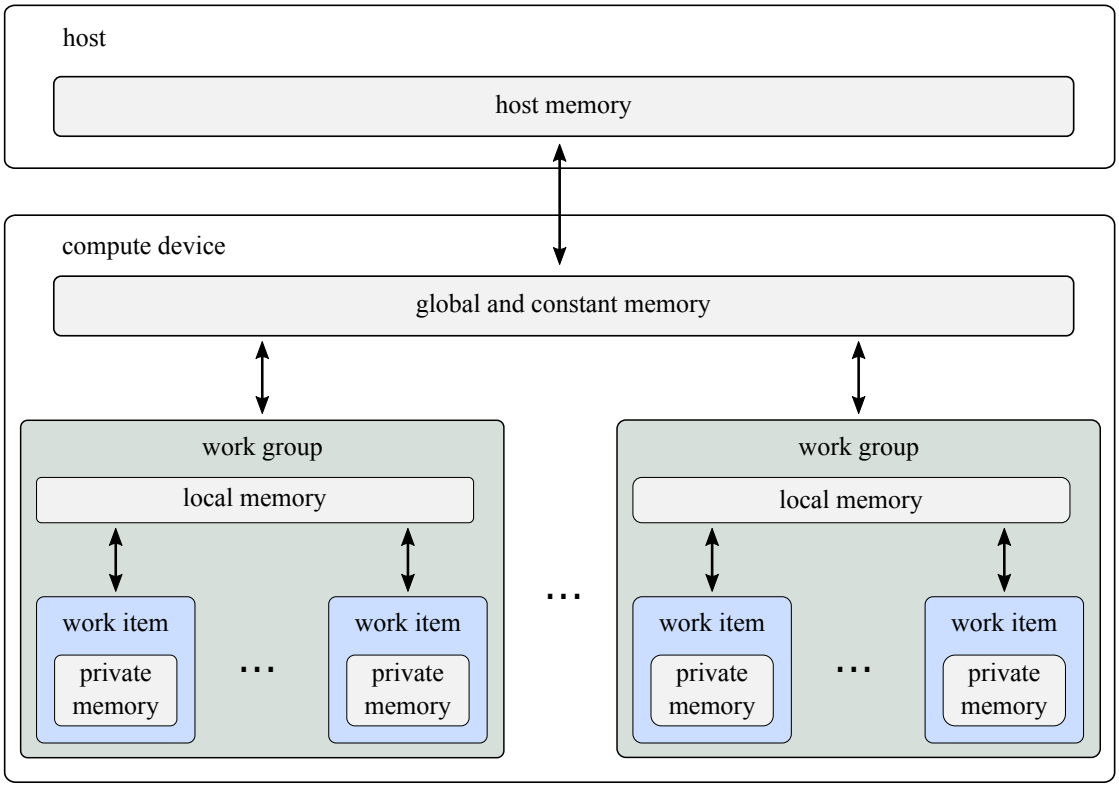



*Figure 2.3: The OpenCL memory model illustrates the different memory regions within the host and the compute devices [Gro].*

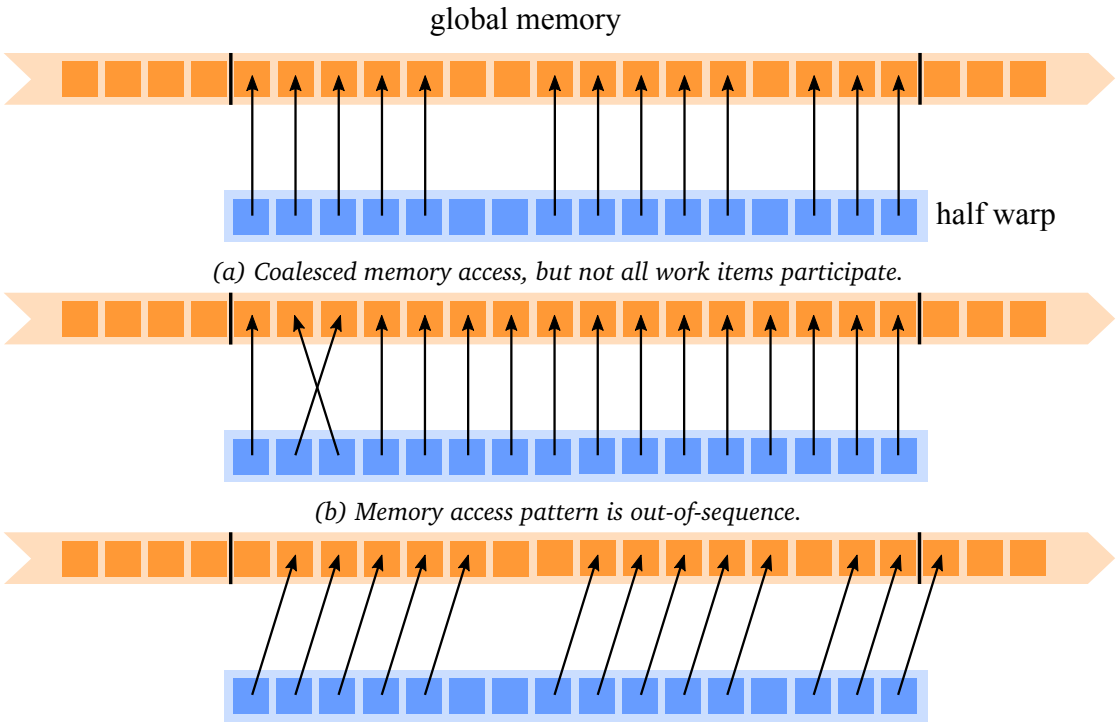
### 2.1.3 Memory Model

The Khronos Group developed the memory model in a way that closely resembles the one of current GPU hardware, although this does not limit the adoptability by other devices. The memory model, as shown in Figure 2.3, consists of distinct memory regions that are specific to certain execution elements. Host memory is visible from the host only, but not further defined by OpenCL. To make data on the host accessible on OpenCL devices, it has to be transferred through the OpenCL API. Unlike host memory, global and constant memory as well as private memory are located on the compute device. The private memory region is accessible per work item. Local memory is the region of memory that is local to the corresponding work group. Global memory can be accessed by any work group of the compute device.

In a typical setup that uses a GPU as compute device, data is transferred between host and global memory on the compute device. The transfer is handled over the PCIe bus. This bus sets the upper limit for data transfer speed between host and compute device, but is generally not the limiting factor of parallel computations.

## 2.2 Performance and Profiling

The performance of parallel programs is largely dependent on three factors. First, the utilization of the hardware, which includes optimal memory transfer between host and compute devices, as well the exploitation of processor capabilities. Second, the memory throughput, which is significantly influenced by the way memory is accessed and copied between the different regions of memory. Finally, the performance of parallel programs is dependent on the instruction throughput. Parallel computing is based on concurrency. Diverging control flows for work items that are executed in parallel contradict this model and cause slowdowns. In the following, two key topics are reviewed that are typical candidates for investigation in the context of performance profiling.




*(a) Coalesced memory access, but not all work items participate.*

*(b) Memory access pattern is out-of-sequence.*

*(c) Sequential but misaligned memory access.*

*Figure 2.4: The illustration shows three different memory access patterns for global memory.*

### 2.2.1 Memory Access

The memory access of local and global memory demands special attention. Inefficient access patterns are often the main cause for a significant slowdown in performance.

#### Coalesced Global Memory Access

One of the most important considerations for performance is coalesced access of global

memory. This means that parallel running work items benefit vastly from accessing nearby memory. A compute device is capable of returning whole blocks of global memory at once if they are accessed simultaneously. The exact circumstances and the size of global memory that a compute device can return in one block depends on the hardware implementation. In the following, memory access patterns for NVIDIA GPUs are reviewed, which only differ for the different compute capabilities. The term *compute capability* is introduced by NVIDIA to classify their hardware into individual groups, each with consistent support for features.

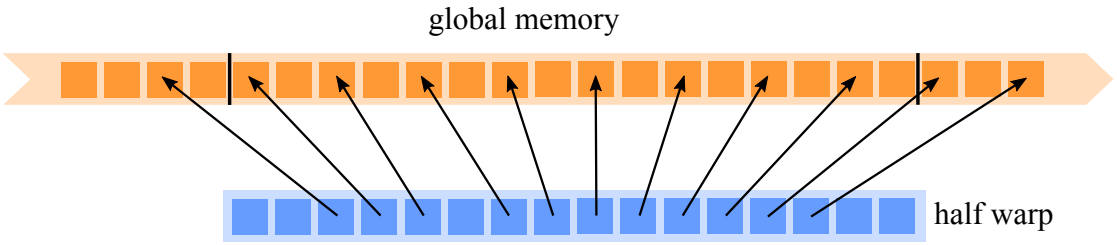



*Figure 2.5: Strided memory access.*

The higher the compute capability, the more relaxed coalesced memory access is defined. In general, memory accesses are based on half warps (16 work items). On NVIDIA hardware, code is actually executed in batches of 32 work items, which is called a warp. NVIDIA [NVId] illustrates various access patterns for coalesced and non-coalesced accesses. For compute capability 1.0 and 1.1 every $k$-th work item of a half warp has to access the $k$-th word in a memory segment to achieve coalescing. However, not every work item has to participate. Figure 2.4a shows such an coalesced access pattern. This example would result in either one or two memory transactions, as explained in the following. Memory segments are transferred in the size of 32-, 64-, or 128-byte blocks per transaction. In case the accessed word size is 4-, or 8-byte, the example of Figure 2.4a results in a single transaction. In the case of 16-byte addressing, two transactions are executed with 128-byte segments. The example shown in Figure 2.4b has a memory pattern that is out-of-sequence. This results in 16 separate 32-byte memory transactions and a corresponding decrease of performance. Figure 2.4c illustrates misaligned memory access of two different segments, hence the memory access is not coalesced. Devices with compute capability 1.2 and higher can access words in any order. Each addressed segment in a half-warp results in one memory transaction. However, even in compute capabilities above 1.2, efficient memory access is important for optimization. For instance, a common appearance in parallel computing is strided access of memory, as shown in Figure 2.5. Even though memory access is coalesced, the performance is still compromised since only half of the transferred memory is actually used for computation.

### Bank Conflicts in Local Memory

Local memory is on-chip memory and therefore much faster than global memory. Note, that the term *local memory* has a different meaning in the CUDA specification. CUDA uses the term *shared memory* for OpenCL's *local memory*. Local memory is equally divided into

| Bank | 1 | 2 | 3 | ... |
|---|---|---|---|---|
| Address | 0, 1, 2, 3 | 4, 5, 6, 7 | 8, 9, 10, 11 | ... |
| Address | 128, 129, 130, 131 | 132, 133, 134, 135 | 136, 137, 138, 139 | ... |
| ... | ... | ... | ... | ... |

*Table 2.1: Illustration of local memory byte addresses with a setup of 32 memory banks.*

multiple modules called memory banks. Devices with compute capability 1.x have 16 memory banks and devices with compute capability 2.x have 32 banks. Successive 32-bit words in local memory are assigned to successive memory banks. An illustration of local memory byte addresses with 32 memory banks is shown in Table 2.1.

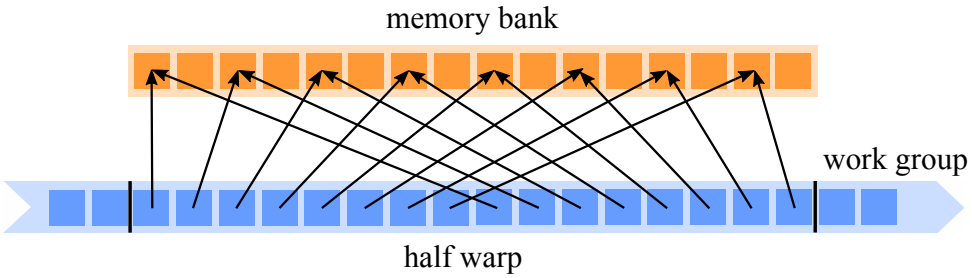



*Figure 2.6: A 2-way bank conflict on local memory with a setup of 16 memory banks.*

Different memory banks can be accessed simultaneously. However, if multiple work items attempt to access the same memory bank at the same time, a bank conflict arises and the access is serialized. The only exception to this rule is a broadcast. In case all of the work items in a half warp access the same address within a memory bank a broadcast happens and no bank conflict occurs. Figure 2.6 shows a common example that uses a strided access pattern. Since memory banks are accessed by more than one work item, bank conflicts occur. This incident is known as a *2-way* bank conflict.

### 2.2.2 Control Flow and Synchronization

Using efficient memory access patterns is important to achieve a high memory throughput. However, also the instruction throughput plays a considerable role in performance. The main subjects in this context are control flow and synchronization.

#### Control Flow

Control flow in parallel programs plays a significant role regarding performance. Parallel computation in OpenCL follows the SIMD architecture of the GPU. Several processing elements perform the same instruction on multiple data elements. In case different work

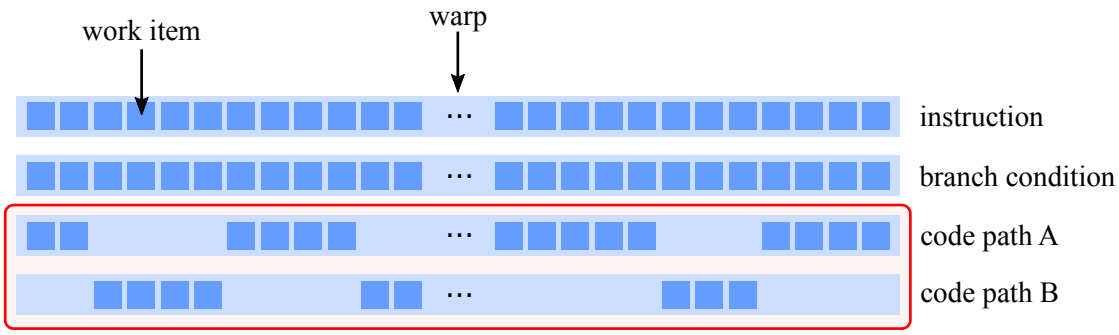



*Figure 2.7: This illustration shows the execution of code for the case of a work item divergence.*

items need to execute different instructions, a so-called work item divergence can occur. Work item divergence only plays a role within a warp. In case these work items execute different branches of conditionals, all work items have to execute both branches (see Figure 2.7). The execution cost in case of a branch divergence increases to the sum of the different branches.

### Synchronization

In case the order of the execution influences the outcome of the program, data dependency occurs. When a work item writes to a memory location of local or global memory, it is not guaranteed that the memory is updated if another work item addresses this memory location. This can result in unpredictable behavior and is called out-of-order execution, and leads to the necessity of memory synchronization. The OpenCL C language supports local and global barriers, which are responsible for the synchronization of local and global memory, respectively. A barrier statement in a program forces all work items of a common work group to wait, until all of them encounter this statement. Afterwards, their parallel execution continues [Gro]. When a barrier is passed, all write-accesses to the memory of the corresponding work group are guaranteed to have executed. Obviously, barriers can slow down the performance of a program dramatically, since work items are temporarily stalled. Synchronization of memory between work groups within the kernel is not possible. It can only be realized on kernel-level through the termination of kernel programs, as shown in Chapter 7.3 by means of parallel reduction. The consecutive steps of the reductions are synchronized with several invocations of the same kernel.

If parallel programs are not properly synchronized, data races can occur. A data race happens if two work items access the same memory location and at least one of them writes to this location. The user can resolve this issue by inserting a barrier in between the two operations, otherwise the behavior is undefined. The OpenCL specification [Gro] states that in case a barrier occurs in a conditional statement or in a loop, all work items must encounter the barrier before any is allowed to continue execution. However, GPUVerify [BCD+12] shows that in practice, programs can be implemented with barrier divergence resulting in different results depending on the GPU architecture. Barrier divergence describes a behavior where work items of the same work group diverge and reach different barriers. This illustrates

the issues that come along with synchronization of parallel programs. Aside from corrupt results that can be produced with wrongly placed barriers, parallel programs can often be implemented in different ways requiring more or less memory synchronization, which strongly effects performance. Chapter 7 shows examples of the effect of different implementations.

CHAPTER 3

# Related Work

The first part of this chapter reviews work that covers visual exploration of parallel programs. Visualizations are frequently used to depict the behavior of software and to display collected debug or profiling information. In general, the visualization of information that is related to the runtime behavior of systems or applications is called software visualization [GME05]. Software visualization utilizes techniques like tracing and recording, or measuring techniques to gather information, which is subsequently presented to the user through visual representations. The second part of this chapter briefly discusses the current state of the art in domain-specific languages associated with parallel computing.

## 3.1 Visual Exploration of Parallel Programs

There is as wide range of applications for visualizations in the context of parallel programs. The main reasons to apply visualizations are the support in understanding the behavior of the program, the provision of profiling information, and the detection of erroneous code. Visualizations facilitate a rapid insight into the behavior through visual representations of the hardware. The behavior of parallel programs is significantly more complex than that of their sequential counterparts.

A broad overview of the visualization of parallel programs is presented in the book "Visualization of Scientific Parallel Programs" [TU94]. It does not refer to current frameworks, such as CUDA or OpenCL, but provides a valuable outline of the major challenges occurring during the exploration of parallel programs. The challenges outlined in the book are:

- **Increased Complexity:** Concurrency adds an additional level of complexity to the program in comparison to sequential programs.
- **Nondeterminism:** Programs can become nondeterministic and may show non-reproducible behavior, due to data races.

- **Probe Effect:** Observing the parallel program often leads to additional instructions in the code, in case the observation is not integrated in hardware. This can influence the program and may change the outcome [Gai86].
- **Lack of a Synchronized Global Clock:** Difficulties arise in reproducing the exact order of events, because of the missing of a global clock system.

Furthermore, the book outlines several approaches to parallel debugging. The following points are not only applicable for debugging, but also for profiling and the provision of a general understanding.

- **Extending Traditional Debugging:** Fundamental concepts of traditional debugging, e.g. exceptions, can be extended to parallel programs.
- **Event Histories:** Event histories or trace files are responsible for logging information about the program's execution. After the termination, they can be utilized for the investigation of anomalies.
- **Use of Graphics:** The benefit of visualizing information for a better understanding has become evident, especially in the visual analytics community. Collected information can be visualized through simple textual representations up to complex interactive renderings.
- **Static Analysis:** Certain anomalies can be detected automatically and do require visualizations, such as synchronization errors and data-usage errors (e.g. reading of uninitialized variables).

There are several tools, presented in the following, which exploit these points in different ways. In our context, software visualization is divided into two categories:

- **Static Software Visualization:** Static software visualization refers to visual representation of software before it executes. Related data is usually acquired through the analysis of the source code. Most of the approaches that fall into this category make only very limited use of visualizations. The acquiring of data is realized through the analysis of the program's source code. Tools like GPUVerify [BCD+12] formally analyze OpenCL and CUDA kernels in order to prove their correctness. The correctness is shown through the detection of issues like data races, which make the execution non-deterministic.
- **Dynamic Software Visualizations:** During the execution of a program, it runs through a complex process of interaction with the hardware, which builds the basis for related visualizations. Visual representations of dynamic software basically depicts the behavior of the program in a broad sense. This includes visualizations of memory accesses, communication between different parallel units, and control flow of the execution.

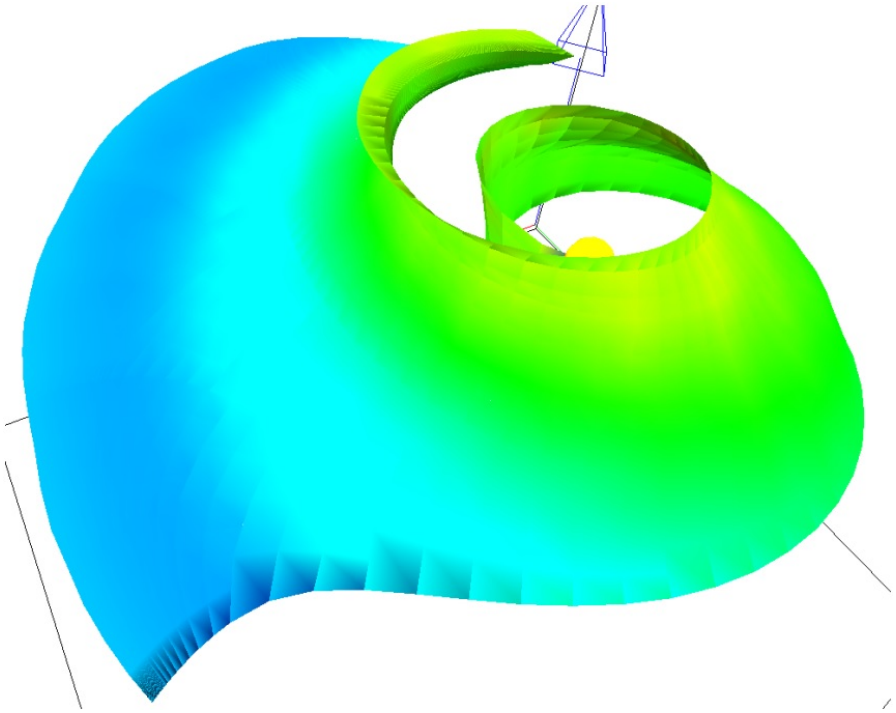


*Figure 3.1: The visualization of the surface normals indicates errors that appear on the edges of the above shown surface [CE99].*

### 3.1.1 Visual Exploration for Understanding and Debugging

A common concept for visualization software, is to encode debugging information within the already existing structures. Figure 3.1 illustrates a geometry with color encoded surface normals. Errors become clearly visible on the edges of the surface. A case study [CE99] indicates that conventional debugging tools are inadequate and that visualizations support the user to quickly grasp errors that would take hours to find otherwise. They state that due to the capabilities of the human eye, debugging visualizations immediately reveal patterns or mistakes indicating wrong behavior of the program.

One of the issues in debugging parallel programs, is the lack in control from the compute-device side. In current GPU hardware, the only way to hand control over to the CPU, is the termination of the kernel. Due to this reason there are several approaches that tackle this issue. Hou et al. [HZG09] present a GPU interrupt mechanism, which is used for the debugging of GPU stream programs. Data flow instructions are automatically added to a program in order to detect memory errors and to trace errors back through related work items. The GPU interrupt mechanism facilitates the calling of CPU functions from within the kernel, thus data flow recordings can be directly sent to CPU memory. The interrupt mechanism is realized by a compiler that translates the interrupt call into a kernel termination. After the kernel termination, the CPU takes control and executes the required functions. Subsequently, the

kernel is re-launched and the processor context is set to the preceding state. This approach has certain limitations, such that processor states like shared memory and barrier synchronization are not supported. Additionally, the approach implements a dataflow visualizer, which, however, only provides rudimentary visualization metaphors, such as the encoding of data into circles with varying color and size.

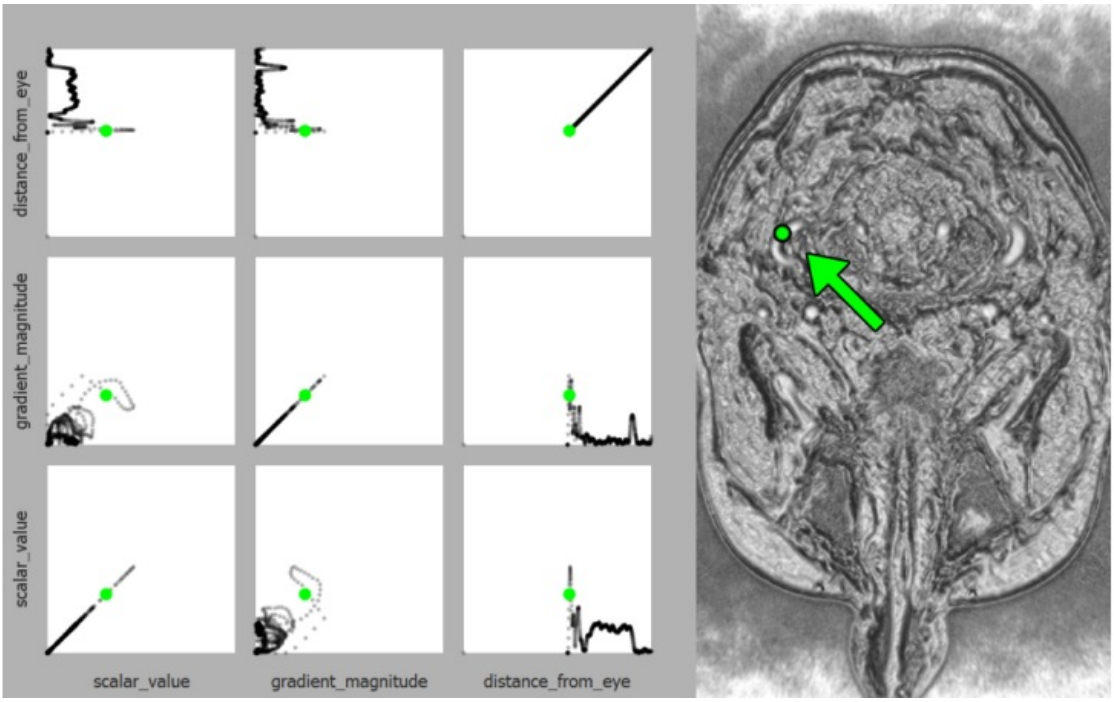



*Figure 3.2: Visual Exploration of a parameter space through scatterplot matrices of feature vectors [MBRG13].*

### 3.1.2 Visual Exploration for Profiling

Profiling parallel applications is usually an experimental multi-objective task. Several constraints have to be balanced to achieve a suitable setup for maximum performance. The purpose of visual exploration for profiling is the provision of visual representations that support this process. A comprehensive state of the art report about performance visualization is presented by Isaacs et al. [IGJ+14]. The works reviewed in the following are categorized by the method that is used for data extraction:

#### Instrumentation-based Methods

Mindek et al. [MBRG13] propose a technique for the visual exploration of parameter space in GPU programs. They consider a GPU program for visualization as a mapping between the data and a visual representation controlled by parameters. The set of possible combinations of values of these parameters build the parameter space. The usual way of analyzing visual-

ization algorithms and the effect of parameter variations is to examine the resulting image. In this case, the parameter space itself is visualized to enable visual exploration. For this reason, a domain-specific language is introduced that can be added into the existing shader program in the form of code annotations. The language is capable of extracting feature vectors that consist of parameters or data attributes used in the shader. Feature vectors are then visualized in scatterplot matrices to provide an overview of their relationships, as shown in Figure 3.2. This figure provides a linked view between the rendered image (right side) and the scatterplot matrices of the feature vectors (left side). The green dot in the rendered image depicts one pixel and is linked to the corresponding feature vectors in the scatterplots, also depicted as green dots.

### Interception-based Methods

The work of Strengert et al. [SKE07] implements a system to analyze shader code written in the OpenGL Shading Language. It intercepts OpenGL calls as they are invoked and replaces the original shader program with an instrumented one. Instrumentation of the program facilitates the inspection of variables at arbitrary code positions and stores information about loops, conditionals and function calls. A debug environment allows the user to inspect any OpenGL call and step through the associated vertex, fragment or geometry shader code.

### Emulation-based Methods

The work by Rosen [Ros13] is closely related to our approach. It visually investigates the behavior of shared and global memory. The approach provides a hierarchical view on memory access. The topmost stage provides an overview of all work items, structured in separate work groups (see Figure 3.3 (left)). The view is broken down into a visualization of separate warps, as shown in Figure 3.3 (middle), where the lowest level of the view hierarchy shows memory access within a single warp. The color encoding in Figure 3.3 (right) highlights differences in the warps specified by a metric. For instance, the three blocks (label C) in the warp view indicate difference in active work items, shared memory accesses, and global memory accesses. However, the approach is not based on a real execution of CUDA code but on simulation with GPU Ocelot [DKK09], a binary translation framework.

### Miscellaneous Methods

Isaacs et al. [IBJ+14] investigate event traces of large-scale parallel programs. They propose the utilization of a timeline view for the visualization of parallel execution traces. Logical steps are represented by blocks and lines indicate communication between them. Instead of using the physical time for ordering blocks in the timeline, they propose to use a concept called logical time, which corresponds more closely to the code structure. The logical time represents before and after relationships instead of exact timings. This is similar to our approach for the sorting of memory access events, as described in Chapter 4.2.1.

A novel visualization for cache behavior is proposed by Choudhury et al. [CR11]. They

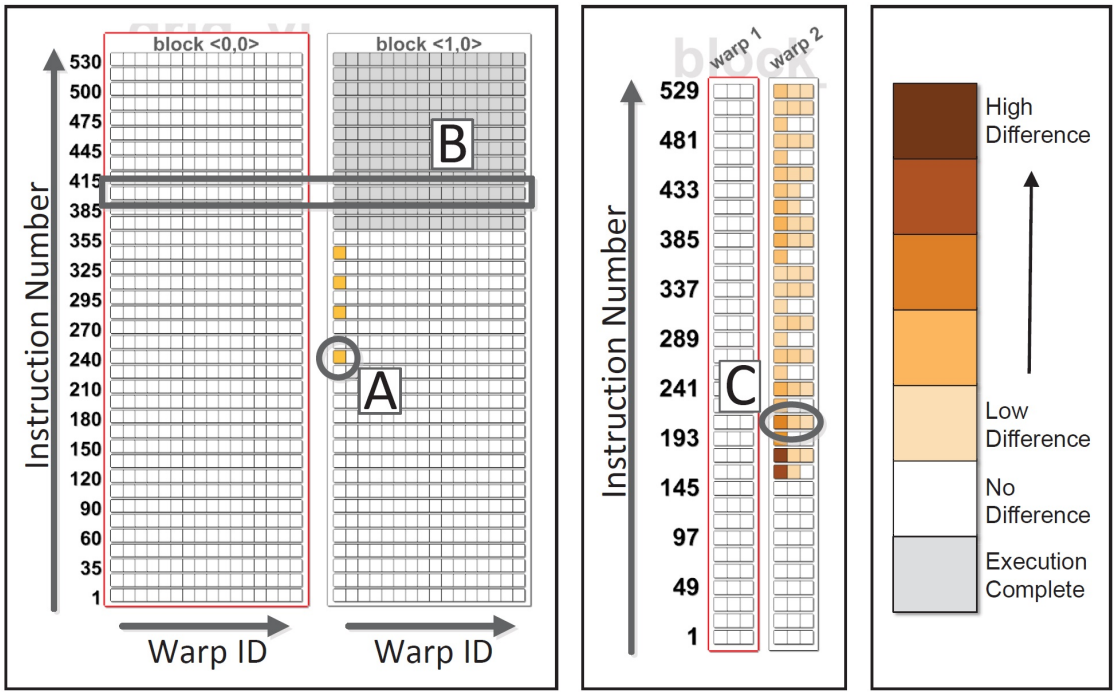



*Figure 3.3: The overview on the left side shows the analysis of memory accesses in a parallel program for all work items, which are structured in separate work groups. The middle part breaks this view down to the level of separate warps. Each row in the warp section consists of three blocks that show the result of different metrics, which indicate the amount of difference within a warp [Ros13].*

map different cache hierarchies to a circular view, as depicted in Figure 3.4. An increase in distance of the cache to the processor is encoded accordingly in the view with an increasing distance to the center. The lines represent cache accesses. The three views in the figure illustrates the behavior of a bubble sort algorithm. Bubble sort continuously sweeps over a work set and swaps the remaining largest element to the correct location. The sweeps become progressively shorter and caching improves, which is reflected in the images.

Moreover, there are a number of commercial tools, such as NVIDIA Nsight [NVIc], AMD's gDEBugger [AMDb] and its successor CodeXL [AMDa].

## 3.2 Domain-Specific Languages for Visual Computing

The definition of a domain-specific language (DSL) is rather vague, as it leads to the question of what constitutes a domain. Some languages may offer features for a particular domain in addition to general language features. The general consensus is that a DSL incorporates particular domain knowledge into the language in contrast to a general-purpose language (e.g. C, Java or Python), which is supposed to be applicable for many domains. According

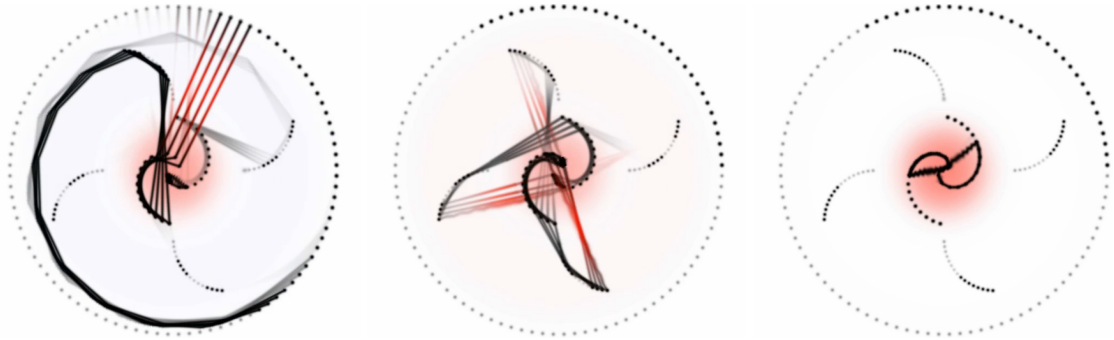


*Figure 3.4: The illustrations depict cache hierachies and corresponding accesses. An increasing distance to the center indicates an increasing distance of the cache to the processor. A bubble sort algorithm applies progressive sweeps through a list of elements to be sorted and swaps the remaining largest element to the correct location. Since the sweeps continuously become shorter, caching of the work set progressively improves, as indicated in the above image [CR11].*

to this definition, even low-level shader-languages, such as Cg or GLSL, are regarded as DSLs, since they incorporate the knowledge of the graphics pipeline. This shows that the line between general-purpose languages and DSLs is blurred. In this work, low-level languages for parallel programming, eg. OpenCL and CUDA, are regarded as general-purpose languages, even though they are explicitly developed for the parallel programming domain. Languages that incorporate domain knowledge beyond this point are considered DSLs.

A comprehensive overview of the general development of DSLs is provided by Lengauer et al. [BDO04] and by Fowler [Fow10]. The latter classifies the context of DSLs into three categories: internal DSLs, external DSLs and language workbenches.

- **Internal DSL:** Internal DSLs, also called embedded DSLs, use the syntax of their host language while adding domain-specific data structures, methods or routines. They are similar to a library of this language, although they differ in interface style. The interface of the DSL is more fluent and uses local domain definitions. An examples is a domain-specific xml scheme.
- **External DSL:** An external DSL is defined in an own language, uses its own syntax and semantics and also its own parser.
- **Language Workbench:** Language workbenches basically consist of toolsets for DSLs. They support designers in the creation of a parser and offer features like text editors with syntax highlighting.

There are many occurences of variations and hybrid styles of these groups. For instance, ViSlang [RBGH14] can be regarded as a language workbench even though it also provides its own DSL. It supports the implementation of new DSLs via a plugin mechanism. DSL plugins are integrated into the existing language. A special keyword in the ViSlang code (”using“)

followed by the name of the DSL, indicates that the remaining code is parsed and executed through the plugin until the keyword is used again. ViSlang is used for the creation of our DSL.

*Algorithm 3.1: Halide code example [RKAP+12].*

```
Func halide_blur(Func in)
{
    Func tmp, blurred;
    Var x, y, xi, yi
    // the algorithm
    tmp(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;
    blurred(x, y) = (tmp(x, y-1) + tmp(x, y) + tmp(x, y+1))/3;
    // the schedule
    blurred.tile(x, y, xi, yi, 256, 32);
           .vectorize(xi, 8).parallel(y);
    tmp.chunk(x).vectorize(x, 8);
    return blurred;
}
```

There are several DSLs for parallel computing focusing on various objectives. For instance, the DSL Halide [RKAP+12, RKBA+13] introduces a system to decouple algorithms from schedules. A schedule describes the locality and time dimension of an algorithm. This includes memory layouts, execution order and caching structures. This separation facilitates an explicit description of the algorithm logic, but renders it independent of its schedule. Moreover, the user can experiment with different schedules to pursue performance improvements without changing the algorithm logic. Stochastic methods also facilitate automatic search for efficient schedules. Algorithm 3.1 shows a simple code example of a $3 \times 3$ box filter written in the Halide language. The difference between algorithm and schedule is clearly visible. The first part implements the logic of the blur function, whereas the second part defines the schedule. The functions tile, parallel, vectorize and chunk are individual specifications of the schedule.

In contrast to Halide, Vivaldi [CCQ+14] emphasizes flexibility and usability rather than the improvement of rendering quality and pure performance. It is a python-based language, which hides the details of memory management and data communication. It is influenced by Shadie [HWCP10], a domain-specific language for visualization that also uses a python-like syntax. Vivaldi targets volume processing and visualization on cluster architectures of heterogeneous computing nodes and attempts to simplify their usage. Furthermore, it implements library functions to provide a number of tools for the visualization domain, such as neighborhood iterators to efficiently access neighborhood values and halo communicators for accessing neighborhood voxels.

The domain-specific language Diderot [CKR+12] specializes on the particular context of continuous tensor fields in order to support programs for biomedical image analysis and visualization. It adopts the mathematical notation of vector and tensor calculus for programming, and deploys a C-like syntax for computational notion. A Diderot program consists of global definitions, an initialization part and a so-called strand. A strand represents

the parallel part of the program like kernels do in CUDA or OpenCL.

CHAPTER 4

# Visualization of Parallel Programs

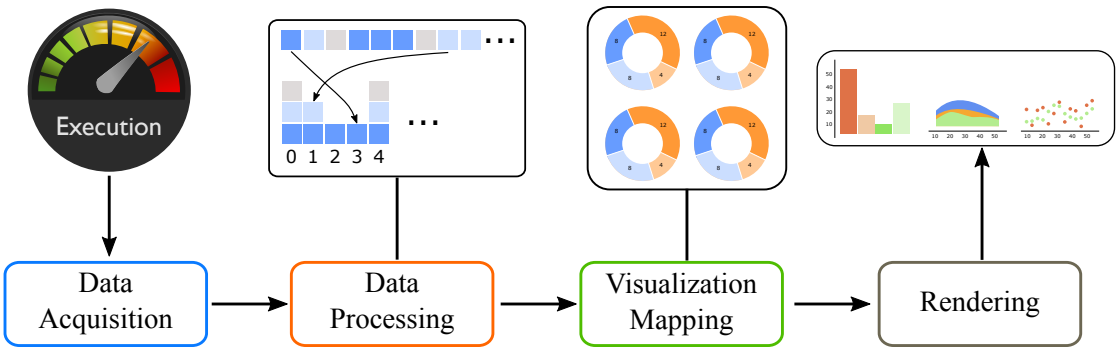



*Figure 4.1: The visualization rendering pipeline in relation to our visual exploration approach of parallel programs.*

The essential approach of visualizing parallel programs is closely related to the standard visualization pipeline, as depicted in Figure 4.1. At first, data is acquired during the execution of the program. It contains information, such as memory accesses, the branching of a program, and work item communication. During the next step, the resulting data is processed for the transformation into a meaningful representation as preparation for the following visualizations. Subsequently, the data is mapped onto visual representations before it is finally rendered. The rendering part is realized by the web browser, which renders canvas or svg elements.

### Detail Level

We divide the investigation of parallel programs into three different levels:

- **Application-level:** Provides a broad overview of the program's execution. It often

consists of timeline views [NVIf], which depict activities for both, host and compute device side. For instance, it facilitates the investigation of kernel runtimes, and memory transfers between host and compute devices.

- **Kernel-level:** Analysis on the kernel-level offers information about the inner behavior of individual kernels. Usually, statistics about occupancy, memory traffic and pipe utilization are shown. A typical tool for this task is NVIDIA Nsight [NVIc].

- **Source-level:** The source-level analysis drills further down to the source code of kernels. It provides detailed information about, e.g. memory transactions, including its backtracing to instructions, statistics about branch divergence, as well as information about the execution number of instructions [NVIc].

The above mentioned abstraction levels are usually deployed in the named order, to start with an overview and then proceed with details on demand. After the issue is roughly localized, kernel- and source-level analysis are utilized to further narrow it down. In the course of this work, the focus is put on source-level analysis and its corresponding visualizations. However, in contrast to other tools, the presented framework puts emphasize on the "*why* something goes wrong" instead of "*where*". In some cases, we assume that the kernel, or part of the kernel that is supposed to be investigated, is already known.

### Data Types

Parallel processors usually provide special dedicated hardware for performance investigation. They appear in the form of registers that are used to record data about processor events over time. These hardware counters store events like cache misses, number of floating-point operations and number of memory accesses. More elaborate hardware can also store hardware samples, which contains additional information, such as the corresponding function pointer to an instruction instead of simply aggregating counts. NVIDIA grants access to their hardware counters with their CUPTI API [NVIb], which many profiling tools use for data acquisition. The usage of such counters benefits from the hardware implementation though limits the analysis to general statistics.

Traces facilitate a more comprehensive analysis of the execution though suffer from additional code that has to be added to record the data. A trace is a log of events, which are recorded during the execution of a program. Examples for events are memory accesses, conditionals, and procedure entry or exit. In contrast to hardware samples, traces can store arbitrary additional information that belongs to the context of the event, such as the current state of the program, the line of the source code it origins from, or the current timing. In order to realize tracing, additional code has to be added to the original program. This is usually done with instrumentation techniques either applied to the binaries or to the source code. Adding code to the program introduces a computational overhead and makes it susceptible to the probe effect. However, keeping the additional code minimal, reduces these risks, and the investigation of the program is not performance critical.

The plain code also provides information that is used for the investigation of the program. It contains the structure of the program, and its semantics enable later inferences from collected data back to the code. Furthermore, the static analysis of the code facilitates data independent investigation, that reveal possible defects like data races [BCD+12].

### Purpose

Visual exploration of parallel programs facilities various purposes, where the most common is usually profiling. It enables the revealing of performance bottlenecks along with the detection of possible causes. Performance optimization of parallel programs is typically characterized with the following measures:

- **Utilization:** Exploiting parallelism on application-level increases the overall utilization of the hardware. The memory transfer between host and device and the idle time of compute devices has substantial influence on performance, and benefits greatly from parallelism.
- **Memory throughput:** Non-coalesced memory patterns, as well as the sheer number of memory accesses influences the memory throughput. Avoiding bank conflicts and optimizing these patterns increases the performance.
- **Instruction throughput:** Any control flow instruction can significantly affect the performance by causing branch divergence. By minimizing control flow and synchronization the instruction throughput increases.

The goal is to utilize visualizations in order to assist the user to pursue the above mentioned objectives. Nevertheless, also debugging purposes are supported by visual exploration. Offering tools for simple extraction of data, connected to suitable visualizations, might help to expose anomalies and helps in the comprehension of a program. Visualizing the execution of the program in a suitable way supports the understanding of the code and ideas for optimization.

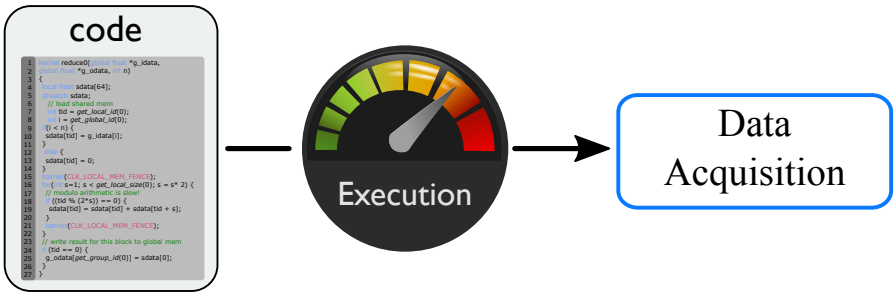



*Figure 4.2: Data for the visual exploration of parallel programs is acquired either from the plain source code, or using recordings during the execution of the program.*

## 4.1 Data Acquisition

Acquisition of data for the visual exploration of parallel programs has basically two different sources (see Figure 4.2). First, the raw code of the program itself. It is often analyzed to infer its structure and various static information as mentioned above. Second, the execution of the program, during which data is recorded. These recordings are either data from hardware counters or from data traces. The acquisition of data from hardware counters is realized with dedicated APIs, which are usually provided by the hardware vendors. A common way of generating traces is by manual code insertion by the user. This is useful for special cases where the user demands uncommon data or simply wants to investigate parts of his own data structures. However, in general it is cumbersome and defeats the purpose of a quick investigation. Code instrumentation simplifies this task by automatically modifying the program and inserting the necessary code. It is applied on either the source code before compiling, or applied on the binaries after the compilation.

A major challenge in data acquisition of parallel programs is the one-sided communication between host and devices. Once a device starts its dedicated task, there is no communication until the task is finished. That means the results of a kernel can only be read back after its termination. Current graphics drivers do not allow communication from the GPU to the CPU. Collected data is hence transferred after termination, which limits the amount of data available for one kernel run to the available device memory. In order to circumvent this issue, Hou et al. [HZG09] present a mechanism to interrupt a kernel and to call the CPU from inside the kernel. This enables the direct sending of data to the host, although the algorithm is limited to save the state of the processor. After the interruption, the processor state has to be fully restored in order to ensure correct results. For instance, features such as the restoration of local memory and synchronization barriers are not supported. Stuart el al. [SCO11] present an approach, called GPU-to-CPU callbacks, for the communication from GPU to CPU. This approach uses zero-copy memory to realize the callbacks. Zero-copy memory directly maps host memory to GPU memory and enables its access from the GPU. However, depending on the architecture and the implementation by the vendor, this can result in expensive data reads over PCIe.

To overcome the drawbacks of the above mentioned approaches we utilize device memory and store data in global memory. The apparent disadvantage of this method is that we are bound to the device memory size. Furthermore, since OpenCL does not allow dynamic memory allocation, the required memory has be determined beforehand. This sets an upper limit on the traces. Many profiling tools deal with this issue by invoking the kernel multiple times. In the following we present the recordings and tracing methods of our framework.

### 4.1.1 Manual Recording

Manual recording requires the user to change the code, which is tedious but a reasonable option for extracting arbitrary data values that are not covered by our provided concepts. It is common practice to manually write debug data into buffers and send it to the host for inspection. We shorten this approach by offering simple code annotations that automatically

deal with memory allocation and data transfer to our visual explorer. Furthermore, our approach benefits from fast turnaround times. It is often critical to change code and quickly see its results. Long build times and setups contradict this principle. A typical OpenCL application has to compile the OpenCL API code (C code) in addition to the kernel. Our approach solely requires the re-compilation of the OpenCL kernel, in case the user makes changes. This significantly improves turnaround times.

### 4.1.2 Memory Access Traces

As mentioned before, the investigation of memory accesses is one of the most important steps for profiling. In our approach, we instrument every memory access in the code to acquire the concerning data during the execution. This collection is configured by the user through code annotations. Our resulting memory events consist of the following data fields:

- global id
- event type
- code line
- memory address

The event type specifies the kind of memory access happening (read or write), or the occurrence of synchronization barriers. It is important to store the barriers in the same trace to restore the proper order of events afterwards. It is not possible to access a clock during the execution of a kernel and therefore all events are stored within work items but not in between work groups. The global id determines the work item that invoked the event, and the memory address reveals the reference of the accessed memory. Using the work-group size and the global id we can compute the local id. In order to link the access to the source code, the line of code is also stored. Note that this is the line of the original source code (not the instrumented one), which allows us to link events with the source code in the visualization.

### 4.1.3 Conditional Branch Traces

Branch divergence has a significant impact on the performance of parallel programs. In order to collect related data, we store the branching of each work item in a individual event with the following data fields:

- global id
- branch stream
- stream length

We can store the exact branching through the instrumentation of every conditional and iteration in the source code. In every resulting event the global id is stored to trace back the work item it origins from. Additionally, each event contains a binary branch stream in combination with its length. The branch stream contains true or false values for each executed iteration or conditional.

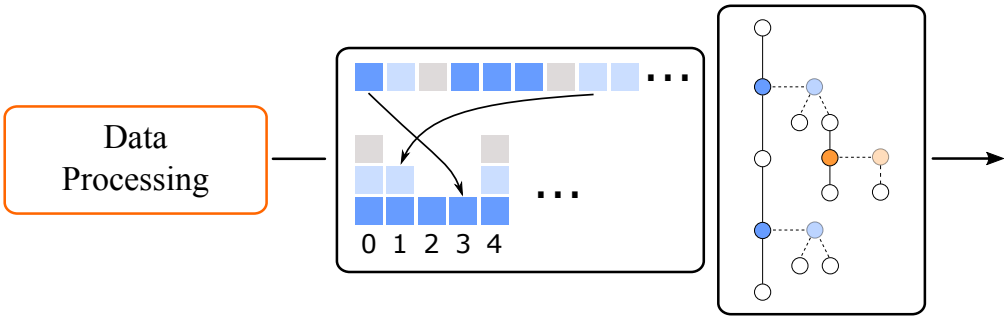



*Figure 4.3: Dataprocessing.*

## 4.2 Data Processing

After the data acquisition, the collected data is processed (see Figure 4.3) for a more suitable representation in regard to visualization, and similar data is aggregated.

### 4.2.1 Sorting of Memory Events

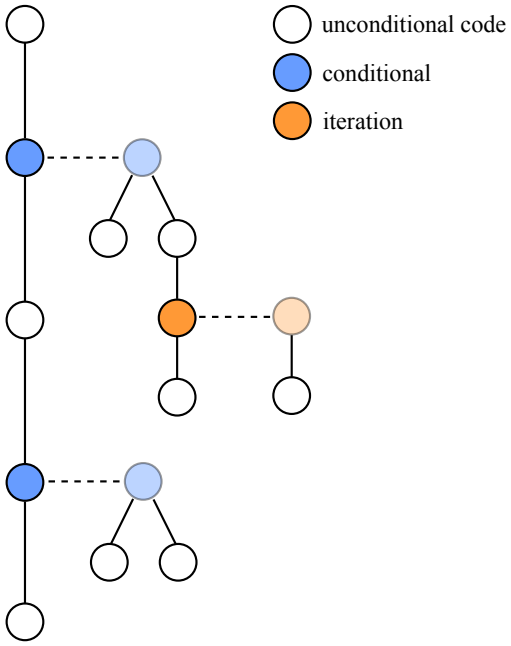
During the tracing of memory accesses all events are stored in a vector. The order of events within a work group is preserved but shuffled in between work groups. The recording of the synchronization barriers enables the sorting of events in regard to the logical time, which defines before and after relationships.

### 4.2.2 Processing of Branching

The processing of the branching uses the plain source code, as well as branch traces recorded during the execution. A semantic model of the program's structure is created during the processing of the code. We call the aggregation of warps with the same branch divergence proxy warps. They are computed after the collection of all traces.

#### Semantic Model

The semantic model is a result of the static analysis of the program's source code. This model reflects the semantic structure of the code regarding its branch points. It enables to trace back any collected branches to the corresponding parts of the source code. The model is represented as a tree, which consists of three different kind of nodes, shown in Figure 4.4a. Unconditional code is illustrated as a white circle. It consists of any code sequence that does not contain a branch statement. Blue circles represent conditionals, and orange circles iterations. Conditional and iteration nodes are both regarded as branching statements. A line connects a node to its child, or its inner part. Iteration and condition nodes are duplicated in the graph for a clear graphical representation and the duplication is depicted in faded colors. Figure 4.4a illustrates an example of a resulting branch tree next to the corresponding code in Figure 4.4b.




*(a)*

```
1  undconditional code;
2  if condition then
3  |  undconditional code;
4  else
5  |  undconditional code;
6  |  for start to end do
7  |  |  undconditional code;
8  |  end
9  |  undconditional code;
10 end
11 undconditional code;
12 if condition then
13 |  undconditional code;
14 else
15 |  undconditional code;
16 end
17 undconditional code;
```

*(b)*

*Figure 4.4: An example for a branching graph is shown in (a) with the corresponding source code in (b). Each white node represents unconditional code, each blue node a condition and orange nodes represent an iteration. Nodes that are connected with a dashed line represent the same node and are depicted in faded color. They are duplicated in the image for the purpose of a clear graphical representation.*

The branching of the execution of a program is stored into a binary stream. Through instrumentation each branching statement of the code appends true in case the branch condition is fulfilled, false otherwise. Due to iterations, a branching statement may append multiple values to the binary stream during execution. Algorithm 4.1 illustrates the decoding of a bit stream for a given branching graph. Note, that the algorithm skips unconditional nodes for simplification. The tree is traversed recursively starting with the first node. It then distinguishes between conditionals and iterations. In case of conditionals the next bit is fetched from the stream and the branch is chosen accordingly. Subsequently, the function is called recursively for the respective part. The function *eat_bit* fetches the current bit from the stream and sets a pointer to the next one. In case of an iteration node, a loop checks the bit of the stream until it is false.

**Algorithm 4.1:** Algorithm for the traversal of a given branching graph and stream.

```
1  Algorithm traverse(node, stream)
2      if type_of(node) == control_flow then
3          if eat_bit(stream) then
4              traverse (node.ifPart, stream)
5          else
6              traverse (node.ElsePart, stream)
7          end
8      end
9      if type_of(node) == iteration then
10         while eat_bit(stream) do
11             traverse (node.innerPart, stream)
12         end
13     end
14     if node.hasChild then
15         traverse (node.child, stream)
16     end
```

### Proxy Warp

During the execution of the program warps we store the branching of each work item. This information is collected to investigate branch divergence. Branch divergence happens on warp level hence we summarize the data accordingly. Then all warps that suffer from branch divergence are filtered. This still leaves an overwhelming number of warps to investigate. Generally many warps possess the same branching behavior. Therefore, proxy warps are introduced in order to limit the number of warps that have to be investigated. A proxy warp represents all warps that contain the same branching. Consider a warp as follows:

$$w = (t_1, t_2, ..., t_n) \tag{4.1}$$

Each warp consists of n work items and each work item follows a certain branching during execution. The branching of the $m$-th work item in warp $w$ is donated with $\beta(w, m)$.

$$\beta(w, m) = (b_1, b_2, ..., b_k),\ b \in \{0, 1\} \tag{4.2}$$

The length $k$ of a branching can change for every execution and is depending on the input data of the program. Each $b$ represents a branching node with a value of either 0 or 1. We define the branching of two warps to be equal with:

$$w_a = w_b \leftrightarrow \forall\ n \in [1\ ..\ N] : \beta(w_a, n) = \beta(w_b, n) \tag{4.3}$$

A proxy warp represents all warps with equal branching. Using this aggregation, the user can focus on investigating the proxy warp with its representative behavior for all equal

warps. A use case, which shows proxy warps in the context of understanding and analyzing performance in an image processing example is presented in Chapter 7.2.

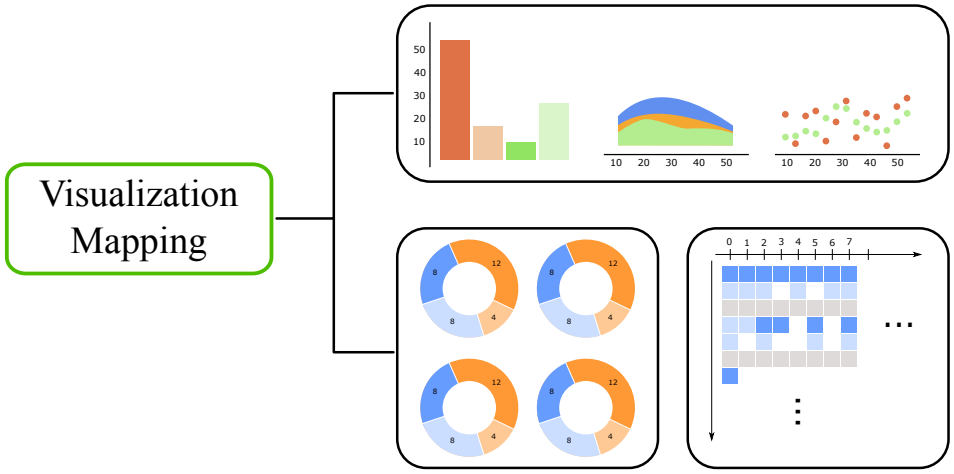



*Figure 4.5: Visualization mapping.*

## 4.3 Visualization Mapping

This visualization mapping step of the pipeline deals with the mapping of recorded data to meaningful visual representations. In the case of visualizing parallel programs there are three prominent topics to differentiate:

- **Hardware:** The visualization of the underlying hardware is essential to understand its influence on performance and the overall behavior of the program. Memory fields are frequently visualized as blocks in grids [Ros13, CPP08] with color coded read and write access.
- **Software:** One of the main aspects of the exploration of parallel programs is the visualization of the set of instructions (software) that controls the application. Recorded events of instructions are often used for visualization and shown in icicle plots [KL83] to depict the concurrent and chronological component of the execution. Node link layouts [CHZ+07] are used to represent call graphs of functions and hovering over a node of the graph highlights the corresponding source code [ABF+10].
- **Synchronization and Communication:** The visualization of the synchronization and communication between work items grants insight into their relation. It also reveals sections of idle times, which occur if work items wait for synchronization or kernels wait for results.

Our visualization views incorporate the above mentioned aspects and are presented in the following.

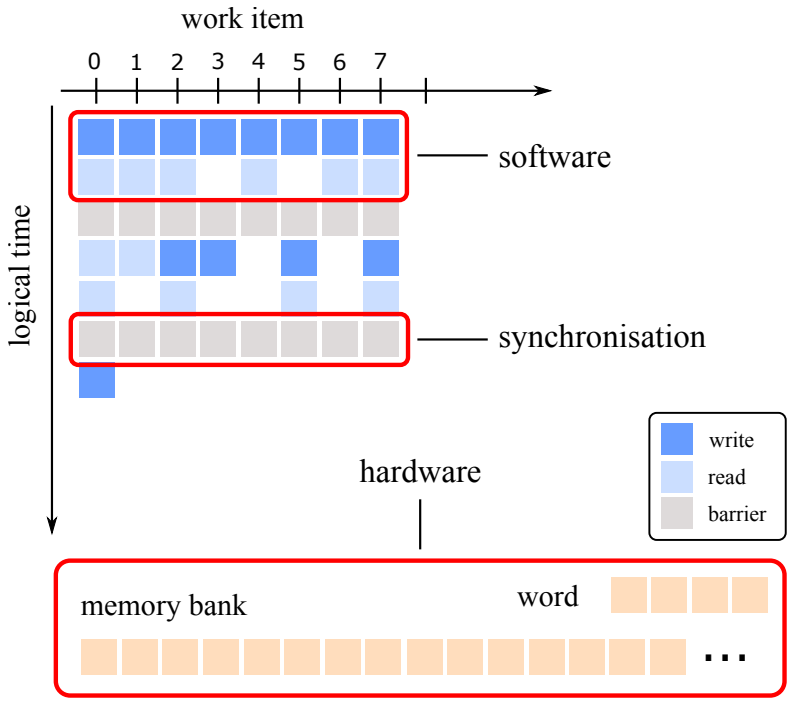



*Figure 4.6: Visualization of local memory accesses in a graph. A blue block depicts a write access, a light blue block depicts a read access and a gray block encodes a synchronization barrier. Different work items are spread along the x-axis. The y-axis represents the logical time.*

### 4.3.1 Visual Encoding of Memory Accesses

We introduce a view that depicts memory access and synchronization barriers in a block view, as shown in Figure 4.6. Each block represents one specific event. The y-axis shows the trace of events from each work item. Read and write accesses are sorted according to the synchronization barrier. There is no exact timing of the events, synchronization barriers provide only before and after relations. Nevertheless, this means that all events within two barriers are executed during the same time span. The view grants insight into the number of events each work item produces, as well as their relation to the hardware and the line of code. Hovering over an event, highlights the corresponding field in the memory bank and the corresponding word, in addition to the related line in a source code view. Hovering over a memory bank highlights all events that accesses this memory bank, as shown in Figure 4.7. This visualization facilitates the exploration of efficient memory accesses, as well as the exploration of bank conflicts.

### 4.3.2 Visual Encoding of Branching

Branch divergence is visually encoded in a donut chart, as shown in Figure 4.8. The donut chart depicts the different branching patterns of a warp. Equal branchings are aggregated and depicted as a connected region with a distinct color. In order to reduce the total number

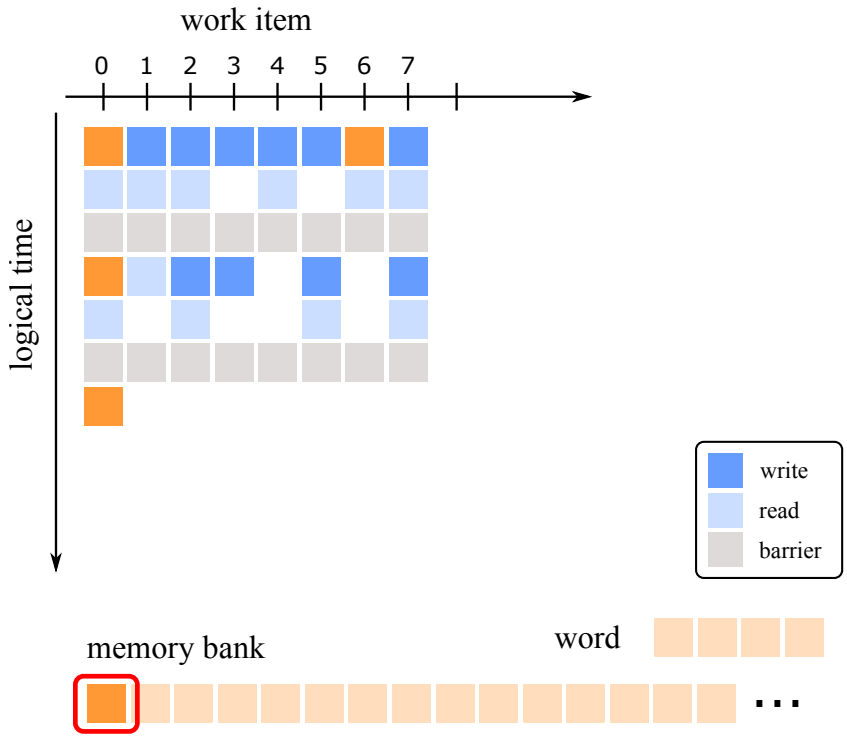



*Figure 4.7: If the user hovers over a memory bank field, all corresponding memory access blocks are highlighted.*

of warps in the visualization, only proxy warps are depicted. They build an aggregated view of warps with the same branching pattern. The number inside the donut chart shows the total number of warps that are represented by this chart.

### 4.3.3 General Visual Mapping with D3

The proposed visualization approach is easily extensible using JavaScript and the D3 library. Annotated data structures are already inserted automatically in a web-browser environment and directly accessible. A collection of D3 charts is provided with simple configuration parameters. They follow the guidelines of reusable charts proposed by Bostock [Mik]. The available visualizations include: Line Charts, Scatter Plots, Multi-Bar Charts, and Stacked-Area Charts.

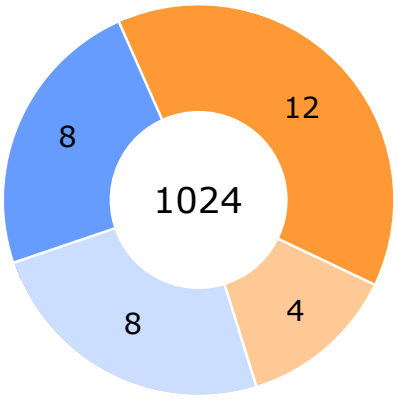



*Figure 4.8: Illustration of the branch divergence of a warp as a donut chart. The different regions shown in the illustration depict distinctive branchings within the warp. Each branching is executed by a number of work items displayed by the numbers in the individual regions. The number in the center of the illustrations shows the number of warps that are represented by the proxy warp.*

CHAPTER 5

# Language Design

Our system design is based on two scenarios: first, a user has an existing kernel, or second, the user utilizes our system to create a kernel from scratch. Both approaches benefit from the visual exploration support, either in the course of implementation, or through the investigation of the already implemented kernel. In both cases, we concluded that the utilization of a DSL for this purpose proves to have numerous benefits, as outlined in the following:

- **Fast prototyping:** Our framework automatically creates the setup and boilerplate code that is required for the generation and execution of parallel programs. They remain hidden from the syntax of our DSL. In addition to that, the DSL does not need to separate between host and device data, since their transfer is also managed by the framework. Furthermore, the turnaround time significantly benefits from our approach, since our DSL is a combination of interpreted and just-in-time compiled language. The resulting OpenCL kernels are compiled in the background.

- **Instrumentation:** In order to record data during the execution of the program, certain parts in the code have to be instrumented. Instrumentation requires the knowledge of the code structure. A DSL creates this structure during the parsing process. The resulting abstract syntax tree (AST) can be utilized for automatic instrumentation.

- **Semantic Model:** Not only instrumentation, but also the analysis of branching patterns requires the structure of the code. Again, this structure can be inferred from the AST.

- **Code Annotations:** The recorded data during the execution of a parallel program usually produces numerous and broadly defined traces. Typically, the user is only interested in a subset of the traces. Code annotations facilitate a fine-controlled level of granularity to configure the creation of traces.

- **Link to visualizations:** Linking of resulting data to visualizations can be directly incorporated into the DSL. This is advantageous, because it only presents one simple interface to the user for the whole process of kernel creation and exploration.
- **Data structures:** The usage of a DSL enables extensibility for new and optimized data structures. For instance, JiTTree [LBG+16] shows this by implementing a sparse volume data structure integrated in our language system. This structure is optimized and able to adapt to specific datasets during runtime. The access of the data structure is hidden from the user and does not add complexity to the language.

In case these points are regarded separately, numerous approaches for implementation would be possible. However, in combination, a DSL is the perfect fit, since it provides the means to implement all of the above mentioned aspects in one consistent interface. The design of our DSL is presented in the following.

## 5.1 Language Structure

Our DSL is closely related to OpenCL 1.2 but consists of three additions:

- **Kernel:** Equivalent to OpenCL, the kernel is the part of the code that executes parallel computations on compute devices. Kernel definitions are marked with the keyword "*kernel*" and are implemented in a similar fashion to the definition of a common method. Invoking a kernel in our language will automatically transfer its passed arguments from the host to the compute device. Example code is:

```
using DSL;
kernel myKernel(global float *in, global float *out)
{
    int id = global_id(0);
    out[id] = in[id]*2.0;
}
using;
float[512] in;
float[512] out;
// generate input data
// ...
DSL.setLocalWorkSize(32,1);
DSL.setGloballWorkSize(512,1);
DSL.myKernel(in, out);
```

- **Annotations:** Annotations can be inserted in between standard instructions of the implemented code. They configure the instrumentation of the source code. Annotations are not part of the logic of the algorithm and are only used to extract data for debugging and profiling purposes. Annotations can also be deactivated on demand. Compiling the code without the instrumentation leads to a clean OpenCL kernel that is not influenced in its performance. Example code is:

```
kernel myAnnotatedKernel(global float *in, global float *out)
{
    local float dataA[128];
    local float dataB[128];
    @watch dataA;
    @watch dataB;
}
```

- **Visualization:** The configuration of the visualization is defined outside of the kernel source code. It enables the linking of collected data to visual representations. They are defined in a declarative style, in order to provide fast and simple specifications. Example code is:

```
kernel myKernel(global float *in, global float *out)
{
    //...
    int idX = get_global_id(0);
    local float data[128];
    @emit float emittedData[128];
    emittedData = data[idX];
}
@bind ["class":"stackedAreaChart", "data":["emittedData"]]
//...
```

## 5.2 Syntax

Our syntax closely resembles the OpenCL C language, which provides two benefits. Firstly, familiar users are not required to learn new language concepts, which is usually the greatest drawback of a DSL. Secondly, it enables the insertion of existing OpenCL kernels without changes to the existing code. The supported language constructs combine the definition of the actual program, code annotations, and visualization linking. Figure 5.1 shows the syntax in extended Backus-Naur form for the most important language constructs. Trivial language constructs are not further defined here, hence their syntax is equal to OpenCL C. Our language supports constructs, such as conditionals, loops, declaration and assignments. In addition to that, we support a range of OpenCL constants and address qualifiers. The ultimate goal is to be completely compatible with any OpenCL kernel. One advantage of our DSL in comparison to OpenCL C is that data is automatically available from the host as well as the compute device side. Furthermore, kernel arguments are not set via an API, as in the case of OpenCL. They are automatically set, triggered by the invocation of a kernel. The whole setup of a kernel and its specification are also handled in the background and not part of the syntax. Furthermore, adding specific data structures to the language can be realized without changes to the syntax, as explained in Section 6.

The user can add code annotations in between any instructions of the code. To clearly distinguish code annotations from the basic language they start with a leading ”@“-sign. Code

```
program          =  device_program, configuration
device_program   =  {kernel}
configuration    =  {chart_declaration}
kernel           =  "kernel", identifier
                 ,  "(", type, identifier , {",", type, identifier} ,")"
                 ,  "{" , {instruction} , "}"
instruction      =  if_statement
                 |  while_statement
                 |  for_statement
                 |  function_declaration
                 |  declaration
                 |  assignment
                 |  code_annotation
expression       =  terminal_expression
                 |  arithmetic_expression
                 |  boolean_expression
code_annotation  =  emit_statment
                 |  watch_statement
watch_statement  =  "@watch" identifier
emit_statment    =  "@emit" identifier
chart_declaration =  "@bind[ class:", identifier ,
                 ,  "data: [", identifier , {",", identifier}, "]"
```

*Figure 5.1: DSL syntax in extended Backus–Naur form.*

annotations are less intrusive to the program code than the manual adding of instructions for data extraction. Through the insertion of annotations in a specific line of code, the user specifies where the recording of the data starts. However, the watch_statement and the emit_statement have to be inserted after the definition of the targeted variable.

## 5.3 Data Types

Besides the support of primitive data types, such as boolean, integer and float, the language supports a number of vector and abstract types. A list of supported data types is presented in Table 5.1. The definition of vector-component addressing is described in the OpenCL Specificaton [Gro]. The size of a vector data (n) in our language is either, 2, 3, 4, 8 or 16. Besides common C-style arrays, our language also supports 2D images and samplers. Images are read with the instruction "read_imagef(image2d_t *img*, sampler_t *sampler*, int2 *coord*)", which return a float between zero and one. A sampler object controls how elements of the image are read. For the detailed description of samplers see the OpenCL Specification [Gro].

| Scalar Types | Description |
|---|---|
| bool | either true or false (size depends on vendor implementation) |
| char | 8-bit signed |
| uchar | 8-bit unsigned |
| short | 16-bit signed |
| ushort | 16-bit unsigned |
| int | 32-bit signed |
| uint | 32-bit unsigned |
| long | 64-bit signed |
| ulong | 64-bit unsigned |
| float | 32-bit float |
| half | 16-bit float |

| Vector Types | Description ( $n \leq [2,3,4,8,16]$ ) |
|---|---|
| char*n* | n 8-bit signed |
| uchar*n* | n 8-bit unsigned |
| ushort*n* | n 16-bit unsigned |
| int*n* | n 32-bit signed |
| uint*n* | n 32-bit unsigned |
| long*n* | n 64-bit signed |
| ulong*n* | n 64-bit unsigned |
| float*n* | n 32-bit float |

| Abstract Types | Description |
|---|---|
| image2d_t | 2D image handle |
| sampler_t | sampler object |

*Table 5.1: List of supported data types.*

## 5.4 ViSlang Environment

Our DSL is developed as plugin for ViSlang. ViSlang is a comprehensive system for domain-specific languages in the context of scientific visualizations. It provides its own base language, which facilitates easy data handling and processing and realizes calls of implemented plugin methods. A major advantage of the DSL being a ViSlang plugin is the provision of already existing resource types and data structures. Furthermore, it allows the user to build up on existing plugins, such as plugins for data generation, or data processing.

Kernels written in our DSL are registered in the ViSlang environment after they are parsed. The user can utilize ViSlang to create and process the data before the kernel is invoked.

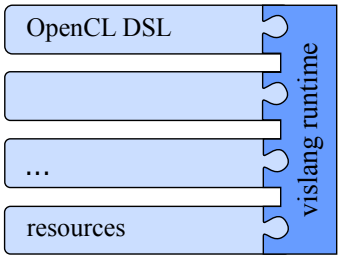



*Figure 5.2: The illustration shows the relationship between ViSlang and its plugins.*

For instance, a kernel with the name *sobelKernel* and arguments is invokable through the ViSlang runtime with DSL.sobelKernel(...). The passed parameters have to correspond to the defined arguments. For a detailed definition of the language and a specification of features see [RBGH14].

CHAPTER 6

# Framework

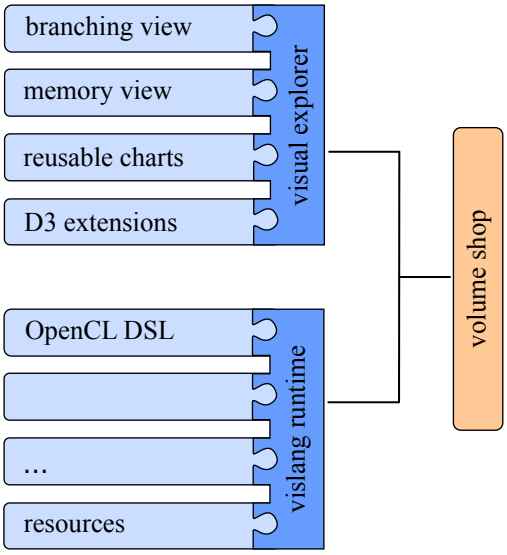



*Figure 6.1: The framework architecture of our system. Our OpenCL DSL serves as a plugin for the ViSlang framework. The VolumeShop framework underlies all modules and provides the connection between the vislang runtime and the visualizer.*

The presented domain-specific language is implemented in C++ and based on the ViSlang [RBGH14] and VolumeShop [BG05] framework. The parser utilizes the spirit parser framework, which is part of the well-known boost libraries [Kar05]. The interactive visualizations are web-based and created with HTML, CSS, JavaScript and the D3 library [BOH11]. The set of charts provided in our visual explorer are created with the aid of NVD3 [NVD], a JavaScript library for re-usable charts, implemented with D3.

Figure 6.1 presents an overview of the framework architecture. Our OpenCL DSL serves as a plugin of ViSlang. The Volume Shop core underlies the architecture and provides the connections between visual explorer and the ViSlang runtime. The visual explorer provides different views for the visualization of branchings, memory accesses and individual charts.

## 6.1 Interface

Figure 6.2 shows the graphical user interface of our integrated system. The editor shown in Figure 6.2a is used for the implementation of kernels in our DSL, as well as for the generation of test input data and the computation of kernels. When the program is run it automatically translates the DSL to OpenCL code, interprets the commands for the input generation, enqueues the kernel, waits for the result and visualizes the additional data that was generated during the execution of the kernel. An immediate visual response is shown in the visual exploration panel in Figure 6.2b. For convenience our system also integrates a console shown in Figure 6.2c and a global variable view shown in Figure 6.2d.

## 6.2 Language Implementation

The language is implemented as a source-to-source compiler. The code submitted by the user runs first through a parser and the resulting AST is then interpreted by a code generator, which creates the actual OpenCL code. The particular steps of the code generation process, shown in Figure 6.3, are explained in the following.

### 6.2.1 Lexical and Syntactic Analysis – Parser

The first step after a user submits the program is the lexical and syntactic analysis. The input is then split into a sequence of tokens. Subsequently, the parser examines the syntactic correctness of the tokens. For the examination a predefined grammar is used. The parser recursively analyzes and matches the tokens for confirmation to the rules of the grammar.

The tokenizer and parser of this framework are built on top of the boost spirit framework, which is a recursive descent parser generator implemented in C++. It handles the lexical analysis by automatically structuring the input into the above mentioned tokens. During the parsing step, the AST is computed, which represents the structure of the program. Furthermore, our parser supports error messages that contain the corresponding line of code and the exact character of the occurring error.

### 6.2.2 Semantic Analysis and Code Generation

The code generator module is responsible for the semantic analysis, as well as the generation of OpenCL code. This step verifies the semantic correctness with the aid of a symbol table, which is progressively built during the analysis process. In case of a semantic error, the code generator outputs an error message, which contains the line of code where it occurs

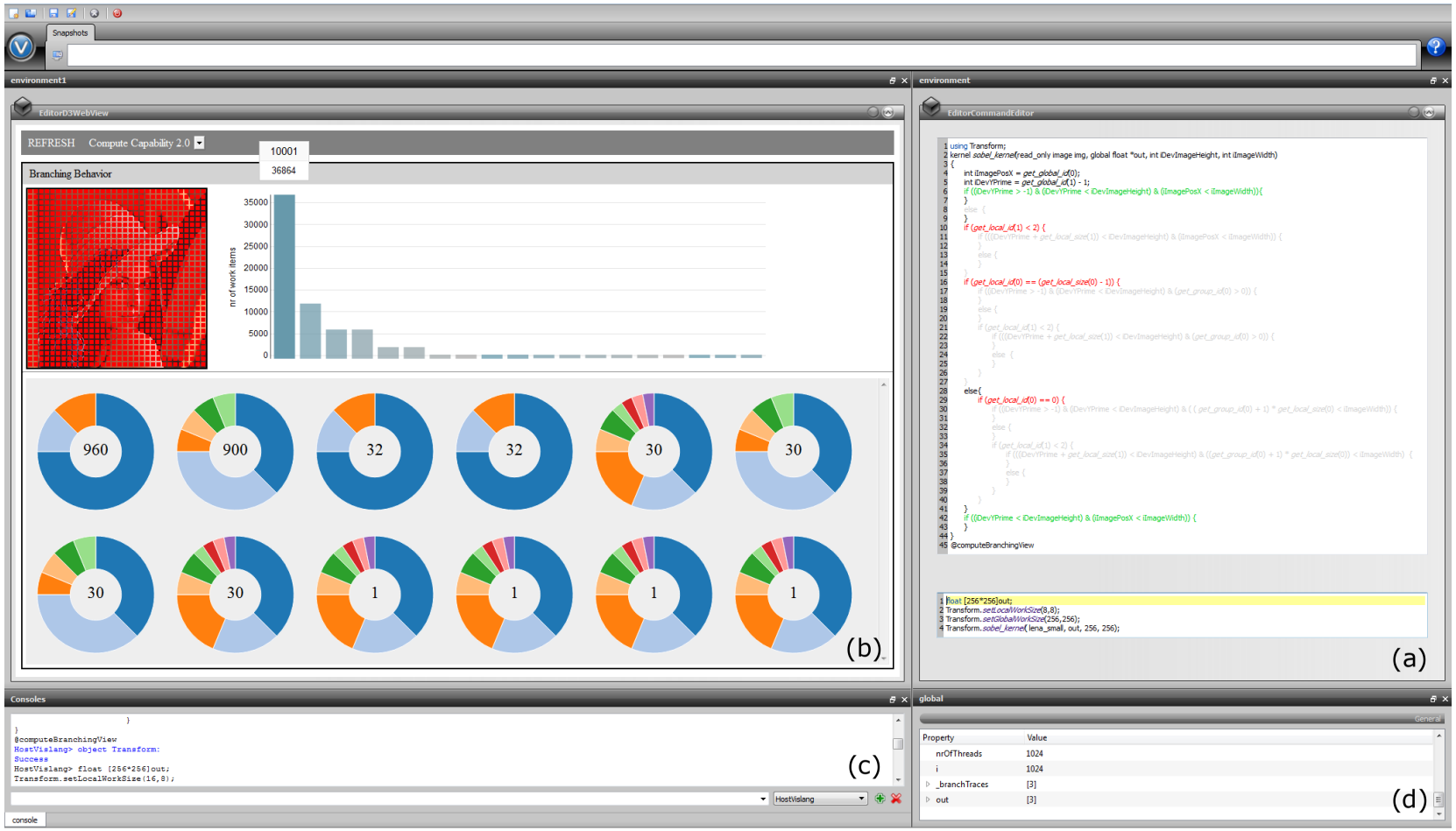



*Figure 6.2: Graphical interface of our integrated system that provides interactive visualizations of parallel programs: (a) shows the source code editor, (b) shows the visual exploration panel that is linked with (a). (c) shows a console and (d) the global variable view.*

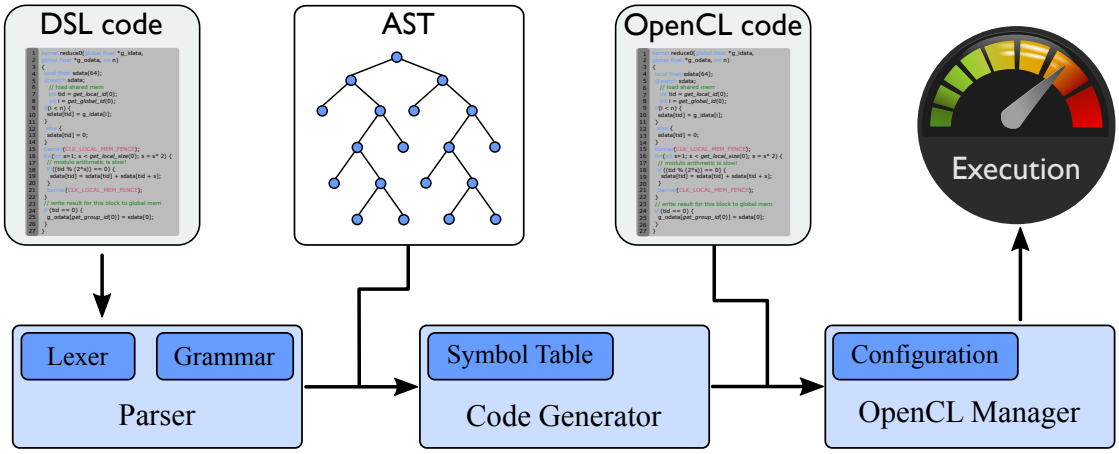



*Figure 6.3: The consecutive steps of the code generation process. First the code is submitted by the user and then analyzed in the parser. The lexer splits the input into tokens and the parser validates its syntactic correctness on the basis of a predefined grammar. The resulting AST is then used to generate valid OpenCL code by the code generator. In the last step, the OpenCL manager compiles and executes the code.*

and indicates the wrong symbol. Furthermore, it provides hints about the error cause, e.g. revealing if a used symbol is not yet defined.

### 6.2.3 Code Execution

Once the actual OpenCL code is generated, it is dispatched to the OpenCL manager for compilation. The manager also comprises a configuration for the OpenCL execution. The configuration consists of information, such as work-group size, deployed OpenCL extensions, and OpenCL build options. Our DSL provides two methods (setLocalWorkSize(*int*, *int*), setGlobalWorkSize(*int*, *int*)) to set the local and global work-group size, respectively. Our implementation currently supports two-dimensional kernels only. Nevertheless, this is not a limitation of the approach and can easily be extended in future work.

## 6.3 Code Instrumentation

The code instrumentation step is incorporated into the code generation process, as described in Chapter 6.2. During the traversal of the AST, we instrument conditionals and variables to produce traces. In detail, every time a conditional is traversed, we insert a function call that appends the corresponding trace. We utilize the comma-operator in order to be minimally intrusive to the code. Algorithm 6.1 provides example code that contains a simple for-loop. The corresponding instrumented code is shown in Algorithm 6.2. The instrumented code consists of a structure definition for branching traces and a function for adding a binary encoded branching to the current trace. Furthermore, right before the condition in the for loop, the function *add_bit* is called to store the outcome of the condition.

Algorithm 6.3 shows an example that contains the declaration of a local variable, as well as its read and write access. The corresponding instrumented code is shown in Algorithm 6.4. The instrumentation shows that after each access, a function call is added to the code, in order to create a new event that stores the related information.

*Algorithm 6.1: Original code of branching example.*

```
kernel void program(args)
{
    for(int i = 0; i < n; n++)
    {
        // ...
    }
}
```

*Algorithm 6.2: Instrumented code of branching example.*

```
typedef struct
{
    uint2 gID;
    uint branchStream;
    uint streamLength;
} _BranchTrace;

void add_bit(bool c, _BranchTrace *trace)
{
    trace->branchStream += c;
    trace->branchStream <<= 1;
    ++trace->streamLength;
}

kernel void program(args, __global _BranchTrace *_traces, __global int *_counter)
{
    for (int i = 0; _add_bit(i < n); i < n; n++)
    {
        // ...
    }
    _traces[atomic_inc(&_counter[0])] = trace;
}
```

*Algorithm 6.3: Original code of memory access example.*

```
kernel void program(args)
{
    local float sdata[64];
    float x;
    @watch sdata;
    sdata[0] = x;
    x = sdata[0];
}
```

*Algorithm 6.4: Instrumented code of memory access example.*

```
typedef struct
{
    uint gID_0;
    uint gID_1;
    uint type;
    int address;
    uint codeLine;
} _MemoryEvent;

void _add_event(__global _MemoryEvent *traces, uint type, uint codeLine, uint
    address, __global uint *_counter)
{
    _MemoryEvent event;
    event.gID_0 = get_global_id(0);
    event.gID_1 = get_global_id(1);
    event.type = type;
    event.codeLine = codeLine;
    event.address = address;
    traces[atomic_inc(&_counter[0])] = event;
}

kernel void program(args, __global _MemoryTrace *_traces, __global uint *_counter)
{
    local float sdata[64];
    float x;
    sdata[0] = x;
    _add_Event(traces, 0, 3, &sdata[0]), _counter);
    x = sdata[0];
    _add_Event(traces, 1, 4, &sdata[0]), _counter);
}
```

## 6.4 Structures and Memory Management

We use the boost compute library [Lut] for efficient management of data structures. It supports arrays with complex structures and also handles the memory transfer between host and compute device. Our system automatically triggers the memory transfer when a kernel is called. A memory manager keeps track of the allocated memory and organizes the proper deletion. A common interface for data structures provides the means to extend the system with new data structures. The implementation of the interface has to handle the corresponding argument setting for the kernel and to expose functions for the read and write access of the data structure.

## 6.5 Visualization Implementation

We use the QT WebKit [QT] module to deploy the web browser of QT to insert our visualization in our web-based environment. The data that is acquired during the execution of the program is translated into the QT Object Model in order to enable its transfer into the QT web browser. Every time, the user changes the visualization configurations or executes the kernel we send the required information to the browser and call the corresponding update functions of the web page. Furthermore, we pass information about our used hardware to our visualization system, such as memory bank size and warp size in addition to information about the execution setup like local and global work group size.

We utilize the D3 (data-driven documents) framwork [BOH11] to create visualizations. The set of re-usable charts is created with the NVD3 library [NVD]. The data transfer between the program and the visualization plugin is realized mainly with JSON encoded objects.

CHAPTER 7

# Case Studies and Evaluation

Parallel programs are typically designed for high-throughput and performance. The complexity of the underlying hardware architecture directly impacts the complexity of the code. In many cases multiple implementations of the same algorithm compete for optimal performance and memory usage. Our system aims to enhance the understanding of parallel programs, their competing implementations, and their possible impact on performance.

In this Chapter we discuss the design choices of our system and highlight the features that help to achieve these goals. We evaluate them with two case studies that exemplify the implementation of parallel programs. The evaluation focuses on the following criteria:

(1) **Code Understanding:** The goal of a visualization is to improve the understanding of the explored algorithm. It is important that the semantics of parallel programs and especially of the hardware are mapped to visual representations. The visualization should improve the readability of the algorithm and thereby enhance the user's understanding.

(2) **Program Comprehension:** The connection between the visualizations and the source code of the algorithm is important to enable the user to understand the program not only in its functionality, but also on a code level, as well as its runtime behavior. The system is useful if different implementations as well as different invocations of kernels result in visualizations that clearly show these differences.

(3) **Predictability:** The impact of different runtime patterns, branching patterns, and memory access patterns should be reflected in the visualization. The system should allow the user to derive where performance issues originate. The measurable performance should improve if the visualizations indicate that the implementation improved.

In Section 7.1, we first describe the composition of views and analyze the turnaround times of our system. The analysis is performed in order to evaluate the usability and flexibility

of our system. Then, we present two detailed case studies of an image filtering program (Section 7.2) and the investigation of multiple implementations of the parallel reduction algorithm (Section 7.3).

## 7.1 Turnaround Time and View Composition

A major aspect in the workflow of debugging and profiling is the provision of fast response times and the possibility to quickly change various setups and implementations. The usual approach of testing different setups is realized by manually changing the code of the application and the subsequent re-compilation and execution. Visual debugging of parallel programs typically involves tedious setup of visualizations that are cumbersome and rigid. Our system simplifies and shortens this process through fast turnaround times of our language. The reason for fast turnaround times and increased flexibility is that is a combination of an interpreted and just-in-time compiled language, which omits a costly pre-compilation and the manual mapping from debug data to visualizations. The user simply changes code, adds code annotations and triggers the execution during the runtime of the parallel programs. All steps are integrated in our system, happen during the runtime of our system and are hidden from the user. We first describe the composition of views that were used in our case studies an later report the turnaround times for these cases.

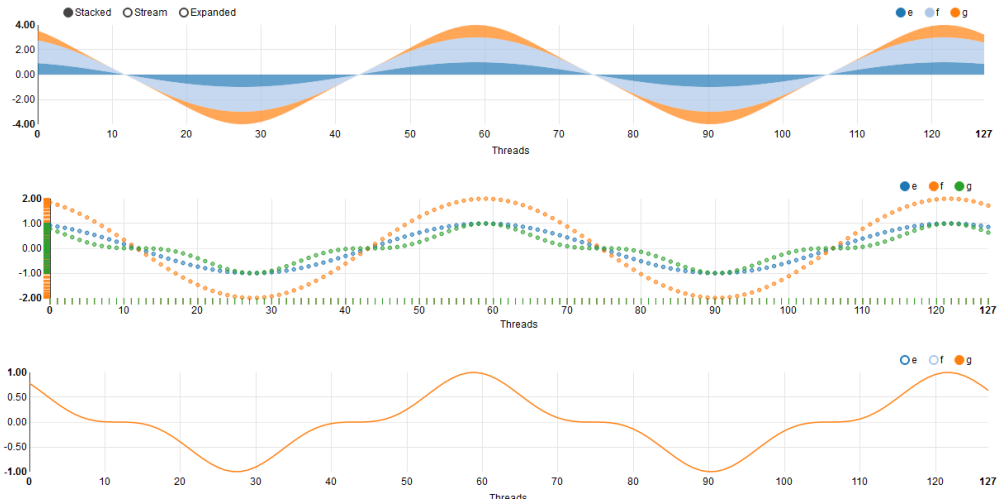



*Figure 7.1: Illustration of the debug view, which depicts 2D data in various charts.*

### 7.1.1 Debug View

The debug view is a simple composition of various charts, which depict 2D data. In most debugging cases, the user manually adds code to the implementation to transfer intermediate data of the kernel execution and then inspects the raw data. In our approach the user can add a simple code annotation to emit data and subsequently a code annotation to bind it to selected visualizations (see Chapter 5.2). The debug view automatically composes the

chosen charts and inserts the corresponding data values, as illustrated for three variables (e,f,g), in Figure 7.1.

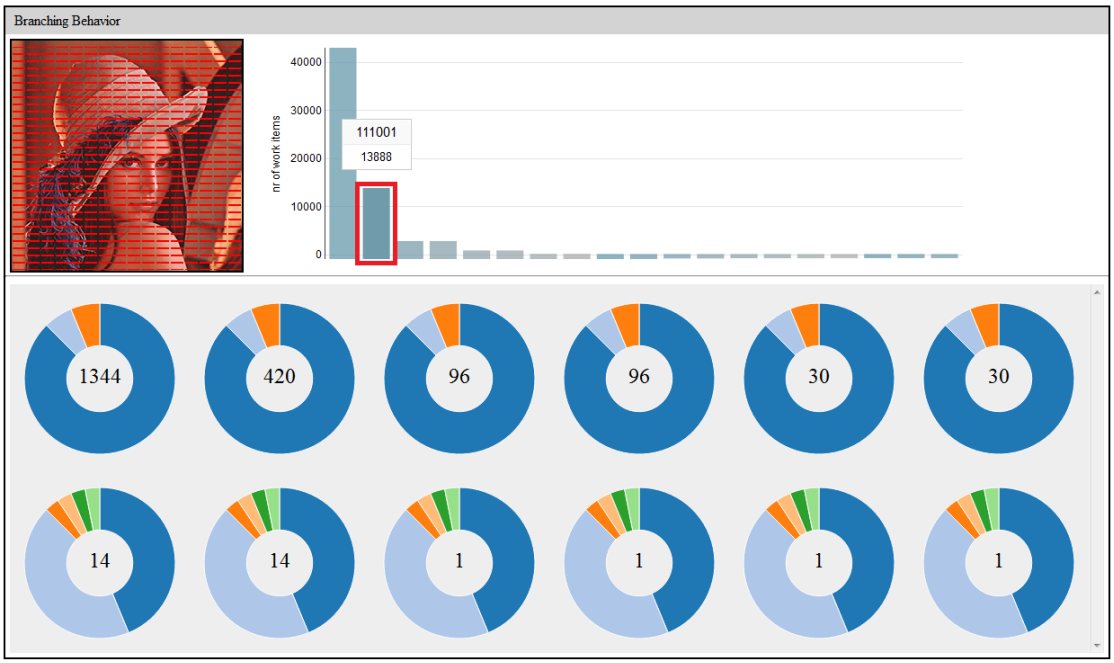



*Figure 7.2: The branching view is divided into an overview (top) and a number of proxy warp illustrations (bottom). The overview consists of a histogram that reveals different branching patterns and an image for image processing kernels. When hovering over an histogram bin (as highlighted in the image) or over a region of the donut charts, work items that took the corresponding branching pattern are highlighted in the input image. We assume a sequential mapping of a pixel in the image to the corresponding computing work item.*

### 7.1.2 Branching View

The purpose of the branching view is the support in the understanding of the investigated implementation as well as to assist performance improvements. Control flow statements in parallel programs potentially lead to branch divergence, significantly affecting the performance. In our system, we analyze different branching patterns by interpreting all branches as binary decisions of the corresponding work item. These decisions are encoded as a bit string, so that each work item can easily record the traversed branches during execution. The resulting data is presented in the branching view that consists of an overview and visualizations of several proxy warps.

#### Branching Histogram Visualization

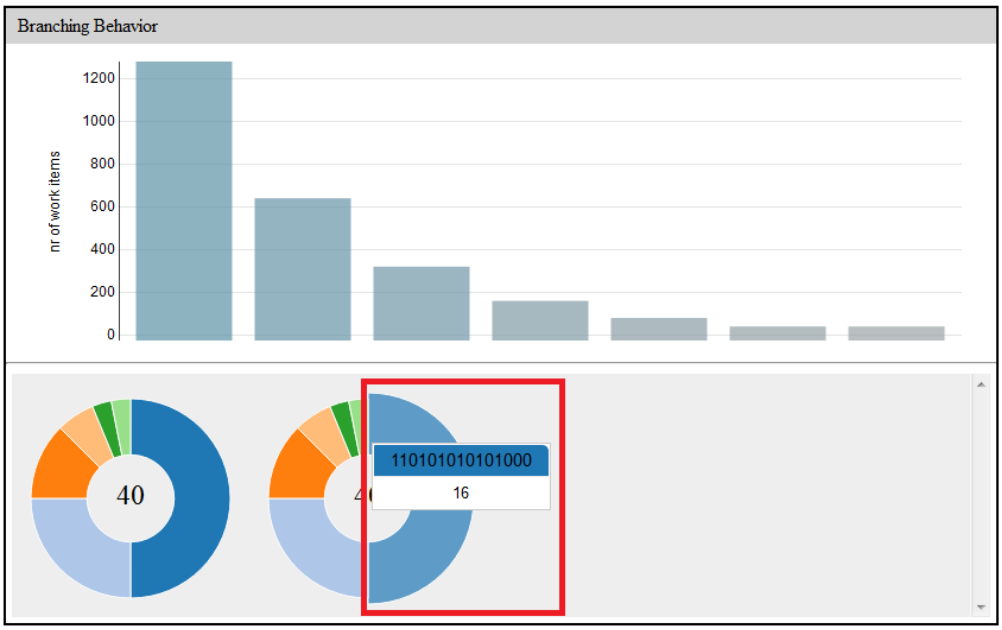
The top part of the branching view (see Figure 7.2) provides a first-level overview of the investigated program. It consists of a histogram that reveals different branching patterns

and an image that represents the input. The image only appears if the investigated program actually has an image as input. The purpose of the histogram is to provide a rough but rapid understanding of the execution of the program. Each bin of the histogram depicts a different branching pattern. The height of a histogram bin indicates the number of work items that execute the corresponding branching pattern. Since the x-axis in this visualization has no semantic meaning by itself we sort the bins by their size. When hovering over a histogram bin, as indicated with a red box in Figure 7.2, work items that took the corresponding branching pattern are highlighted with red in the input image. Note, that we assume a sequential mapping of a pixel in the image to the corresponding computing work item. Additionally, a label appears, which shows the binary encoded branching pattern (top of label) and number of work items that executed this branching pattern (bottom of label).

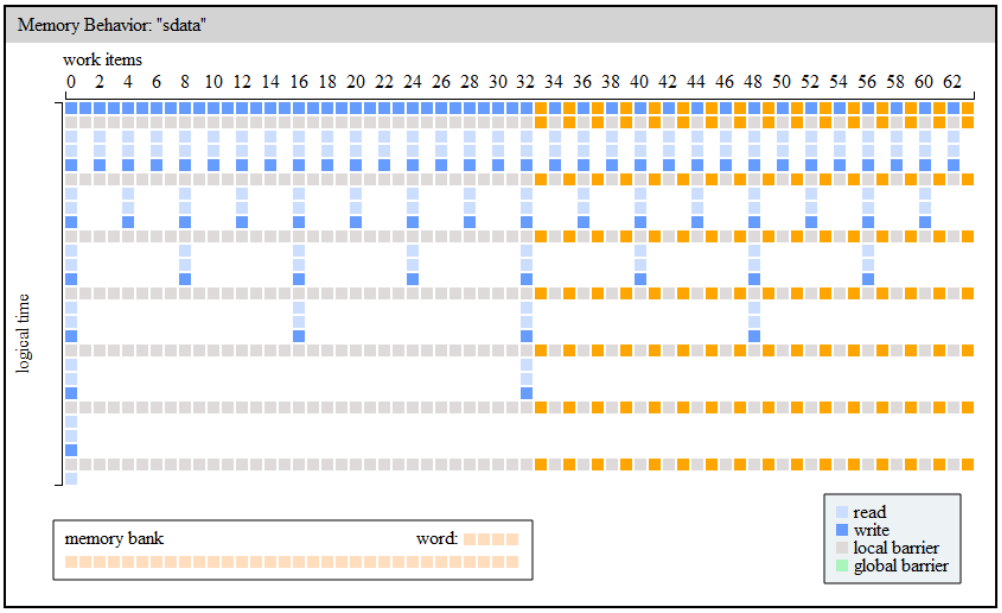
The branching view is also linked with the source code view. When hovering over a histogram bin corresponding lines of code are highlighted. Figure 7.4 illustrates an example of the code highlighting, which corresponds to the hovering happening in Figure 7.2. The binary encoding of the branching pattern in Algorithm 7.4 is *111001*. Code that is not executed by the work items that are represented by the chosen bin is faded out. Conditionals are color encoded with green, if their condition is evaluated to true, or red otherwise. Each digit of the binary encoded branching pattern can be visually matched to the green (*1*) and red (*0*) colored conditionals.

### Proxy Warps Visualizations

The proxy warps visualizations in Figure 7.2 (bottom) reveals branch divergence and supports the user in finding their causes. A proxy warp is visualized as donut chart. The number of donut charts that appear in the visualization corresponds to the number of groups of equal warps (as explained in Chapter 4.2.2). The number in the midst of the donut chart shows the total number of warps that are represented by the proxy warp. The donut charts are sorted in descending order accordingly to this number. Each region of the donut chart represents a distinct branching pattern. The size of a region corresponds to the number of work items that exhibit a particular branching pattern. The cumulative number of all regions always corresponds to the number of work items within a warp, which is 32 for all hardware architectures under investigation in the following use cases. When hovering over a region of a donut chart, pixels that were processed with the corresponding work items that took the corresponding branching pattern are highlighted with red in the input image. Furthermore, different views are dynamically linked with each other. When the user hovers over a region of a donut chart every memory access of corresponding work items are highlighted in memory views, as illustrated in Figure 7.3.




*(a)*




*(b)*

*Figure 7.3: Linking between the branching view (top) and the memory view (bottom). When hovering over an interactive item in the branching view, memory accesses of the corresponding work items are highlighted in the memory view. For instance, if the user hovers over a region of a proxy warp, as highlighted with the red rectangle in the image, corresponding elements are highlighted in orange in the memory view.*

```
kernel sobel_kernel(read_only image img, global float *out)
{
  int iImagePosX = get_global_id(0);
  int iDevYPrime = get_global_id(1) - 1;
  int iDevGMEMOffset = (iDevYPrime * get_global_size(0)) +
                       iImagePosX;

1 if((iDevYPrime > -1) & (iDevYPrime < iDevImageHeight) &
  (iImagePosX < iImageWidth)) {
  }
  else  {
  }
1 if (get_local_id(1) < 2) {
1   if (((iDevYPrime + get_local_size(1)) < iDevImageHeight) &
    (iImagePosX < iImageWidth)) {
    }
    else {
    }
  }
0 if (get_local_id(0) == (get_local_size(0) - 1)) {
    if ((iDevYPrime > -1) & (iDevYPrime < iDevImageHeight) &
    (get_group_id(0) > 0)) {
    }
    else {
    }
    if (get_local_id(1) < 2) {
      if (((iDevYPrime + get_local_size(1)) < iDevImageHeight) &
      (get_group_id(0) > 0)) {
      }
      else  {
      }
    }
  }
  else{
0   if (get_local_id(0) == 0) {
      if ((iDevYPrime > -1) & (iDevYPrime < iDevImageHeight) &
      ((get_group_id(0) + 1) * get_local_size(0) < iImageWidth)) {
      }
      else {
      }
      if (get_local_id(1) < 2) {
        if (((iDevYPrime + get_local_size(1)) < iDevImageHeight) &
        ((get_group_id(0) + 1) * get_local_size(0)) < iImageWidth) {
        }
        else {
        }
      }
    }
  }
1 if((iDevYPrime < iDevImageHeight) & (iImagePosX < iImageWidth)) {
  }
}
```

*Figure 7.4: Branching pattern corresponding to Figure 7.2 (branching:* 111001*). Each conditional (rectangle) that is evaluated to true (1) is represented with green, each conditional that is evaluated to false (0) is represented with red. Code that is not executed is faded out.*

### 7.1.3 Memory View

Our system traces specified variables during the execution of a program. The tracing results in a sequence of events that are recorded for each work item. In the memory view (see Figure 7.5) read- and write-access to variable "sdata" as well as local and global barriers

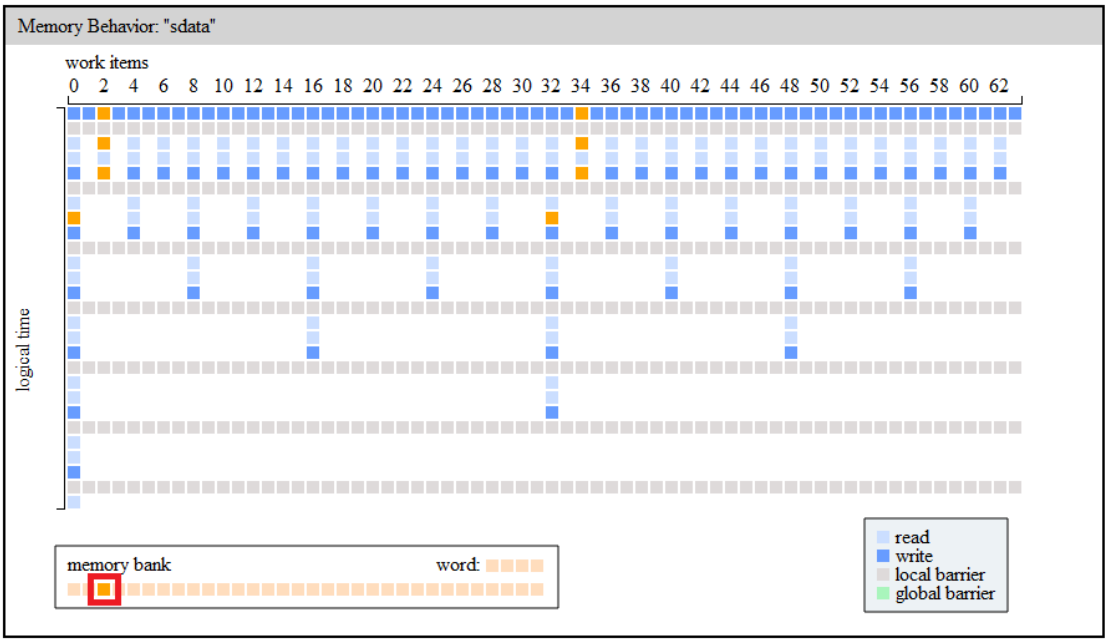



*Figure 7.5: The memory view depicts memory accesses of a specified variable in blue and light blue blocks. An illustration of the memory bank is shown in the lower part of the image. When hovering a memory bank (indicated with red rectangle) the corresponding memory accesses are highlighted in orange.*

constitute events that are visualized with colored blocks. A light blue block represents a read-access, a dark blue blocks a write-access, gray and green blocks represent local and global barriers, respectively. The graph in the lower left part of the visualization shows the memory banks of the local memory. The hardware architecture for the following use cases has 32 memory banks. The x-axis represents the different work items of the execution. The y-axis shows the events, in chronological order from top to bottom.

### 7.1.4 Turnaround Time

Table 7.1 shows the turnaround times of our system, in particular the turnaround time for the parallel reduction implementation version 1 with varying numbers of elements to process. The number of elements also indicate the global work size. The local work group size in this setup is 64, and the maximum number of memory traces is 500.000, and 5.000.000 for branching traces. We need to set the maximum number of traces because the memory of the compute device is statically allocated before the execution. We have measured the performance on a desktop computer using a NVIDIA Quadro 6000, Intel Xeon X5675 @3.06 GHz and 32 GB RAM. The turnaround times include parsing, just-in-time compilation and execution of the program for all cases and additionally the processing of profiling data, view composition and rendering of the visualizations for the cases with instrumentation. The branching view includes the visualizations for different control flows and the memory view

shows visualizations for memory access patterns of a variable. They are further discussed in the following. We only test a small number of elements since this is usually sufficient for analysis due to the repetitive behavior of parallel algorithms. A higher number of maximum traces would increase the turnaround time and the strain on GPU memory.

| Nr Elements | No Instrumentation | Branching View | Memory View | Both Views |
|---|---|---|---|---|
| 256 | ≈ 730 ms | ≈ 1163 ms | ≈ 996 ms | ≈ 1354 ms |
| 2.560 | ≈ 843 ms | ≈ 1263 ms | ≈ 1489 ms | ≈ 1773 ms |
| 25.600 | ≈ 873 ms | ≈ 2680 ms | ≈ 4102 ms | ≈ 5372 ms |

*Table 7.1: Turnaround times for the parallel reduction implementation version 1 (see Chapter 7.3.1) using a local work size of 64. The leftmost column shows the number of elements used in the parallel reduction and also indicates the global work size. The measurements include the entire time, starting with the submission of the program, until the views are completely rendered. The setup uses a maximum of 500.000 memory traces and 5.000.000 branching traces.*

The measurements show that the user is able to experiment with different setups and implementations within a matter of seconds. This facilitates the rapid testing and profiling of competing implementations. The views are flexibly composed during runtime corresponding to the generated profiling data. Our system provides three different views, the debug view, the branching view, and the memory view.

## 7.2 Visual Exploration of Control Flow in Image Processing

A common concept in image processing is stencil computation. These stencil access pattern sweeps over an array and applies operations on a local neighborhood. Accessing the values of neighbors results in challenging control flows, due to the required border handling of the image and the work groups. Issues related to border handling are an inherent part of stencil computations on the GPU, but also frequently occur in other use cases like bricking, tiling, and finite element simulations.

The algorithm we present in the following is an optimized implementation of a stencil computation. It is a real world example in the sense that it is part of the Sobel Filter implementation of NVIDIA's OpenCL SDK [NVIe]. The complete conditional structure of the stencil computation is shown in Algorithm 7.1. In this use case we focus on the memory load part of the algorithm. Even though the memory load pattern is fairly complex, we show that our branching view is capable of providing rapid insight into this pattern and the underlying design choices that were made in regard to performance.

### 7.2.1 Algorithm Outline

A naïve way of realizing the memory loading part of stencil computations is to perform a global memory access for every neighbor that is involved in the current computation, as

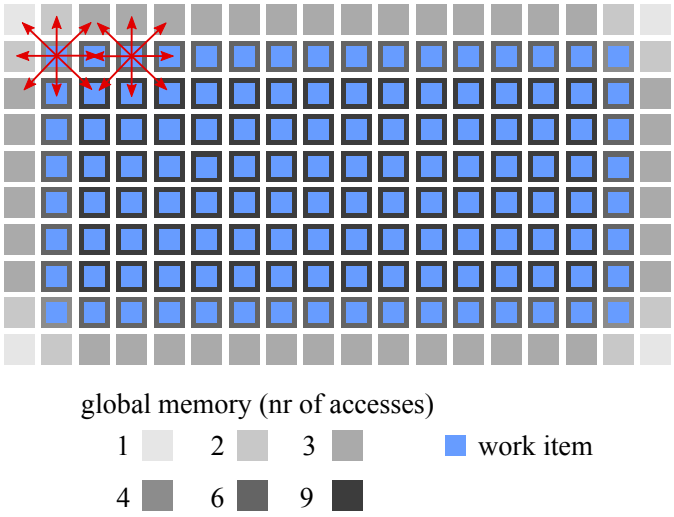



*Figure 7.6: Overlay of a global memory region (gray) with region of work items (blue). The different shades of gray indicate the number of accesses to this memory address for a 8-point stencil. The number includes the additional access of the work item on the memory it overlays to store the stencil computation. The red arrows illustrate the memory accesses to neighbors for two exemplary work items. The local memory region at the border, which is not overlayed with work items is called apron.*

illustrated in Figure 7.6. The figure shows a region of memory (gray) overlayed with work items (blue). The work items in their entirety represent the local work group size [16,8]. The different shades of gray indicate the number of accesses. The red arrows illustrate memory access to neighbors for two exemplary work items. Since different work items perform the memory accesses, many addresses of global memory are loaded multiple times. Redundant access of global memory has a bad impact on performance.

A more efficient way is to load the required data into local memory, preceding the stencil computation. Thus, every address of global memory is only accessed once in a a within work group and the stencil computation is applied on local memory. The loading task is usually executed in parallel by the available work items of the work group with the goal of approaching a uniform workload. Stencil computation usually requires a larger memory region than the size of the work group, therefore a border of memory remains to be loaded. This border is often called the apron or ghost pixels. In this example we use a stencil kernel with a radius of one, hence an apron of one memory block remains, as shown in Figure 7.6.

The loading of memory that lies in the apron introduces several control flow statements in the code and leads to branch divergence and idle work items. In order to keep branch divergence minimal, the algorithm uses a memory loading pattern that bundles the same control flow in separate warps whenever possible.

*Algorithm 7.1: Conditional structure of the implementation of the Sobel Filter example of NVIDIA's OpenCL SDK [NVIe].*

```
1  kernel sobel_kernel(read_only image img, global float *out, int iDevImageHeight,
       int iImageWidth) {
2      int iImagePosX = get_global_id(0);
3      int iDevYPrime = get_global_id(1) - 1;
4      if((iDevYPrime > -1) & (iDevYPrime < iDevImageHeight) & (iImagePosX <
           iImageWidth)){
5      }
6      else  {
7      }
8      if (get_local_id(1) < 2) {
9          if (((iDevYPrime + get_local_size(1)) < iDevImageHeight) & (iImagePosX <
               iImageWidth)) {
10         }
11         else {
12         }
13     }
14     if (get_local_id(0) == (get_local_size(0) - 1)) {
15         if ((iDevYPrime > -1) & (iDevYPrime < iDevImageHeight) & (get_group_id(0)
               > 0)) {
16         }
17         else {
18         }
19         if (get_local_id(1) < 2) {
20             if (((iDevYPrime + get_local_size(1)) < iDevImageHeight) & (
                   get_group_id(0) > 0)) {
21             }
22             else  {
23             }
24         }
25     }
26     else{
27         if (get_local_id(0) == 0) {
28             if ((iDevYPrime > -1) & (iDevYPrime < iDevImageHeight) & ( (
                   get_group_id(0) + 1) * get_local_size(0) < iImageWidth)) {
29             }
30             else {
31             }
32             if (get_local_id(1) < 2) {
33                 if (((iDevYPrime + get_local_size(1)) < iDevImageHeight) & ((
                       get_group_id(0) + 1) * get_local_size(0)) < iImageWidth)  {
34                 }
35                 else {
36                 }
37             }
38         }
39     }
40     if((iDevYPrime < iDevImageHeight) & (iImagePosX < iImageWidth)) {
41     }
42 }
```

Figure 7.7a shows that the initial load of memory has one block offset in comparison to the naïve approach. The naïve approach would need separate border handling for the top part of the apron and bottom part. Using the above mentioned offset eliminates boarder handling for the top part. The bottom part of the apron now consists of the connected rows, as highlighted in orange in Figure 7.7b. The highlighted region exactly matches the number of work items within one warp and can be processed without further introduction of conditional statements. This reduces the occurrences of branch divergence in the execution of the algorithm. The complete memory copy approach consists of seven steps:

(**a**) Shift the initial naïve mapping of work items to memory by an offset of one to the top, as shown in Figure 7.7a. Each work item then fetches the required data from global memory and loads it into the highlighted local memory region.

(**b**) Using the above mentioned offset enables the last two rows of the work group to process the memory load of the two bottom rows within one warp without introducing new conditionals, as shown in Figure 7.7b (highlighted in orange). This warp fetches the required global memory and loads it into the last two rows of the local memory region.

(**c**) The left and right border of local memory are still remaining. The rightmost column of work items is used to load the leftmost row of local memory without the last two blocks, as shown in Figure 7.7c.

(**d**) Corresponding procedure to (c), just for the left column of work items, as shown in Figure 7.7d.

(**e**) The two blocks currently accessed memory the lower left are loaded by the two active work items on the right side of the work group, as shown in Figure 7.7e.

(**f**) Corresponding procedure to (e), just for the two active work items at the lower left of the work group, as shown in Figure 7.7f.

### 7.2.2 Visual Exploration using the Overview Histogram

A closer look at the overview part of the branching view (see Figure 7.8) shows that the first six bins of the histogram already represent the majority of the work items. Hovering over these bins from left to right creates the pictures presented in Figure 7.9. These images provide insight into the behavior of the program. Comparing the resulting highlights (see Figure 7.9) with the abstract described execution of the algorithm (see Figure7.7) reveals a direct correspondence with the illustrations. For instance, Figure 7.9a shows the highlighted image created by hovering over the first bar of the histogram, which is the branching executed by the majority of work items. Examining a red block in the image, shows that this block reflects the same pattern as shown in Figure 7.7a. In particular, the load of a big block of memory and the gap of two blocks in between. The highlights of thinner lines in Figure 7.9a correspond to the loading of the lower two rows of local memory, as shown in Figure 7.7a. Similar correspondences also occur in the remaining figures.

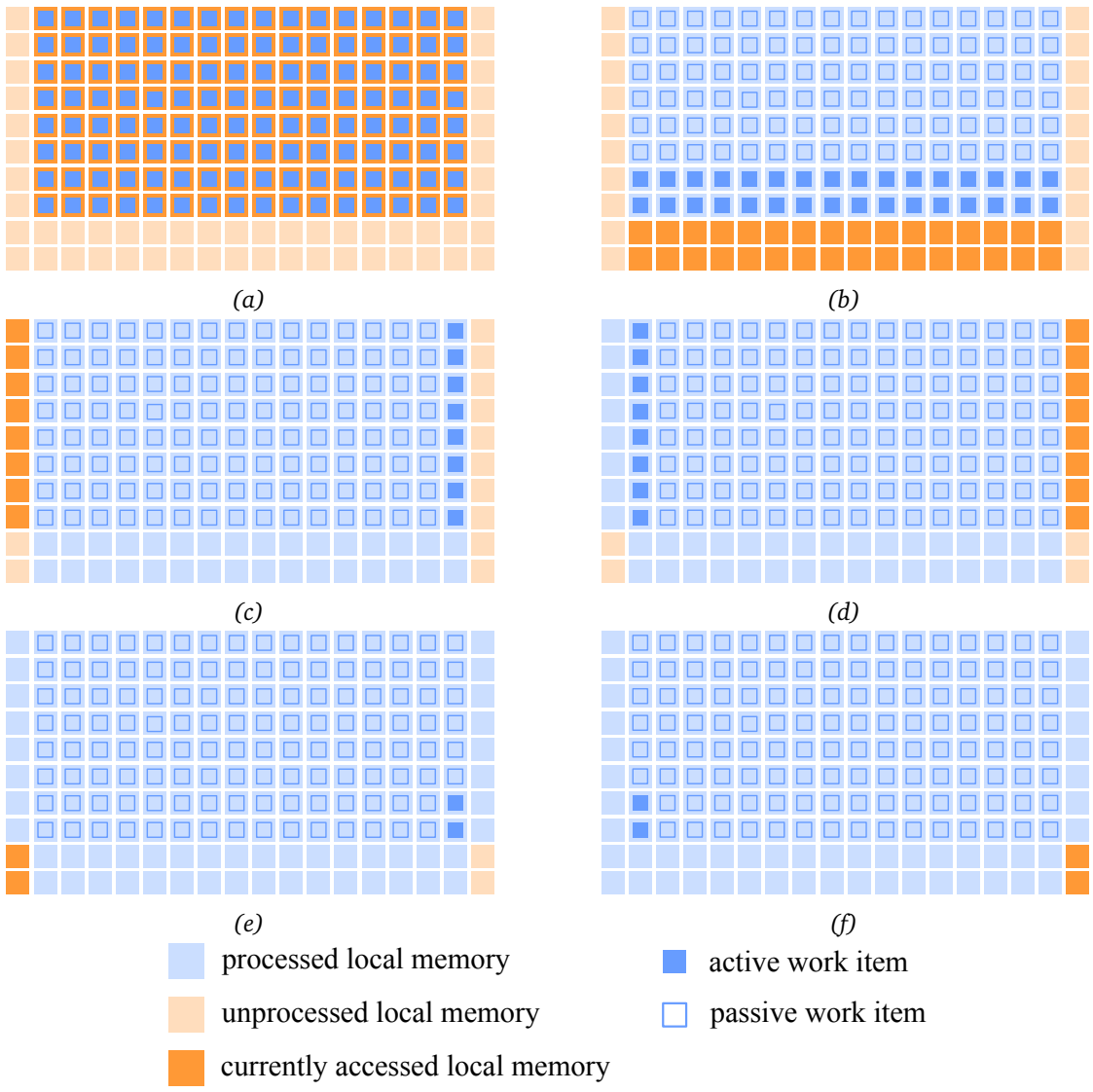



*Figure 7.7: Outline of an efficient pattern to load a region of local memory using the work items that are required for a stencil computation. The size of the work group is [16,8] and the region of the local memory is [18,10]. Steps (a) through (f) illustrate the access pattern that is used to load the memory.*

Our example demonstrates that, in this case, the visualization provides insight into the behavior of the algorithm. With the first examination of the overview, the user gains a "*feeling*" for the behavior of the algorithm. It facilitates the understanding of an unknown implementation, as well as the debugging of a known implementation to find erroneous code. Further, it aids in rapidly iterating over variations of the algorithm.

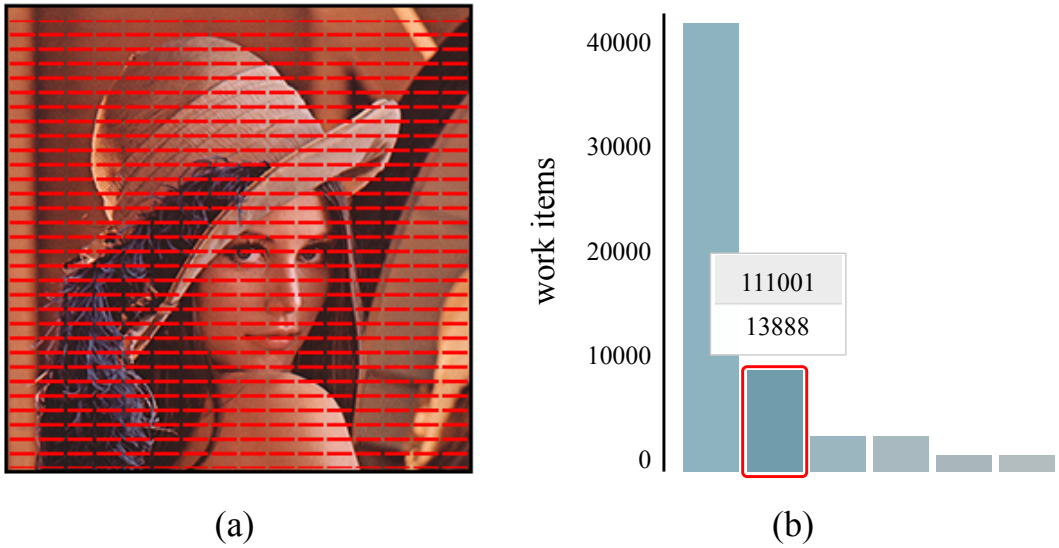



*Figure 7.8: Overview of the control flow in an image processing example. Each bar of the histogram depicts a different control flow. The height of the separate bars indicate the number of work items that execute a certain branching pattern. Hovering over a bar, as highlighted in the image, colorizes the corresponding pixels in the image. It also shows the label of the bar with the binary encoded branching pattern (top of label) and number of work items that execute this branching pattern (bottom of label).*

### 7.2.3 Visual Exploration using Proxy Warps

Proxy warps support the investigation of occurring branch divergence. In this analysis, we use various work-group sizes ([4, 8], [8, 8], [16, 8] and [32, 8]) for the execution of the algorithm to compare different results of proxy warps. The chosen sizes vary in the first dimension, due to the reason that these changes stronger influence the mapping of warps on work items. Warps are mapped horizontally along the first dimension. If the first dimension of the work-group size is not a multiple of 32, some warps have to be split to fit into the work group.

Figure 7.10 and 7.11 show the resulting proxy warps for the different work-group sizes. The number of color-coded regions of each proxy warp directly show the number of containing branching patterns. The first few proxy warps are usually the most important ones, since they are the ones that represent the largest number of warps. This number is depicted in the middle of the proxy warp representation. The proxy warps that contain the majority of warps have in most cases the greatest influence on performance, hence they are sorted accordingly in the visualization.

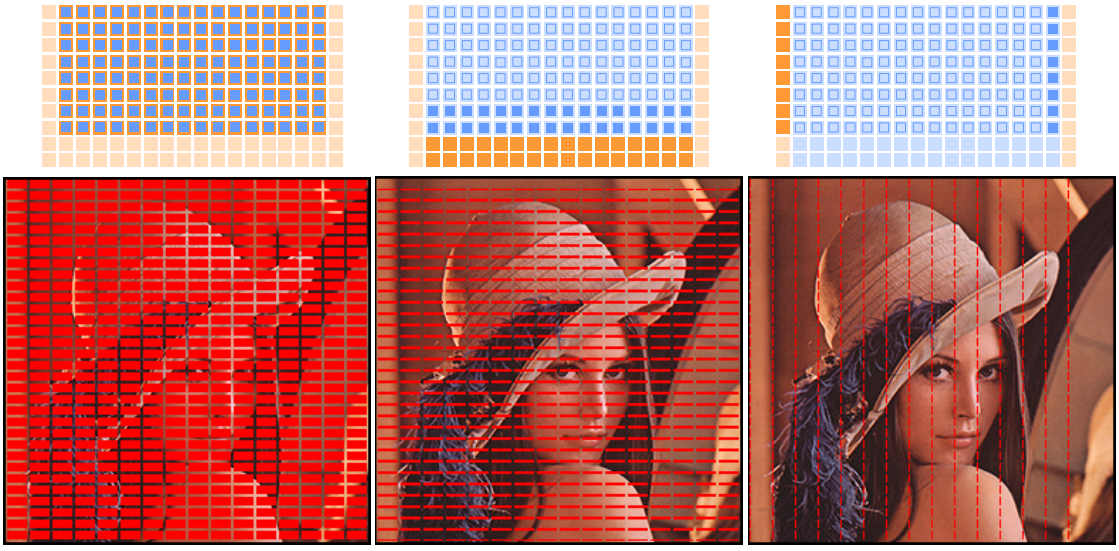
Figure 7.10a shows the different proxy warps for a work-group size of [4, 8]. The majority of the warps (1860) are represented by the first proxy warp. It contains six different regions (six different colors of the donut-chart), which represent six different distinct branching

patterns. Consequently, 1860 warps diverge and all of their work items have to execute the six different branching patterns, potentially leading to a poor performance. Using the code highlighting, shows that the different branching pattern occur due to different border handling of the local memory region. Increasing the work-group size to [8, 8] (see Figure 7.10a) already leads to an improvement, so that the largest proxy warp with 960 warps only contains three different branching patterns. The second biggest proxy warp with 900 containing warps though still has four distinct branching patterns. Increasing the work-group size to [16, 8] and [32, 8] shows only minor improvements.

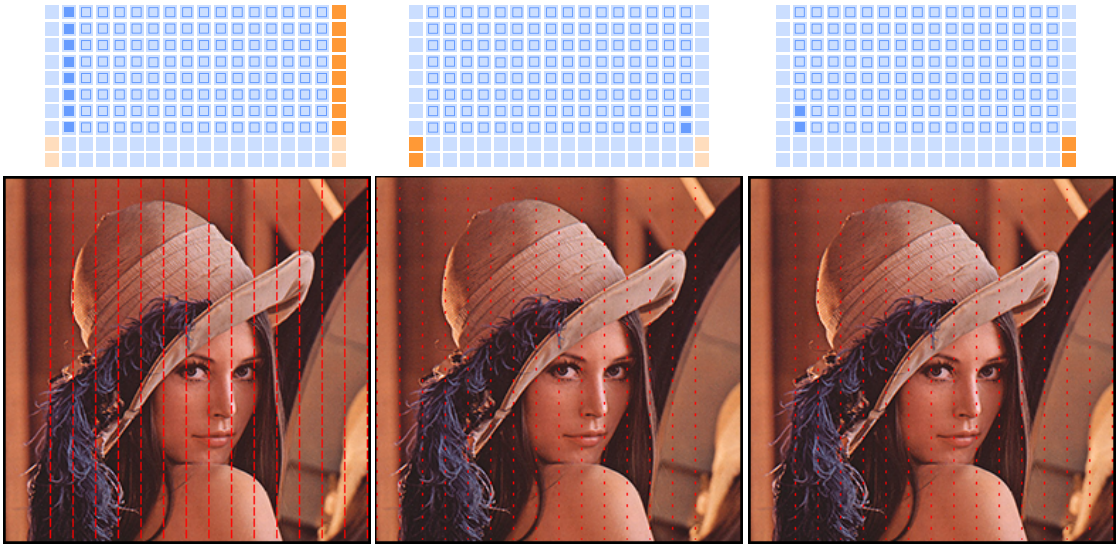
### 7.2.4 Discussion

Our first use case has shown that the branching histogram linked to the input image and the source code view, provides rapid insight into the behavior of the algorithm (evaluation criterion 1). With only little knowledge about the algorithm the user can understand how the border handling is realized. The highlights in the image depict the different control flow patterns and map directly to the above described outline of the algorithm (see Chapter 7.2.1). By linking the corresponding patterns to the code the user additionally understands how the behavior is realized in the implementation.

Furthermore, the investigation of the proxy warps has shown that an increase of the work group size reduces the number of warps that suffer from branch divergence (evaluation criterion 2). Nevertheless, performance analysis is always a multi-objective problem. Increasing the work-group size may decrease branch divergence, but also influences the memory throughput. The size of the local memory has to change according to the work-group size. The provided tools are supposed to be used in combination with existing tools that provide further information about performance. In the next example we demonstrate how the proxy warp visualization works in combination with the memory view.

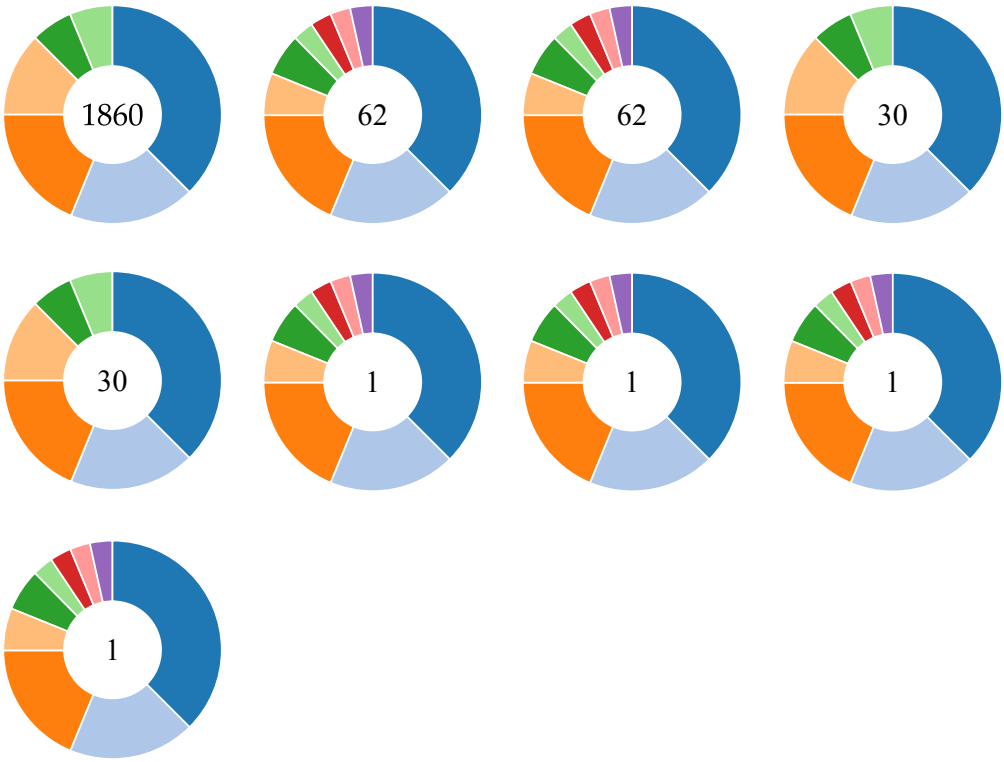


*(a)* *(b)* *(c)*

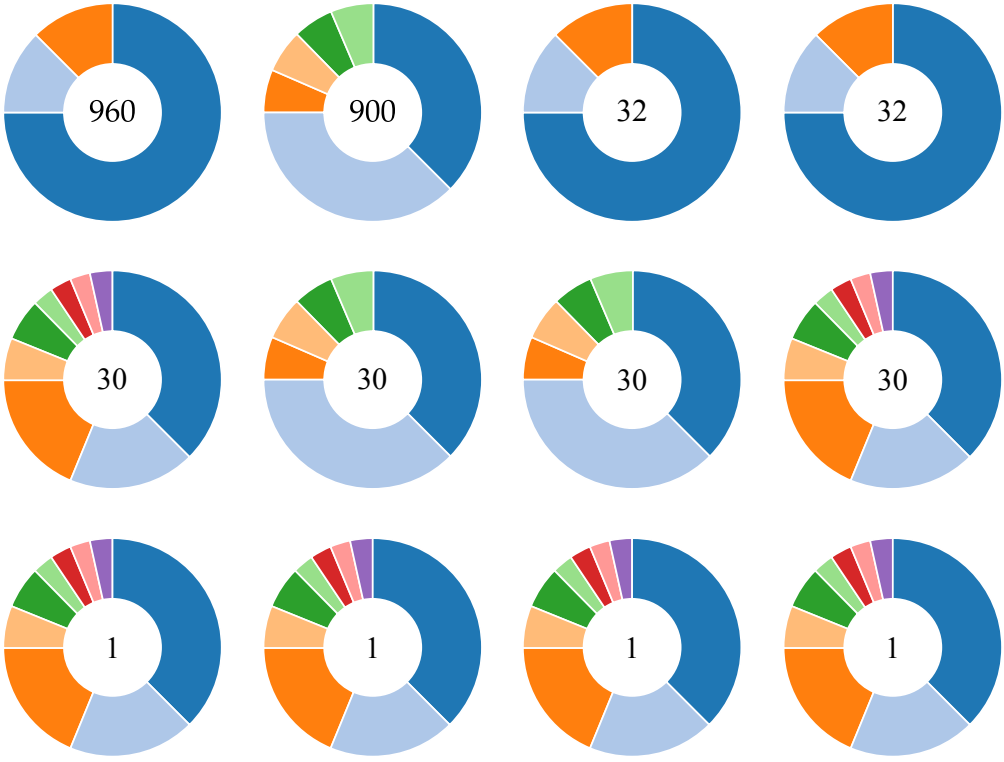


*(d)* *(e)* *(f)*

*Figure 7.9: Visual exploration of branching behavior using an image processing example. Red highlighted areas show a distinct branching pattern. The highlights provide a rapid insight into the different control flows of the implementation and lead to a better understanding.*



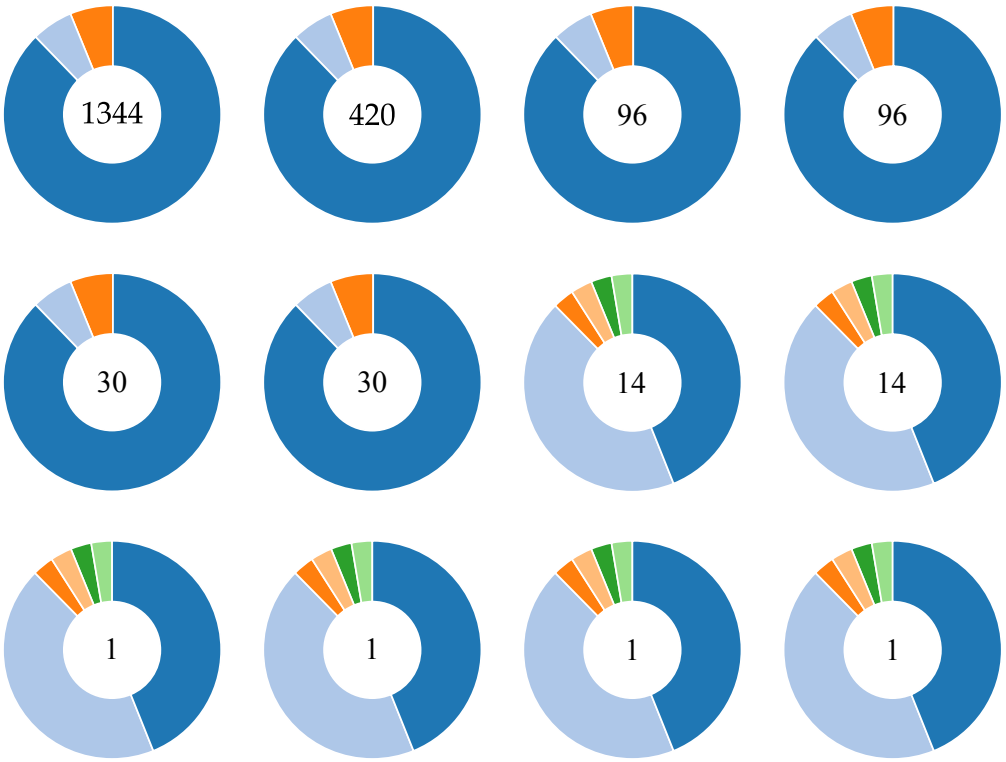

*(a) Proxy warp result using a work-group size of [4,8].*



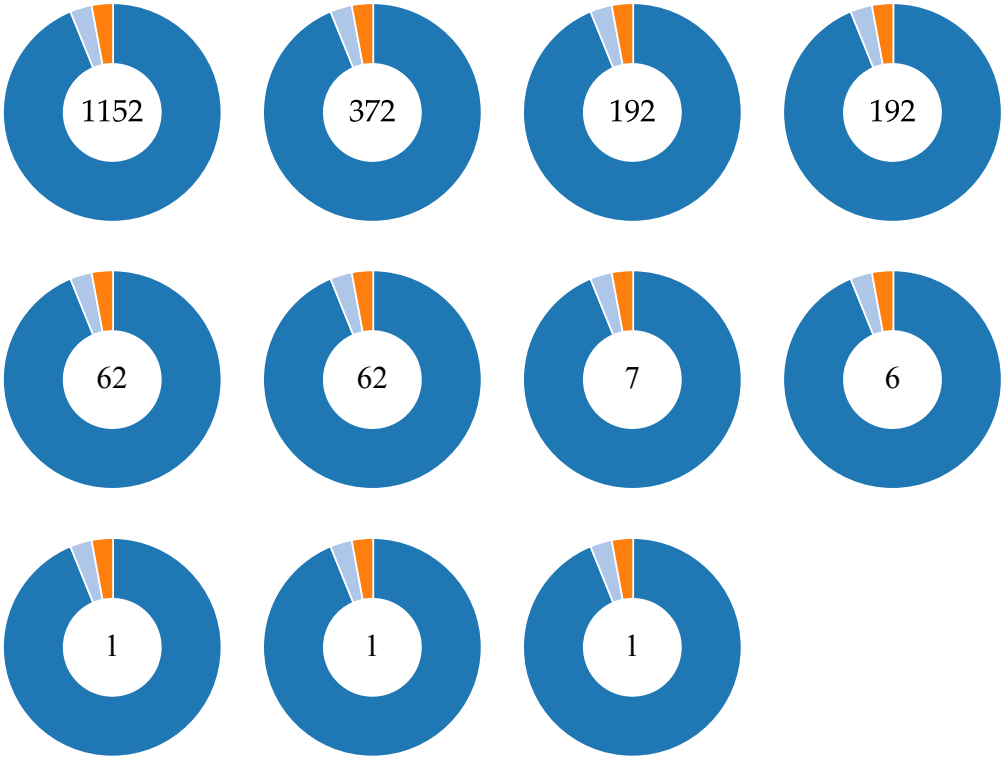

*(b) Proxy warp result using a work-group size of [8,8].*



*Figure 7.10: The illustration shows the resulting proxy warps in the case of an image processing example. Increasing the work group size from [4,8] (a) to [8,8] (b) shows that the largest proxy warp in each image (the first one), decreases from five distinct branching patterns to three (shown by the color encoded regions of a proxy warp). However, visualization (b) still a large proxy warp containing 900 warps with five distinct branching patterns.*




*(a) Proxy warp result using a work-group size of [16,8].*




*(b) Proxy warp result using a work-group size of [32,8].*

*Figure 7.11: The illustration shows the resulting proxy warps in the case of an image processing example. Increasing the work-group size from [16,8] (a) to [32,8] (b) shows only minor improvements in terms of branch divergence.*

## 7.3 Visual Exploration of Parallel Reduction Implementations

Parallel reduction is a well understood algorithm and optimizations for GPUs are well documented [Har07, AMDc]. Although reduction is a fairly simple concept, parallel implementations thereof are much harder to comprehend. The algorithm is frequently used in education as an example to explain GPU optimization strategies.

In this case study we demonstrate the usage of our framework for the visual exploration of different implementations of parallel reduction. We show how the visual exploration can reveal bank conflicts leading to a better understanding of optimization strategies. With the use of our source code annotations, we investigate the memory accesses of one variable during the execution.

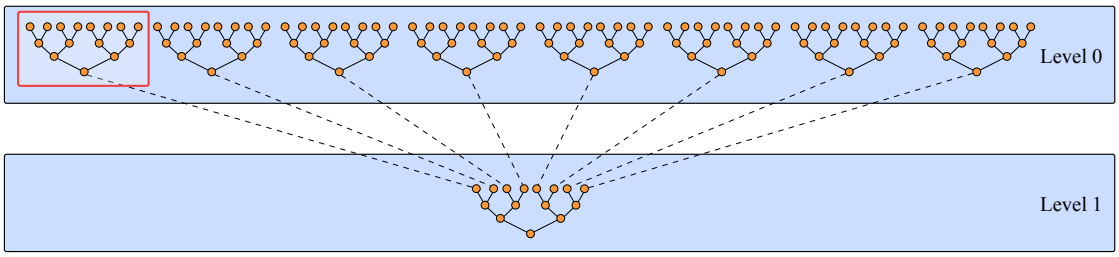



*Figure 7.12: Kernel decomposition splits the result in several invocations of the kernel. This decomposition is applied to achieve a global synchronization during the computation. Our visualizations of the reduction algorithm show only the first work group of the first invocation level, as highlighted above, since they all show the same behavior.*

The examples presented in the following are ported versions of the reduction implementations of NVIDIA's OpenCL SDK [NVIe]. We use a work-group size of 64 and a global work size of 1024, which results in 32 warps. The illustrations presented in Figure 7.15 and Figure 7.17 show only the first work group. Since parallel reduction exhibits a highly repetitive access pattern the visualization of a small subset of work items is sufficient. The parallel reduction implementation uses kernel decomposition (see Figure 7.12), which splits the result in several invocations of the kernel. This decomposition is applied to achieve a global synchronization during the computation. The complete execution of the parallel reduction uses several kernel invocations resulting in several pyramidal reduction levels. We only show part of the lowest level of the reduction pyramid in this example, indicated with the red rectangle in Figure 7.12.

The different implementations progressively make changes to the code for performance improvements. The first step is the same for all variations and loads global memory into local memory, to achieve efficient access of subsequent instructions:

```
local float sdata[64];         // allocation of local memory
int tid = get_local_id(0);     // retrieve load index of work item
sdata[tid] = g_idata[i];       // load from global memory into local memory
barrier(CLK_LOCAL_MEM_FENCE); // synchronize local memory within a work group
```

The synchronization barrier at the end ensures that the transfer to local memory is completed before the code continues. After the local memory is loaded the implementations continue differently, as shown in the following.

### 7.3.1 Reduction Version 1

*Algorithm 7.2: Reduction Version 1: This reduction determines which work items are active by using the modulo operator. This operator is very expensive on GPUs, and the interleaved inactivity means that no entire warps are active, which is also very inefficient [Har07].*

```
1  kernel reduce0(global float *g_idata, global float *g_odata, int n)
2  {
3      local float sdata[64];
4      @watch sdata;
5      // load shared mem
6      int tid = get_local_id(0);
7      int i = get_global_id(0);
8      if(i < n) {
9          sdata[tid] = g_idata[i];
10     }
11     else {
12         sdata[tid] = 0;
13     }
14     barrier(CLK_LOCAL_MEM_FENCE);
15     for(int s = 1; s < get_local_size(0); s = s*2) {
16         // modulo arithmetic is slow!
17         if ((tid % (2*s)) == 0) {
18             sdata[tid] = sdata[tid] + sdata[tid + s];
19         }
20         barrier(CLK_LOCAL_MEM_FENCE);
21     }
22     // write result for this block to global mem
23     if (tid == 0) {
24         g_odata[get_group_id(0)] = sdata[0];
25     }
26 }
```

The first version of the reduction (see Algorithm 7.2) is implemented using a loop with interleaved memory addressing. In the first iteration step, every second work item handles the first reduction step by reducing two elements of the local memory. In the next iteration step, every fourth work item reduces two elements from the previous result. The following iteration steps repeat this step with every 8th, 16th and 32nd work item, respectively. The usage of only every n-th work item and the interleaved addressing of the memory becomes clearly visible in our visualization of memory accesses Figure 7.15a.

The algorithm uses six iterations for our case of a work group size of 64. The memory accesses of the six different iterations are directly visible in our memory visualization through the six different regions in between each row of synchronization barriers. Our proxy warp

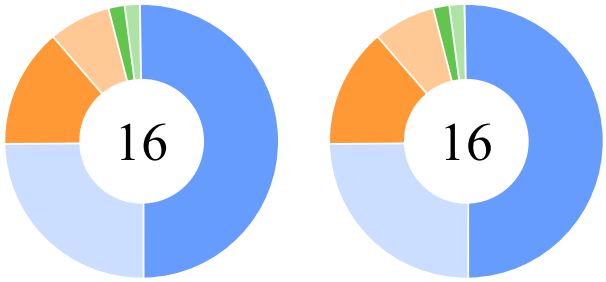



*Figure 7.13: Proxy warps for parallel reduction version 1 with a work group size of 64.*

visualization also reflects the iterations with six distinct regions, as shown in Figure 7.13. The proxy warps have the same visual appearance, however, hovering over the dark green bar reveals that one proxy shows different evaluation of the last conditional in the code:

```
// write result for this work group to global memory
if (tid == 0) { // only for local id = 0
    g_data[get_group_id(0)] = sdata[0];
}
```

This part of the code writes the result of the reduction back into global memory. Only the first work item of each work group executes this task. Due to this reason, the two warps of every work group (see Figure 7.13) show a different branching pattern.

Furthermore, this view reveals that all 32 warps suffer from branch divergence. Increasing the work group size does not change this issue, only results in slightly different proxy warps. Figure 7.14 shows this for a work group size of 128.

*Algorithm 7.3: Reduction Version 2: This version uses contiguous work items, but its interleaved addressing results in many shared memory bank conflicts [Har07].*

```
1  kernel reduce1(global float *g_idata, global float *g_odata, int n)
2  {
3      local float sdata[64];
4      @watch sdata;
5      // load shared mem
6      int tid = get_local_id(0);
7      int i = get_global_id(0);
8      if(i < n) {
9          sdata[tid] = g_idata[i];
10     } else {
11         sdata[tid] = 0;
12     }
13     barrier(CLK_LOCAL_MEM_FENCE);
14     // do reduction in shared mem
15     for(int s = 1; s < get_local_size(0); s = s*2) {
16         int index = 2*s*tid;
17         if (index < get_local_size(0))  {
```

```
18              sdata[index] = sdata[index] + sdata[index + s];
19          }
20          barrier(CLK_LOCAL_MEM_FENCE);
21      }
22      // write result for this block to global mem
23      if (tid == 0) {
24          g_odata[get_group_id(0)] = sdata[0];
25      }
26  }
```

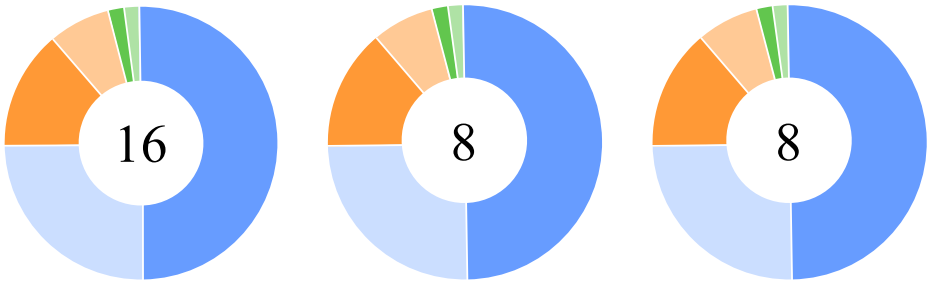



*Figure 7.14: Proxy warps for parallel reduction version 1 with a work group size of 128.*

The interleaved addressing always results in branch divergence. However, hovering the individual regions reveals an interesting aspect. For half of the work items (blue region) the conditional in the code is always false:

```
if ((tid % (2*s)) == 0) {
    sdata[tid] = sdata[tid] + sdata[tid + s];
}
```

and for all remaining work items (remaining regions) it is true:

```
if ((tid % (2*s)) == 0) {
    sdata[tid] = sdata[tid] + sdata[tid + s];
}
```

This statement where the conditional is false corresponds exactly to the work items (every second) that appear inactive in Figure 7.15a. This leads to the idea behind the next version of the algorithm, in particular, to pack this branching behavior into one warp instead of spreading it over all warps.

### 7.3.2 Reduction Version 2

This version of the parallel reduction (see Algorithm 7.2) replaces the statement in the inner loop that causes the branch divergence. The idea is to change the code, such that the same branching behavior is packed into the same warp. The resulting visualization in Figure 7.16 shows that with this version of the reduction only half of the warps suffer from branch divergence. The gray proxy warp shows warps that do not suffer from branch

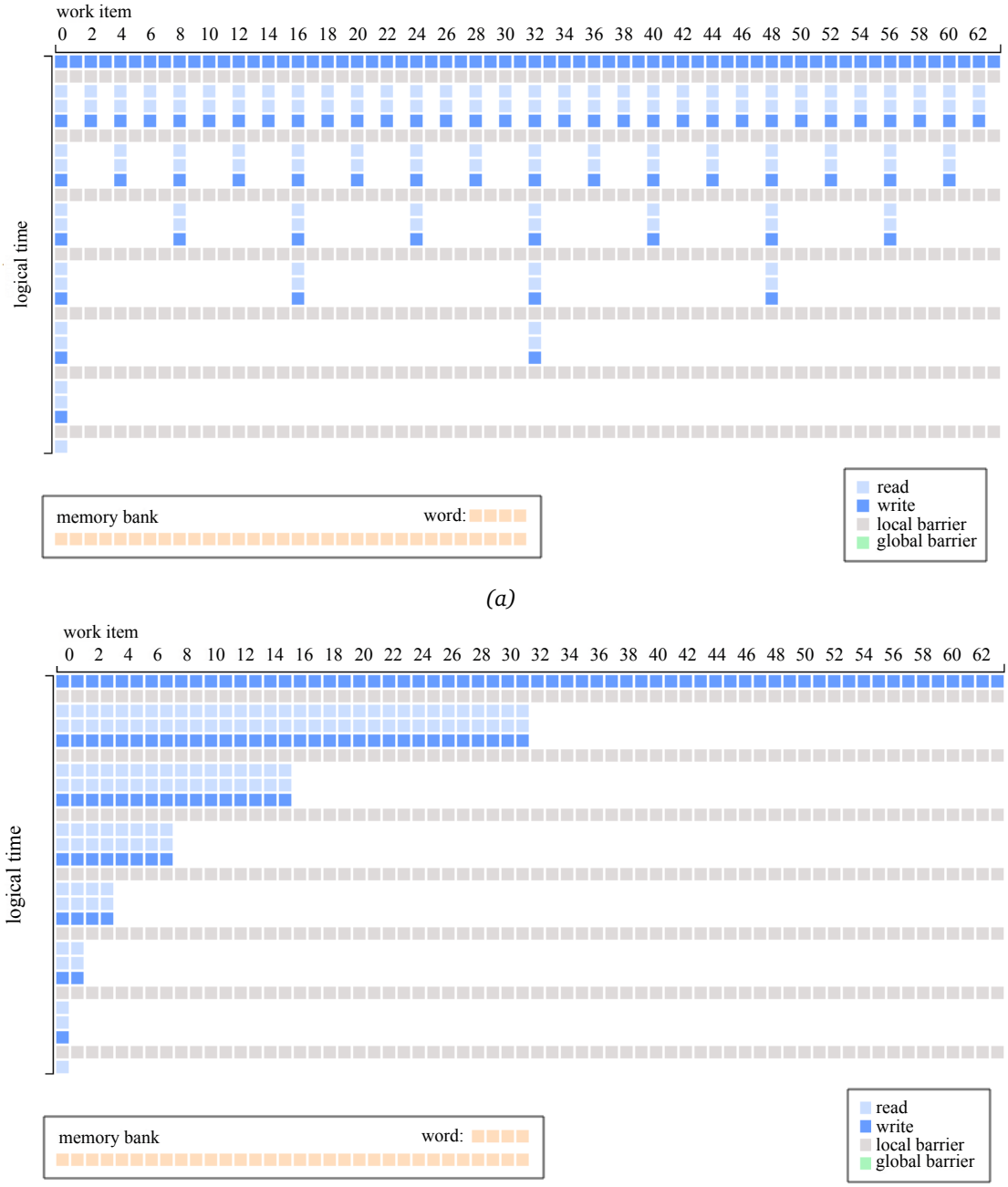



*Figure 7.15: Two implementations of the parallel reduction algorithm: The visualization in image (a) shows that interleaved inactivity of work items occurs. Image (b) shows that work items of one warp are either all active or all inactive.*

divergence. The work group size is too small to completely pack the same branching pattern into separate warps. Increasing the work group size again to 128 shows no more resulting proxy warps, meaning branch divergence is completely removed. The access structure shown in Figure 7.15b also directly visually indicates a more compact distribution of the memory accesses in regard to different work items.

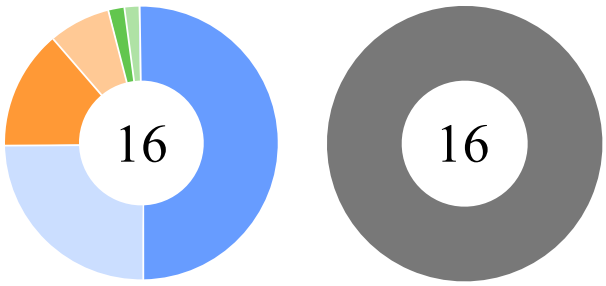



*Figure 7.16: Proxy warps for parallel reduction version 2 with a work group size of 64.*

*Algorithm 7.4: Reduction Version 3: This version uses sequential addressing - no divergence or bank conflicts occur [Har07].*

```
1  kernel reduce2(global float *g_idata, global float *g_odata, int n)
2  {
3      local float sdata[64];
4      @watch sdata;
5      // load shared mem
6      int tid = get_local_id(0);
7      int i = get_global_id(0);
8      if(i < n) {
9          sdata[tid] = g_idata[i];
10     } else {
11         sdata[tid] = 0;
12     }
13     barrier(CLK_LOCAL_MEM_FENCE);
14     // do reduction in shared mem
15     for(int s = get_local_size(0)/2; s > 0; s = s/2) {
16         if (tid < s) {
17             sdata[tid] = sdata[tid] + sdata[tid + s];
18         }
19         barrier(CLK_LOCAL_MEM_FENCE);
20     }
21     // write result for this block to global mem
22     if (tid == 0) {
23         g_odata[get_group_id(0)] = sdata[0];
24     }
25 }
```

However, this implementation may have removed branch divergence, but as mentioned above, performance optimization is always a multi-objective problem. The changes in the code

introduce memory bank conflicts. Using the memory view and hovering over a memory bank field, as indicated in Figure 7.17a with an arrow, reveals several bank conflicts. Each level between two synchronization points has two read and two write accesses to the same memory bank. The next version of the parallel reduction algorithm deals with this issue.

### 7.3.3 Reduction Version 3

The previous implementations use an interleaved indexing of the local memory, which consequently leads to bank conflicts. The third implementation of the parallel reduction (see Algorithm 7.4) uses sequential addressing instead. The iteration loop is changed in order to base the indexing of the local memory on the work item id. Since only indexing is changed, the static image of our memory view does not change, shown by the comparison of Figure 7.17a and Figure 7.17b. However, hovering over the first memory bank reveals that the bank conflicts are resolved. The highlighted memory accesses in Figure 7.17b do not occur concurrently.

### 7.3.4 Discussion

This case study shows the visual exploration with the combination of the memory and branching view on the basis of competing implementations of the parallel reduction algorithm. The investigation of the first version of parallel reduction clearly shows the arising issue of branch divergence (evaluation criterion 2). We also show that the code branching view in combination with the code highlighting provides significant hints to the source of the performance issue (evaluation criterion 1) and even indicates ideas for improvement. The performance increase is also directly reflected by comparing Figure 7.15a and Figure 7.15b (evaluation criterion 3). Figure 7.18 shows the actual performance of the three different version of the parallel reduction. The setup uses 256.000.000 elements for the reduction and we have measured the performance on a desktop computer using a NVIDIA Quadro 6000, Intel Xeon X5675 @3.06 GHz and 32 GB RAM. The results show that the work-group size has the most significant influence on performance. The measurements indicate a consistent performance difference of the three different versions of parallel reductions over all tested work-group sizes. However, the performance compared between version 1 and version 2 shows the largest difference. Tests with different numbers of elements (25.600.000 and 256.000) show the same order of difference between the implementations.

## 7.4 Conducting of a User Study

In order to further evaluate our system we plan to conduct a quantitative user study to show that our system supports the understanding of parallel programs. The actual conduction would transcend the scope of this thesis but we describe the basic design here. Although, our system is broadly applicable for understanding, debugging, profiling and presenting parallel programs, we focus in our study design on educational scenarios where our system is used for educational purposes to explain well-known optimization strategies for parallel programs.

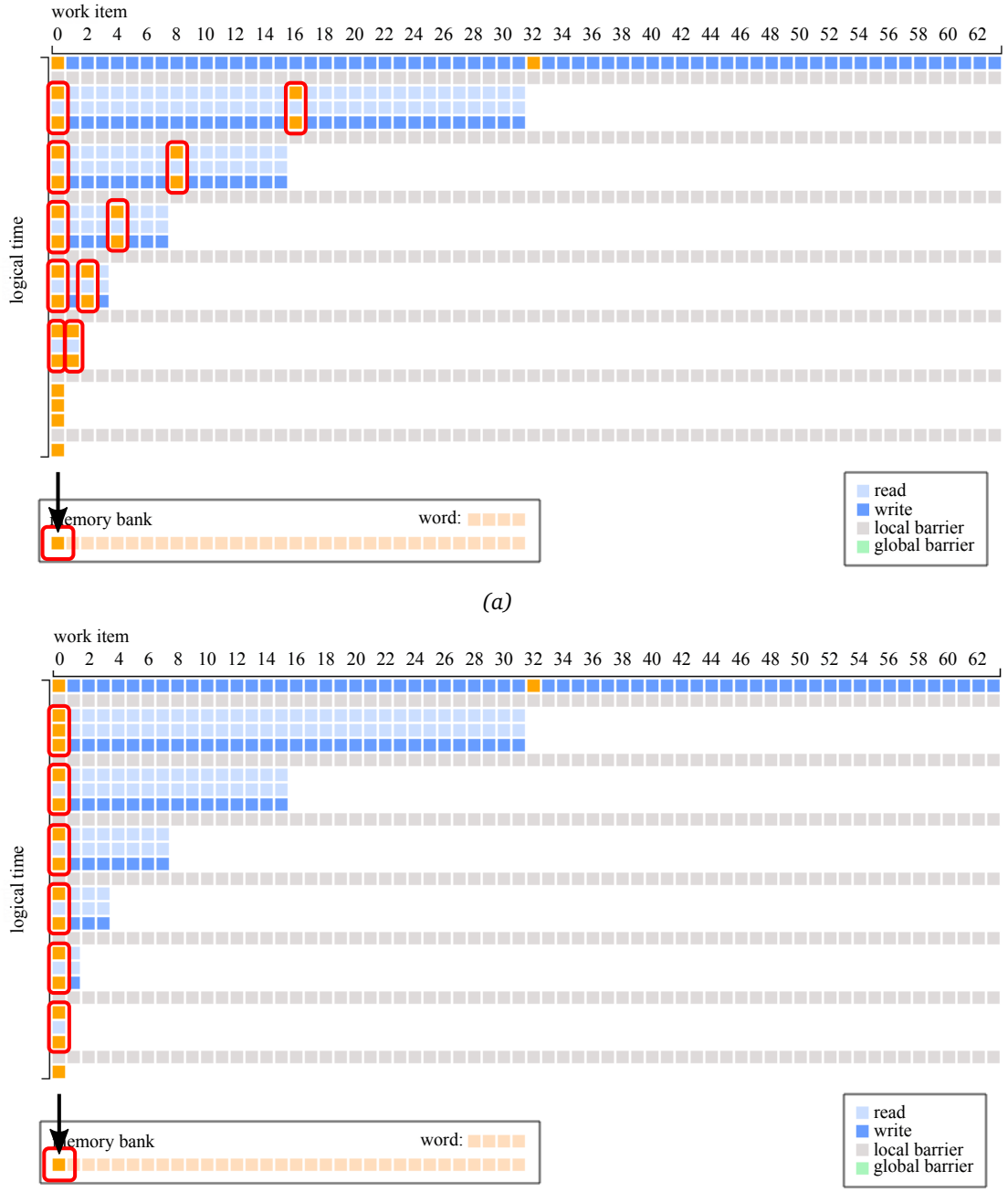



*Figure 7.17: The visualization of the two implementations result in the same image. However, the interactive visualization reveals that the implementation shown in (a) results in bank conflicts while sequential addressing in (b) resolves these conflicts.*

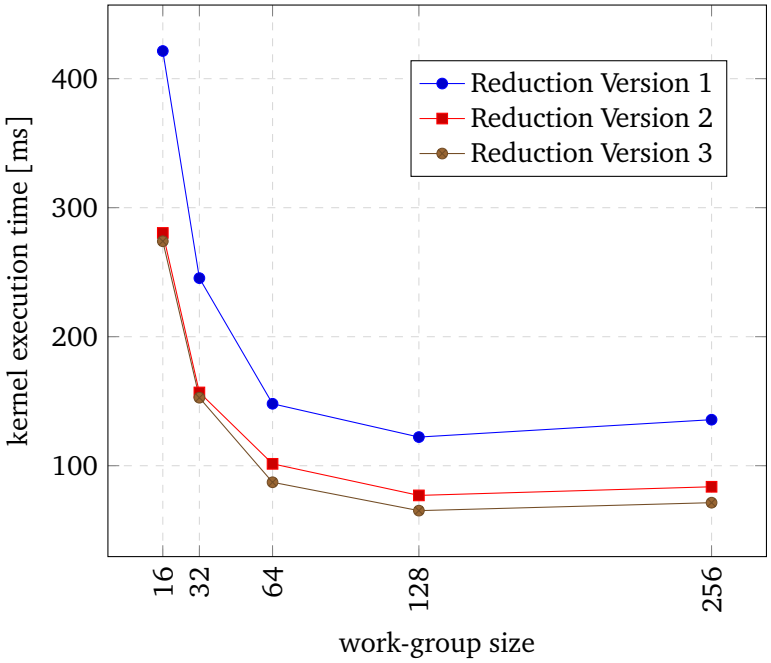



*Figure 7.18: The kernel execution time in ms for different implementations of parallel reduction using 256.000.000 elements. The measured time only includes the first invocation of the kernel (see Figure 7.12).*

### 7.4.1 Hypotheses

Our hypotheses for the user study are:

**H.1** Participants will be able to understand parallel programs more efficiently. We assume that our visualizations provide fast insight into the program's behavior and hence improve the understanding of the algorithm.

**H.2** Participants will be able to understand the relation between the implementation and the performance. We assume that the linking between the source code and our visualizations will improve the understanding of the performance impact of competing implementations.

### 7.4.2 Participants and Apparatus

Since the targeted user group of our systems lies in education we plan to achieve a possible large number of participants by recruiting computer scientists students, preferably balanced in gender and progression of study. We require that all participants have normal color vision (not color-blind) and are familiar with OpenCL to have the minimal background knowledge to participate in the study. The user study should be conducted using a desktop computer or laptop with sufficient GPU capabilities.

### 7.4.3 Tasks and Procedure

A set of different implementations of the same algorithm will be shown and the participants will be asked perform different tasks. After each task, the participants should fill-out a questionnaire with several questions where they are supposed to respond on a 7-point Likert scale. In order to measure the subjective workload we take the first six questions from NASA Task Load Index [HS88]:

- How mentally demanding was the task?
  (1 = very low, 7 = very high)
- How physically demanding was the task?
  (1 = very low, 7 = very high)
- How hurried or rushed was the pace of the task?
  (1 = very low, 7 = very high)
- How successful were you in accomplishing what you were asked to?
  (1 = perfect, 7 = failure)
- How hard did you have to work to accomplish your level of performance?
  (1 = very low, 7 = very high)
- How insecure, discouraged, irritated, stressed and annoyed were you?
  (1 = very low, 7 = very high)
- How mentally demanding was the task?
  (1 = very easy, 7 = very hard)
- How easy was it to understand the algorithm?
  (1 = very easy, 7 = very hard)
- How easy was it to comprehend the design choices of the implementation?
  (1 = very easy, 7 = very hard)
- How easily were you able to relate between the implementation of the program and corresponding performance?
  (1 = very easy, 7 = very hard)

#### Experimental Design

The study is a mixed between- and within-subject design with the tools (our system or a simple code editor) and difficulties (complexity of the presented algorithm) as factors. The tool is a between-subject factor, which means that we provide both tools to every participant for the execution of the tasks. The difficulty is a within-subject factor since we do not want to present the same algorithm twice to the same participant. The outcome would be influenced

since the participant is already familiar with the algorithm from the first investigation (practice effect). Our measurements of each task are accuracy (correct or incorrect), efficiency (completion time), and the subjective responses to our questionnaire.

CHAPTER 8

# Conclusion and Future Work

The utilization of the GPU for general parallel computations and the provision of general interfaces to parallel hardware, as provided by OpenCL and CUDA, has functioned as catalyst for parallel programming. The parallel-computing paradigm has also become increasingly prominent in many research domains. However, parallel programming is still a complex and tedious task due to its concurrent nature. The implementation of efficient parallel programs requires deep knowledge about the underlying hardware. Additionally, the execution of parallel programs is often seen as black box, where only input and output are known, but only little is displayed about the execution of the code.

During the course of this thesis, we have presented a novel approach that uses visual explorations to reveal the inner behavior and execution of parallel programs. Our contribution comprises three components:

(1) the definition of a DSL for writing kernel code,

(2) a source-to-source compiler that performs just-in-time compilation of the DSL to fully instrumented OpenCL code, and

(3) an interactive visualization explorer based on the D3 visualization framework.

We have developed a DSL that provides support for fast prototyping of parallel programs and additionally offers annotations for a task-specific investigation of the execution. The DSL is embedded in our framework, which lets us hide the common setup and boilerplate code that is usually required from the user. Data transfer between host and compute devices is also handled automatically and does not appear in the DSL. This helps the user to focus on the implementation of the algorithm and its investigation. Furthermore, the user benefits from the fast turnaround time of our language, which is a combination of an interpreted and just-in-time compiled language. The presented code annotations serve as a simple method

for a specific selection of objectives the user wants to explore. These may be memory access patterns, diverging control flow or any kind of manually specified data. The DSL makes it possible to link the data directly to our visualization system.

The basis of our DSL is a source-to-source compiler that interprets the code of the user and compiles and executes an OpenCL kernel. The compiler translates the code annotations into fully instrumented OpenCL code. This enables to produce traces ,e.g. of memory accesses or the evaluation of conditionals, during the execution of the program. However, the instrumentation of source code has the disadvantage that it does not take into account any optimization made by the OpenCL compiler, which we discuss further in the following future work.

We have presented two visualization views that facilitate the exploration of memory accesses, as well as the exploration of control flow. In order to link our views with the source code, we perform a static analysis of the code that creates a semantic model containing its structure. Our memory view encodes software, synchronization, as well as the hardware architecture. We have shown that the connection of the execution and its interaction with the hardware provides useful insights into the understanding and the performance of the investigated implementation. Furthermore, we have shown that the aggregation of similar behavior in a unified visualization achieves a clearer and more structured overview, as in the case of proxy warps.

During the evaluation of our approach, we have presented two use cases that demonstrate our system. The first use case features the exploration of control flow using a typical image processing example with stencil computations. Thereby, we have shown that the execution behavior of the program is directly reflected in the our visualizations leading to a better understanding even with little knowledge about the code. The analysis of our visualizations has shown that they depict similar patterns as outlined in the algorithm description. Additionally, we have presented the usage of proxy warps, which offer immediate visual cues about branch divergence occurrences in the execution.

The second use case has illustrated the exploration of memory accesses on the basis of different implementations of parallel reduction. We have demonstrated that the memory accesses are illustrated in a clearly structured overview. Furthermore, we have shown that performance differences of the different parallel reduction implementations are reflected in our visualizations.

### Future Work

In the future, we would like to extend the scalability of our approach with the introduction of new features. For instance, our overview of memory accesses presents every single access. Large numbers of events would overwhelm our visualizations, resulting in a large scrollable area. We would like to overcome this limitation by either implementing new overview views, or by meaningful aggregations of events.

We intentionally based our approach on user-defined traces through instrumentation, although, the information of hardware counter and hardware samples would be beneficial to complement input for our tools. Thereby the user could define own traces of events. This would make our approach extendable regarding the recording part of the application. Our approach of user defined data extraction currently supports only simple arrays. It would be advantageous to provide the means for more complex structures.

The instrumentation on the source code level of OpenCL limits our approach in two ways. First, we do not consider changes the OpenCL compiler performs to the code, e.g. through optimizations. This could possibly influence memory accesses and control flow. Second, our approach only supports OpenCL. We would like to extend our system to support CUDA, which requires either the implementation of a new parser with including instrumentation, or the usage of existing instrumentation tools. The instrumentation could also be applied on a binary-level using Parallel Thread Execution (PTX). PTX is the virtual instruction set targeted by NVIDIA's CUDA compiler, as well as OpenCL compilers.

Furthermore, we want to extend our approach from specializing on the investigation of a single kernel to the investigation of multiple kernels. Visualizations that encode the behavior of multiple kernels and their interaction could provide useful insight. Another interesting aspect is the investigation of the usage of multiple compute devices in one application and additionally the investigation of the program execution on GPU clusters instead of on single GPUs.

An interesting idea that emerged during this work is the utilization of our visual exploration approach for educational purposes. This puts more emphasize on the understanding part our system provides. We would need to extend our visualizations in a sense that also "simpler" interactions of the algorithm is visualized. Our use cases partially point in this direction. For instance, the parallel reduction use case provides a comparison of competing implementations. Additionally, the image filtering use case illustrates the required steps that are taken to handle memory accesses of stencil computations. An extension of the second use case for education could let the user interactively chose different stencils and observe the changing behavior of an adapting algorithm. Furthermore, we want to illustrate different types of hardware architectures and thereby visualize their impact on and interaction with the algorithm.

# Bibliography


[ABF+10] Laksono Adhianto, Sinchan Banerjee, Mike Fagan, Mark Krentel, Gabriel Marin, John Mellor-Crummey, and Nathan R Tallent. HPCToolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 22(6):685–701, 2010.

[AMDa] AMD. CodeXL. http://developer.amd.com/tools-and-sdks/opencl-zone/codexl/. Accessed: 01/09/2015.

[AMDb] AMD. gDEBugger. http://developer.amd.com/tools-and-sdks/archive/amd-gdebugger/. Accessed: 01/09/2015.

[AMDc] AMD Whitepaper. OpenCL Optimization Case Study: Simple Reductions. http://developer.amd.com/resources/documentation-articles/articles-whitepapers/opencl-optimization-case-study-simple-reductions/. Accessed: 12/10/2015.

[BCD+12] Adam Betts, Nathan Chong, Alastair Donaldson, Shaz Qadeer, and Paul Thomson. GPUVerify: a verifier for GPU kernels. In *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '12)*, volume 47, pages 113–132. ACM, 2012.

[BDO04] Christian Batory, Lengauer Don, and Charles Consel Martin Odersky. *Domain-Specific Program Generation*. Springer, 2004.

[BG05] Stefan Bruckner and Meister Eduard Gr"oller. Volumeshop: An interactive system for direct volume illustration. pages 671–678, 2005.

[BOH11] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. D3: Data-driven documents. *IEEE Transactions on Visualization and Computer Graphics (Proc. InfoVis)*, 17(12):2301–2309, 2011.

[Bro75] Frederick P Brooks. *The mythical man-month*. Addison-Wesley, 1975.

[CCQ+14] Hyungsuk Choi, Woohyuk Choi, Tran Minh Quan, David G. C. Hildebrand, Hanspeter Pfister, and Won-Ki Jeong. Vivaldi: A domain-specific language for volume processing and visualization on distributed heterogeneous systems. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):2407–2416, 2014.

[CE99] Patricia Crossno and Angel Edward. Visual debugging of visualization software: a case study for particle systems. In *Proceedings of the conference on Visualization '99: celebrating ten years*, pages 417–554. IEEE, 1999.

[CHZ+07] Bas Cornelissen, Danny Holten, Andy Zaidman, Leon Moonen, Jarke J Van Wijk, and Arie Van Deursen. Understanding execution traces using massive sequence and circular bundle views. In *Proceedings of the 15th IEEE International Conference on Program Comprehension (ICPC'07)*, pages 49–58. IEEE, 2007.

[CKR+12] Charisee Chiw, Gordon Kindlmann, John Reppy, Lamont Samuels, and Nick Seltzer. Diderot: A parallel DSL for image analysis and visualization. *SIGPLAN Notices*, 47(6):111–120, 2012.

[CPP08] A.N.M Imroz Choudhury, Kristin C Potter, and Steven G Parker. Interactive visualization for memory reference traces. In *Proceedings of the 10th Joint Eurographics / IEEE - VGTC Conference on Visualization (EuroVis '08)*, pages 815–822. The Eurographics Association & John Wiley and Sons, 2008.

[CR11] A.N.M Imroz Choudhury and Paul Rosen. Abstract visualization of runtime memory behavior. In *6th IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT '11)*, pages 22–29. IEEE, 2011.

[DKK09] Gregory Diamos, Andrew Kerr, and Mukil Kesavan. Translating GPU binaries to tiered SIMD architectures with Ocelot. Technical Report GIT-CERCS-09-01, Georgia Institute of Technology, 2009.

[Fow10] Martin Fowler. *Domain-specific languages*. Pearson Education, 2010.

[Gai86] Jason Gait. A probe effect in concurrent programs. *Software: Practice and Experience*, 16(3):225–233, 1986.

[GME05] Denis Gračanin, Krešimir Matković, and Mohamed Eltoweissy. Software visualization. *Innovations in Systems and Software Engineering*, 1(2):221–230, 2005.

[Gro] Khronos OpenCL Working Group. The OpenCL specification version 1.2. https://www.khronos.org/registry/cl/specs/opencl-1.2.pdf. Accessed: 01/09/2015.

[Har07] Mark Harris. Optimizing parallel reduction in CUDA. http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf, 2007. Accessed: 15/11/2015.

[HS88] Sandra G Hart and Lowell E Staveland. Development of NASA-TLX (task load index): Results of empirical and theoretical research. *Advances in psychology*, 52:139–183, 1988.

[HWCP10] Milos Hašan, John Wolfgang, George Chen, and Hanspeter Pfister. Shadie: A domain-specific language for volume visualization. Draft paper; available at http://miloshasan.net/Shadie/shadie.pdf, 2010.

[HZG09] Qiming Hou, Kun Zhou, and Baining Guo. Debugging GPU stream programs through automatic dataflow recording and visualization. *ACM Transactions on Graphics*, 28(5):153:1–153:11, 2009.

[IBJ+14] Katherine E Isaacs, Peer-Timo Bremer, Ilir Jusufi, Todd Gamblin, Abhinav Bhatele, Markus Schulz, and Bernd Hamann. Combing the communication hairball: Visualizing parallel execution traces using logical time. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):2349–2358, 2014.

[IGJ+14] Katherine E Isaacs, Alfredo Giménez, Ilir Jusufi, Todd Gamblin, Abhinav Bhatele, Martin Schulz, Bernd Hamann, and Peer-Timo Bremer. State of the art of performance visualization. *The Eurographics Conference on Visualization (EuroVis '14)*, 2014.

[JBK+07] Matthias Jurenz, Ronny Brendel, Andreas Knüpfer, Matthias Müller, and Wolfgang E Nagel. Memory allocation tracing with VampirTrace. In *Proceedings of the 7th International Conference on Computational Science (ICCS '07)*, volume 4488, pages 839–846. Springer, 2007.

[JLP+13] Hwancheol Jeong, Weonjong Lee, Jeonghwan Pak, Kwang-jong Choi, Sang-Hyun Park, Jun-sik Yoo, Joo Hwan Kim, Joungjin Lee, and Young Woo Lee. Performance of Kepler GTX Titan GPUs and Xeon Phi system. In *Proceedings of the 31st International Symposium on Lattice Field Theory (LATTICE '13)*, number 423. Springer, 2013.

[JTD+12] Herbert Jordan, Peter Thoman, Juan J Durillo, Sara Pellegrini, Philipp Gschwandtner, Thomas Fahringer, and Hans Moritsch. A multi-objective auto-tuning framework for parallel codes. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–12. IEEE, 2012.

[Kar05] Björn Karlsson. *Beyond the C++ standard library: an introduction to boost*. Pearson Education, 2005.

[Khr] Khronos. Khronos conformance list. https://www.khronos.org/conformance/adopters/conformant-products#opencl. Accessed: 01/09/2015.

[KL83] Joseph B Kruskal and James M Landwehr. Icicle plots: Better displays for hierarchical clustering. *The American Statistician*, 37(2):162–168, 1983.

[KWm12] David B Kirk and W Hwu Wen-mei. *Programming massively parallel processors: a hands-on approach*. Newnes, 2012.

[LBG+16] Matthias Labschütz, Stefan Bruckner, M. Eduard Gröller, Markus Hadwiger, and Peter Rautek. JiTTree: A just-in-time compiled sparse GPU volume data structure. *IEEE Transactions on Visualization and Computer Graphics (Proc. SciVis '15)*, 22(1), 2016.

[Lut] Kyle Lutz. Boost compute. a C++ GPU computing library for OpenCL. https://github.com/boostorg/compute. Accessed: 01/09/2015.

[MBRG13] Peter Mindek, Stefan Bruckner, Peter Rautek, and M. Eduard Gröller. Visual parameter exploration in GPU shader space. *Journal of WSCG*, 21(3):225–234, 2013.

[MGMG11] Aaftab Munshi, Benedict Gaster, Timothy G Mattson, and Dan Ginsburg. *OpenCL programming guide*. Pearson Education, 2011.

[MH98] Timothy G Mattson and Greg Henry. An overview of the Intel TFLOPS supercomputer. *Intel Technology Journal*, 2(1):1–12, 1998.

[Mik] Mike Bostock. Towards reusable charts. http://bost.ocks.org/mike/chart/. Accessed: 05/11/2015.

[NBGS08] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with CUDA. *Queue*, 6(2):40–53, 2008.

[ND10] John Nickolls and William J Dally. The GPU computing era. *IEEE micro*, 30(2):56–69, 2010.

[NHKM14] Cristobal A Navarro, Nancy Hitschfeld-Kahler, and Luis Mateu. A survey on parallel computing and its applications in data-parallel problems using GPU architectures. *Communications in Computational Physics*, 15(02):285–329, 2014.

[NVD] NVD3. Re-usable charts for d3.js. http://nvd3.org/. Accessed: 01/09/2015.

[NVIa] NVIDIA. CUDA 7 release candidate feature overview. http://devblogs.nvidia.com/parallelforall/cuda-7-release-candidate-feature-overview/. Accessed: 20/10/2015.

[NVIb] NVIDIA. CUPTI. http://docs.nvidia.com/cuda/cupti/. Accessed: 03/11/2015.

[NVIc] NVIDIA. Nsight Visual Studio Edition 4.0 user guide. http://docs.nvidia.com/nsight-visual-studio-edition/4.0/Nsight_Visual_Studio_Edition_User_Guide.htm. Accessed: 15/11/2015.

[NVId] NVIDIA. OpenCL best pratices guide. http://www.nvidia.com/content/cudazone/cudabrowser/downloads/papers/nvidia_opencl_bestpracticesguide.pdf. Accessed: 07/11/2015.

[NVIe] NVIDIA. OpenCL SDK. https://developer.nvidia.com/opencl. Accessed: 15/11/2015.

[NVIf] NVIDIA. Visual Profiler. https://developer.nvidia.com/nvidia-visual-profiler. Accessed: 12/10/2015.

[OD12] Semih Okur and Danny Dig. How do developers use parallel libraries? In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (SIGSOFT '12)*, page 54. ACM, 2012.

[OHL+08] John D Owens, Mike Houston, David Luebke, Simon Green, John E Stone, and James C Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.

[QT] QT. WebKit. http://doc.qt.io/qt-5/qtwebkit-index.html. Accessed: 07/11/2015.

[RBGH14] Peter Rautek, Stefan Bruckner, M. Eduard Gröller, and Markus Hadwiger. ViSlang: A system for interpreted domain-specific languages for scientific visualization. *IEEE Transactions on Visualization and Computer Graphics (Proc. SCIVIS '14)*, 20(12):2388–2396, 2014.

[RKAP+12] Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Transactions on Graphics*, 31(4):32:1–32:12, 2012.

[RKBA+13] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM SIGPLAN Notices (Proc. PLDI '13)*, 48(6):519–530, 2013.

[Ros13] Paul Rosen. A visual approach to investigating shared and global memory behavior of CUDA kernels. *Computer Graphics Forum (Proc. EuroVis '13)*, 32(3pt2):161–170, 2013.

[SCO11] Jeff A Stuart, Michael Cox, and John D Owens. GPU-to-CPU callbacks. In *Proceedings of the workshops of the 16th international conference on parallel processing (Euro-Par '10)*, pages 365–372. Springer, 2011.

[SDS+] Erich Strohmaier, Jack Dongarra, Horst Simon, Martin Meuer, and Hans Meuer. The Top 500. http://www.top500.org/timeline/. Accessed: 09/11/2015.

[SGS10] John E Stone, David Gohara, and Guochun Shi. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in Science and Engineering*, 12(1-3):66–73, 2010.

[SKE07] Magnus Strengert, Thomas Klein, and Thomas Ertl. A hardware-aware debugger for the OpenGL shading language. In *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware (gh '08)*, pages 81–88. ACM, 2007.

[Sut05] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's journal*, 30(3):202–210, 2005.

[TU94] Gerald Tomas and Christoph W Ueberhuber. *Visualization of scientific parallel programs*, volume 771. Springer, 1994.