UTRECHT UNIVERSITY
Department of Information and Computing Sciences

Automatic program analysis for data parallel kernels

A thesis submitted in partial fulfilment of the requirements
for the degree of Master of Science

Author:
Călin JURAVLE

	Supervisor:
Utrecht University	Prof. Dr. S. Doaitse SWIERSTRA
	Daily supervisors:
Utrecht University	Dr. Jurriaan HAGE
Vector Fabrics	Dr. Alexey RODRIGUEZ

July 2011

Abstract

It is widely known that GPUs have more computational power and expose a far greater level of parallelism than conventional CPUs. Despite their high potential, GPUs are not yet a popular choice in practice, mainly because of their high programming complexity. The complexity derives from two factors. First, the existing programming models are tied to the underlying GPU architectures. Because of this, GPU programming methodology is different from conventional CPU programming, and forces developers to think in different terms. Second, a lot of architectural details that heavily influence program performance are not exposed to the programmer. The consequence is that in order to get fast programs developers must have a deep understanding of the targeted GPU platform.

Moreover, mapping existing sequential programs to GPU is even more difficult. The reason is that not every part of a given program is suitable to be mapped to the GPU. Identifying those suitable parts implies looking for dependencies and recognizing data parallel patterns. This is not an easy task if the program code base is very large.

The aim of this thesis is to ease the problem of mapping existing sequential programs to a GPU. To this end, we explore automatic program analyses that enable automatic and guided transformation of sequential programs to data parallel GPU kernels. Our main contributions consist of identification and implementation of key program analyses that enable such transformations. The result is a system that can identify kernel regions in a sequential program, detect GPU specific optimisations and provide additional kernel information that can be used to estimate performance. The work serves as a foundation for an automatic GPU parallelization system.

Contents

1 Introduction

1.1 Context and motivation

Modern processor architectures have embraced parallelism as an important step to satisfy the ever increasing need for performance. For some architectures this meant an increase in the number of processing units. For others, already parallel, it meant an increase in programmability. Typical examples include Central Processing Units (CPUs) for the former and Graphical Processing Units (GPUs) for the latter. CPUs, faced with the technical challenges of higher clock speeds in a fixed power envelope, have improved performance by adding multiple cores. GPUs have evolved from fixed function rendering devices into programmable parallel processors, promising big performance improvements for some classes of problems.

While multi-core CPUs have been around for quite some time already, GPUs became fully programmable just recently. This, combined with the advantages of GPUs over CPUs (more floating-point computational power, greater level of parallelism), has opened new research topics and trends with respect to program parallelization.

One such topic concerns the development of new general programming models. The goal is to help software developers take full advantage of the GPU platform. Programming models like CUDA [38], OpenCL [21], or APP [2] emerge particularly for this purpose.

Another topic, driven by the high complexity of parallel programming in general and of GPU programming in particular, focuses on assisted and automatic parallelization of existing sequential code. The importance of this area exploded with the introduction of the many-core, programmable GPUs to the mass market. It quickly become clear that performing the parallelization manually is not very productive - it takes a long time to get it right, particularly for programs with large code-bases. It is not an easy task to search for optimisation hot spots manually and to look for dependencies through thousands of lines of code. But this is precisely what the programmer has to do in order to make sure that the parallel version is semantically equivalent. This gave birth to a new research question: *how can we achieve automatic or guided GPU parallelization of existing sequential code?* The question, mainly studied in the context of parallelizing compilers, forms our research question.

In order to preserve the semantics of the program, parallelizing compilers need to analyse programs to identify safe transformations. One such analysis is data dependency analysis, which is needed to make sure that the compiler does not reorder statements that depend on each other. Besides identifying, compilers are also responsible to optimize the code. As we will see this step is a must when it comes to GPU parallelization. This is also done through automatic analyses. We can thus argue that *automatic program analysis* is a major component of the answer to the above question.

With respect to program analysis, transformations from sequential to parallel programs can be approached from two different perspectives. The first approach is to let the compiler do the transformation directly, after it *statically analysed the program.* This has the advantage of being fully automatic and sound. The disadvantage is that due to inherent limitations, only a limited number of well structured programs will benefit. The other approach involves letting the compiler also use *dynamic program analyses.* This may lead to unsound results which in turn creates the need for interactive compilation. Although not fully automatic this approach is more precise and is able to deal with to a broader class of programs. Moreover it can also help programmers to better understand the behaviour of their programs.

It is in the second context that we perform our research. The project goal is to explore possible solutions to achieve automatic or guided GPU parallelization of sequential programs.

1.2 Problem definition and goals

The problem that we aim to solve can be formulated with the help of three questions. Given a sequential program and a computational intensive part of the program answer the following:

- Is this program part suitable to be mapped to a GPU?
- If so, how should it be transformed in order to run on the GPU?
- Are there any GPU specific optimisation opportunities?

Each of these questions comes with a number of challenges. First of all, not every program is suitable to be implemented on or mapped to a GPU. In particular, programs based on task oriented algorithms where most of the work is done by a limited number of threads, or programs with a high percentage of conditional branches are better suited for CPUs. GPUs favour programs based on data parallel algorithms where the work can be split among multiple threads dominated by long sequences of computational instructions. Deciding if a particular part of a program is *GPU friendly* is thus not an easy challenge. Sometimes this requires information that cannot be easily retrieved using static analysis. Thus, combining static analysis techniques with dynamic approaches becomes a must in this context. Even if our focus will be on static analysis, we still have to consider the integration of dynamic analysis and balance its advantages and complications.

Second, GPUs are separate devices in the hardware architecture and require special configuration to execute programs. For example, because they have their own memory, different from the system memory, data needs to be copied explicitly to and from it. Detecting these extra parameters needed to create the actual mapping is thus another challenge.

Third, even if we detect that a certain part of a program can be transformed to run on a GPU, a straightforward naive transformation will not always achieve the best results in terms of performance. This fact has been suggested by NVIDIA manuals [38, 37], different independent experiments [27, 30], and has also been confirmed by us through preliminary tests. Analysing the program in order to find suitable optimisations is yet another challenge. Moreover, although existing programming models like CUDA [38] or APP [2] simplify GPU programming and indirectly the job of a compiler, they reduce little if any difficulty in optimising GPU applications. We can even argue that to some degree the added abstractions even complicate the optimisation as they make performance prediction harder and very sensitive to the specific hardware configuration.

In this context our goal is to find answers to the above questions and to solve the mentioned challenges. In other words we aim to *help users to get the most out of their data parallel programs.*

Our main tool to achieve this goal is automatic program analysis. It is important to mention that we focus on program analysis that enables program transformations; extensions like cost models to accurately predict the performance of the generated kernels are outside the scope of this research.

We aim to integrate our work in an already existing production quality compiler and analysis tool [53]. We are targeting programs written in ANSI C99 language. We plan to perform the analyses on an intermediate representation as output by the current compiler infrastructure. We do not strive to actually accomplish automatic transformation yet, but we require that, based on the produced data, it should be possible to do so.

1.3 Organization of this thesis

This thesis is organized as follows. We start by providing the necessary background information needed to understand the rest of the thesis in Section 2. More specifically, Section 2.1 provides extensive details about the GPU architecture and programming model, and Section 2.2 gives an overview of the automatic program analysis techniques that we use to achieve our goals. The rest of the thesis can be divided in two parts which presents our solution in a top down fashion.

In the first part we describe the general approach and the top level features of our solution. Section 3 gives an overview of the approach. Section 4 presents how we identify GPU friendly loops. Section 5 describes how we detect GPU specific optimisations. And finally, Section 6 discusses how we can improve the results obtained so far.

In the second part we focus on the machinery that powers up the main results. Section 7 presents the symbolic computation engine which lies at the heart of our approach. Section 8 presents details about the static analyses that we perform.

We present a sample report of our results in Section 9. Related work is discussed in Section 10. Section 11 concludes.

2
Background

2.1 CUDA platform

GPUs have a history of ever-increasing programmability. Originally developed to enable real-time display of 3D graphics that could be programmed only by using graphics commands, GPUs gradually grew to be not only powerful rendering engines but also powerful ”computational coprocessors”. This enabled developers to write general purpose programs that could run on GPUs and gave birth to GPGPU domain (General Purpose computing on Graphics Processing Units). With a powerful parallel processor at hand, available for general purpose computations, researchers are now further investigating how sequential programs can be transformed so as to benefit from all available computational power.

Before going into depths and discuss program analyses for GPUs we must first understand the GPU architecture and its programming model. Although different kinds of GPUs from different producers share core architectural concepts, there are a lot of specific details that make GPU programming hardware dependent. Efforts have been put into unifying the computing and programming model across heterogeneous platforms consisting of CPUs, different kinds of GPUs and other processors under the OpenCL specification [21]. Unfortunately, this is immature and implementations of the specification vary both in performance and completeness.

Thus, we choose to perform our research using the most mature GPU programming model, NVIDIA CUDA (Compute Unified Device Architecture) [38]. It is important to mention that CUDA is used to denote two different things: an architecture and a programming model; the context will make clear to which one we refer. Among the hardware architectures that supports CUDA, we focus on Fermi architectures [36], the latest and most powerful. Further generalisations should be possible since there exists usually a straightforward mapping to other architectures.

We note that every reference to GPUs from now on will refer to NVIDIA GPUs based on the Fermi architecture.

2.1.1 CUDA architecture

The computational power of GPUs is made possible by a scalable, massively parallel architecture that can reach hundreds of billions of floating point operations per second. In what follows we will elaborate this architecture using CUDA as example.

The CUDA architecture is built around a scalable array of multi-threaded *Streaming Multiprocessors (SMs)*. Each SM contains a set of processor cores called *Symmetric processors (SPs)* or *CUDA cores.*

For example, NVIDIA GeForce 580 GTX has 16 SMs, each consisting of 32 SPs which amounts to a total of 512 cores. Thus, the device is capable of sustaining 512 parallel hardware threads [36]. The SMs and SPs can communicate and synchronise with each other through an explicitly managed memory hierarchy. Different kinds of memories are organised in a hybrid cache and local-store hierarchy, and are available either on-chip or off-chip.

The SPs within an SM communicate through a fast on-chip local memory, called *shared memory*, while the different SMs communicate through a dedicated part (the *global memory*) of the slower off-chip DRAM (the *device memory*).

Moreover, each SP has its own register space. This is allocated dynamically from a larger register file available at SM level (for example, the 580 GTX model has 32768 registers of 32 bits per SM). The SP also has access to a private *local memory*, which is used in case the allocated registers are not enough for a particular task. The name *local memory* may be misleading, because even if it is *local* to an SP it is allocated off-chip, in the *device memory*, and thus suffers from high latency accesses.

The SPs can also access two other kinds of memories, optimised for special tasks: *constant memory* and *texture memory*. The *constant memory* space resides in device memory and is cached on-chip in the special constant cache. It is available for all device SPs and is optimised for reading access. The texture memory space resides also in device memory and is cached on-chip in the texture cache. The texture cache is optimized for 2D spatial locality, which increases the speed of accessing addresses that are close together in 2D.

Although there are multiple types of memories, the Fermi architecture implements a single unified memory request path for loads and stores, with an L1 cache per SM and unified L2 cache that services all operations (load, store and texture). The per-SM L1 cache is configurable to support both shared memory and caching of local and global memory operations.

Figure 2.1 sketches the most important parts of CUDA architecture and presents the memory access path.

Some important notes must be made regarding the access to various types of memories. The off-chip device memory has a very high latency (about ten times higher than the on-chip shared memory). While the constant and texture memories are optimised for particular situations through special caches, the global memory is only subject to the normal cache hierarchy. Thus, the access to it has to be optimised by other means. To this end the GPU employs a particular hardware optimisation - global memory coalescing. That is, accesses from adjacent threads to adjacent locations in global memory are coalesced into a single contiguous aligned memory access. This means that contiguous access to global memory by threads is essential to exploit this architectural feature and is therefore an important optimisation.

In what concerns the on-chip memories, other hardware optimisations are used in order to provide maximum bandwidth. For example, the shared memory is organised into multiple banks such that access to addresses that fall into different banks can be served simultaneously, thus greatly improving memory bandwidth. However, if the accesses are to different addresses of the same bank then a *bank conflict* occurs and the requests will be serialized. This will result in a performance penalty proportional to the number of requests.

With respect to constant memory we note that while accessing the constant cache is fast, the cache has only a single port and hence functions optimally when multiple SPs load the same value from the cache. The texture cache has a higher latency than the constant cache, but it does not suffer when memory read accesses are irregular. Because of this, it is recommended when accessing data with good 2D spatial locality.

We conclude the discussion about the memory hierarchy with the observation that it is extremely impor-

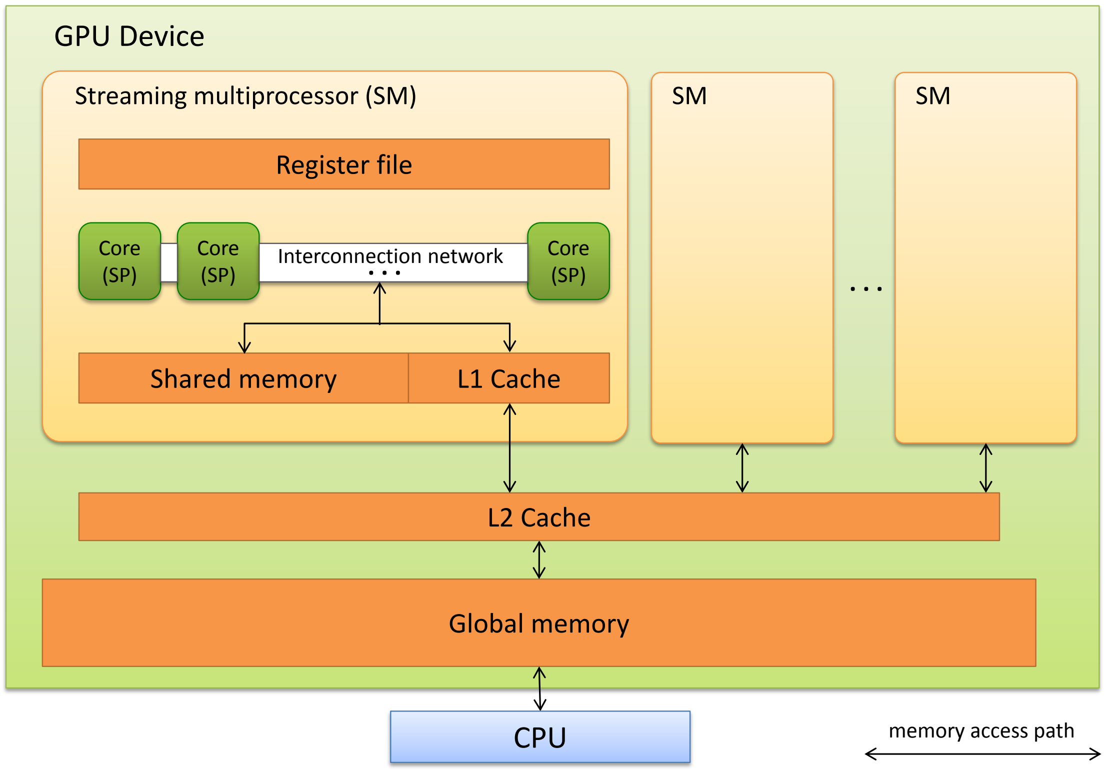


Figure 2.1: CUDA architecture

tant to reduce the number of accesses to off-chip memory and maximise utilisation of on-chip memories.

With respect to thread scheduling, the Fermi architecture has a two level distributed scheduler. At the chip level, a global *work distribution engine* schedules thread blocks to various SMs. At the SM level, there is a *warp scheduler* which distributes groups of fixed number of threads (called *warps*) to their execution units. Each SM features two warp schedulers and two instruction dispatch units. This allows two warps to be issued and executed concurrently. It is important to note that because warps execute independently, the scheduler does not need to check for dependencies from within the instruction stream, allowing for high instruction throughput.

2.1.2 CUDA programming model

The complexity of the CUDA architecture is managed by a multi-level heterogeneous programming model that focuses on algorithm design rather than the details of how to map the algorithm to hardware. The model provides an abstraction of the GPU parallel architecture using a minimal set of programming constructs such as hierarchy of threads, hierarchy of memories, and synchronization primitives.

In the CUDA software platform, as well as in the other similar models (OpenCL framework or APP platform), the computational elements of algorithms are known as *kernels* (a term adapted from its use in signal processing rather than from operating systems). A CUDA program consists of a *host program* which is run on the CPU and a set of CUDA kernels that are launched from the host program on the GPU device. The kernels can be written in the C language (ANSI- standard C99) extended with additional keywords to express parallelism directly rather than through the usual looping constructs. Once compiled, kernels consist of many *threads* that execute the same code fragment in parallel. Typically each thread instance corresponds to a single loop iteration. To make this clearer consider for example an image-processing algorithm (see Figure 2.2) that changes the pixel colours of an image according to some function. While one thread operates on a single pixel, the kernel, represented by all the threads together, operates on the whole image.

Cpu Program

```
int process(char* pixels, int N) {
  for(i = 0; i < N; i++) {
    img.pixels[i] = get_pixel_color(i);
  }
}

void main(){
  ...
  process(pixels,N)

}
```

Cuda Program

```
__global__ void process_kernel(char* pixels, int N){
  int i = threadIdx.x + blockDim.x * blockIdx.x;
  if (i < N)
    pixels[i] = get_pixel_color(i);
}

void main() {
  ...
  dim3 dimBlock(blockSize);
  dim3 dimGrid(ceil(N / (float)blockSize)
  img_kernel<<<dimGrid, dimBlock>>>(pixels,N)
}
```

🟩 kernel 🟥 gpu implicit 🟫 host code 🟦 gpu explicit - thread body

Figure 2.2: Simple CUDA example

Multiple threads are grouped into *thread blocks* which will be run on the same SM. This allows for the threads within the same thread block to synchronise through *synchronisations primitives* and communicate through *shared memory*. Although thread blocks may execute in any order, concurrently or sequentially, they can coordinate themselves by using atomic instructions on the *global memory*.

Thread blocks are also grouped together on what is called a *grid*. Each thread in a thread block is uniquely identified by its *thread id* (`threadIdx` in the example above) within its block and each thread block is uniquely identified by its *block id* (`blockIdx` in the example above). Figure 2.3 presents the different layers of a kernel organisation.

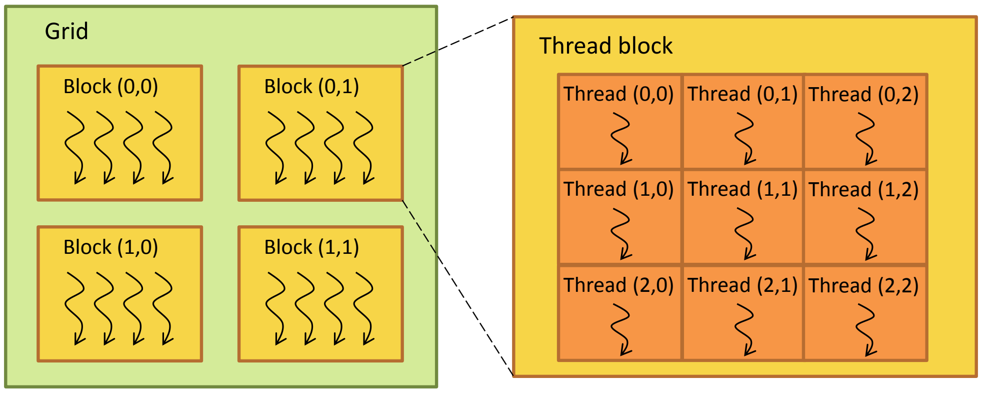


Figure 2.3: CUDA kernel threads organisation

Each thread has access to a memory hierarchy that maps closely to the before mentioned hardware view. Thus, the threads have a private *local memory* space and a private *register space*. Moreover, threads in the same thread block have access to the same *shared memory*. The device memory is accessible by all threads in a kernel through the *global memory*. Figure 2.4 presents how the memory hierarchy is exposed to the kernel.

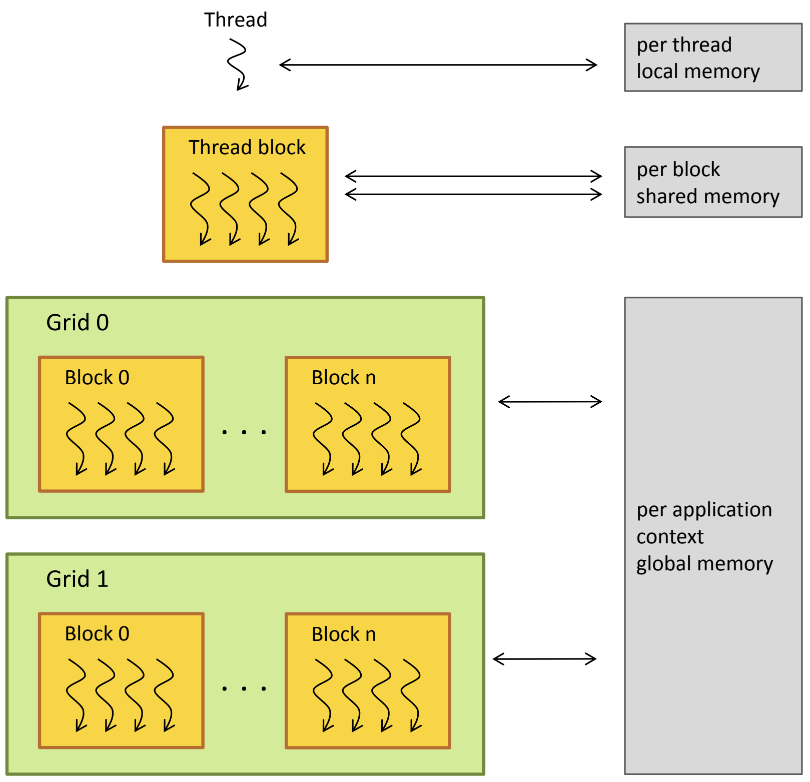


Figure 2.4: CUDA memory hierarchy

2.1.3 CUDA execution model

The CUDA execution model is guided by a SIMT (Single-Instruction, Multiple-Thread) architecture. This is akin to SIMD (Single Instruction, Multiple Data) vector organisations with the key similarity being that all cores in the same group execute the same instruction at the same time. The difference is that while SIMD vector organisation exposes the SIMD width to the software, the SIMT instructions specify the execution and branching behaviour of a single thread. SIMT can be viewed as an abstraction of SIMD where individual vector elements are abstracted to threads. This, in contrast with SIMD vector machines, enables programmers to write thread-level parallel code for independent, scalar threads, as well as data-parallel code for coordinated threads. It is important to note that for the purposes of correctness, the programmer can essentially ignore the SIMT behaviour. However, substantial performance improvements can be obtained by considering its characteristics. For example, grouping threads that follow the same execution path into the same block may lead to important performance gains.

When a CUDA program on the host CPU invokes a kernel grid, the blocks of the grid are enumerated and distributed to multiprocessors with available execution capacity. Furthermore, the multiprocessors partition the given thread blocks into groups of parallel threads (*warps*). The warps get scheduled for execution by a warp scheduler. Individual threads composing a warp start together at the same program address, but they have their own instruction address counter and register state and are therefore free to branch and execute independently.

A warp executes one common instruction at a time, so that full efficiency is realised when all threads of a warp agree on their execution path. If threads of a warp diverge via a data-dependent conditional branch, the warp serially executes each branch path taken, disabling threads that are not on that path. After all paths complete, the threads converge back to the same execution path. Branch divergence occurs only within a warp and different warps execute independently regardless of whether they are executing common or disjoint code paths.

It is also important to note the following. If an instruction executed by a warp writes to the same location in global or shared memory for more than one of the warp threads, the requests are serialised and the order is undefined.

2.1.4 The problem with CUDA platform?

Although CUDA simplifies GPU programming it suffers from several deficiencies that are not easy to overcome. These deficiencies relate to the architecture specific knowledge that developers must have in order to efficiently program the device. Basically, in order to get the most out of the GPU, the developers must know all details presented in this section (and more). This contrasts with CPU programming where knowing architectural details helps but is not essential to get good performance.

The reason for this is twofold. First, the programming model is still tied to the underlying hardware architecture. Second, details that are specific to a given generation of architectures may influence the performance of the kernels by an order of magnitude.

In general, the developers must be aware of at least the following details:

- how threads are scheduled and executed
- how warps are created and what happens if there are insufficient threads to complete a warp
- what is branch divergence and how it influences the execution
- what arithmetic operations are expensive and should be avoided
- what are the peculiarities of different types of memory
  - when coalescing applies and how it impacts the access time
  - what are bank conflicts and how can they be avoided
  - when can a particular type of memory broadcast data
- what type of memory is good for a particular access pattern
- how different mappings of the loop space to the grid space influence the kernel performance

Overall, these deficiencies make GPU programming fairly low level and difficult. In this context, our goal is to lower as much as possible the platform knowledge needed to write efficient CUDA programs.

2.2 Program analysis

Program analysis is the process of analysing computer programs with the purpose of computing approximate information about their run-time behaviour. Applications include compilers (for code improvement), software validation (for error detection), and transformations between data representations. There are two main different approaches to program analysis: *static analysis* and *dynamic analysis*.

2.2.1 Static program analysis

Static program analysis is a semantically driven analysis that relies on compile-time techniques for computing safe and reliable approximations of the set of values or behaviours that arise during program execution [35]. It is performed without executing the program and in most cases is done on the source code, although sometimes object code is preferred (binary or bytecode).

There are four major approaches to static program analysis: *data flow analysis* [35], *constraint based analysis* [35, 50], *abstract interpretation* [35, 11, 32], and *type and effect systems* [39, 24, 35]. In this thesis we use data flow analysis with abstract interpretation. In what follows we will introduce these. For the others, we direct the reader to the literature. We note that we only provide an overview; for complete details please consult the relevant literature.

Data flow analysis. This is the classical form of static program analysis. Data flow analysis views the program as a graph where the nodes are *elementary blocks* (non-jumping instructions or a group of them) and the edges describe how the control is passed between elementary blocks. The elementary blocks are usually referred to as program points. The values for the observed properties are expressed either as *equations* (leading to an equational system) or as *constraints* (leading to an inequational system). Basically, the equations can be viewed as functions which specify how information is propagated from one program point to another. For such a function the input is called the *context* and the output is called the *effect*, or equivalent: *entry/exit* set of values [35]. The preferred way to specify data flow analyses is by using a *monotone framework* which assembles generic algorithms for solving the data flow (in)equations.

The monotone framework assumes two important facts about the analysis. The first one is that the structure that holds program information forms a *complete lattice* [14] for witch the ascending chain condition hods. The second one, is that the functions that propagate information between the program points (called *transfer functions*) are monotone with respect to the chosen lattice. The lattice requirement provides a way to combine information from various program points in a consistent way. This is done using the lattice join operator. The monotonicity of transfer functions is needed in order to ease the correctness proof and the specification of the algorithm.

The classic algorithm used to solve data flow problems, formulated as monotone frameworks, is the MFP (maximal fixed point) algorithm. The algorithm computes the least fix point for a given instance of the monotone framework. It does this by iterating over the program flow, and updating the data stored at program points until the information stabilizes, i.e., it reaches a fix point. For a given program point the information is computed by applying the transfer function to its context. The effects are then propagated to subsequent program points and merged with local information using the lattice join operator. The transfer functions are defined independently, for each program point. Section 8 will provide further details about the peculiarities of the algorithm.

For a formal specification of monotone frameworks we direct the reader to [35].

Data flow analyses can be classified based on two main criteria: flow-sensitiveness and context-sensitiveness. Flow-sensitive analyses take into account the statement order while flow-insensitive ones do not. The former is more precise but the latter is more efficient. Context-insensitive analyses analyse functions independent of their calling context while context-sensitive ones take the context into account. The former is more efficient but also less precise.

Abstract interpretation. This is a general methodology for calculating analyses. Abstract interpretation views analyses as simplifications of running computer programs. Because of this it can be applied (to some extent) independently of the specification style used for presenting the program analyses.

In the context of data flow analyses we use abstract interpretation to lift some restrictions and alleviate some issues about the complexity of the analyses. For example asking for complete lattices with ascending chain condition is sometimes too much in practice. Consider the case of computing the possible values that a variables can take during the program execution. No matter what lattice we choose, none will have ascending chain condition since the set of values is possible infinite. Abstract interpretation offers techniques that make it possible to solve such cases within the monotone framework approach. One such technique introduces a *widening operator* which replaces the lattice join operator. Using the widening we make larger steps in the lattice when combining information. This allows us to achieve termination and reach a fix point solution. The downside is that by making bigger jumps we loose precision. This is corrected by the narrowing technique which tries to regain some of the lost information. We can also use the same approach to reduce the complexity of the analysis when the lattice fulfils the ascending chain condition but the chains are possible very long.

For complete information we invite the reader to consult [35] and [11], where Nielson et al. and Cousot et al. present the formal theory behind abstract interpretation and prove its correctness.

2.2.2 Dynamic program analysis

Dynamic program analysis is the analysis of computer software that is performed by executing programs built from the source code on a real or virtual machine. In contrast with static analyses, dynamic analyses are not sound. This means that the information computed by a dynamic analysis might not hold for all possible program executions. In order to diminish this risk and make dynamic program analysis be effective, the target program must be executed with sufficient test inputs to cover all interesting behaviour. Software testing techniques such as code coverage help to ensure that an adequate slice of the program set of behaviours has been observed.

Because they take into account run-time information, dynamic analyses are usually more precise than the corresponding static analyses. This makes them useful in contexts where the program behaviour heavily depends on specific properties of the input. Well known tools that use dynamic analysis include for example Daikon [17], a system for dynamic invariant detection, and Valgrind [34], a framework for detecting memory management and threading bugs.

In our case, dynamic analysis is important because it provides a way to compute information about the program that is not easily detected through static methods. An example of such information are the memory dependencies of the program, which are particularly hard to compute statically if the code uses indirect addressing, pointers, recursion, and indirect function calls. Using dynamic analysis this information can be computed as precise as the test cases allow. Moreover, dynamic analysis is the only way to gather information about particular program executions.

We also note that the system in which we integrate our work [53] makes extensive use of dynamic analyses.

3
General approach

As stated in the introduction our goal is to *help users get the most out of their data parallel kernels*. The main questions we aim to answer are: is this program part suitable to be mapped to a GPU? If so, how should item be transformed in order to run on the GPU? Are there any GPU specific optimisation opportunities?

Our approach to achieve this goal is by researching automatic program analysis techniques that:

- provide the user with enough information to get an efficient data parallel GPU program
- enable automatic program transformations of sequential programs to data parallel GPU programs

To this end, we designed and developed an automatic program analysis system with the above mentioned capabilities. The system was developed in the context of vfEmbedded, a production quality compiler and analysis tool [53]. Our system extends the analysis capabilities of the tool with an advanced static analysis framework, designed to achieve guided or automatic GPU transformations. It is important to mention that although we target ANSI C programs the techniques are mostly language independent.

In the rest of this thesis we explain in detail the design of the system and how it works. Because we have a high integration with the existing infrastructure we will be clear in what is reused and what is our contribution. We will proceed with the description in a top-down fashion. Because of this, we will often redirect the user to later sections to discover the detailed machinery behind some particular features.

The rest of this section, introduces the main concepts and ideas behind our approach. It will also give an overview of the system flow and its main components. The next sections will present the top level features of the system (Sections 4, 5 and 6). These features are the ones that the user or the transformation engine will actually use during the interaction with the system. And last but not least, Sections 7 and 8 will present the machinery (i.e., the program analysis framework) that makes these features possible.

Solution concept. The main idea behind our approach can be summarise through the following actions. Run automatic analyses to collect information about the program. Use the results to identify feasible program parts for parallelization. Once a suitable part has been found, detect possible optimisations. Aggregate the information gotten so far into a kernel model. It should be possible for an user or for an automatic transformation engine to use the model to map the detected program part to a GPU. Combine the kernel model with actual data values to get an accurate model of the executed program. It should be possible to use this model to predict the mapping performance in the context of the executed tests. If the final results are too imprecise due to a particular analysis, re-evaluate the analysis using actual kernel values found during the execution of the program. Figure 3.1 presents the detailed flow and provides more information about each step.

With respect to the above described flow we can isolate four main phases in the approach:

analysis During this phase we collect various kinds of information about the input program. The goal is to create detailed static and dynamic models of the program which enable automatic or guided transformations. In order to achieve this, we employ advanced static and dynamic analysis techniques.

identification This phase has two goals. The first one is to decide on the feasibility of a given part of the input program to be ported to a GPU. The second one is to identify the mandatory information needed for an initial transformation (e.g., inbound data).

optimisation The focus of this phase is to detect possible optimisations for the initial transformation.

instantiation This phase combines static and dynamic information to get an accurate model of the program with respect to a given test suite. Recall that in order for dynamic analysis to perform well one needs to run it for a representative set of test cases. It further detects imprecisions in the created model and fixes them using dynamic information.

We further explain and motivate specific choices that we made during the design of our solution.

What program parts are we looking to parallelize? A typical sequential program consists of elements that can run independently of each other, and of elements that must be run in order. The elements that can run independently can further be grouped according to the type of parallelism that they expose: bit-level parallelism, instruction level parallelism, data parallelism or task parallelism. Since we target the GPU as the execution platform for the parallel sections of the program we are most interested in the elements that expose data parallelism. It is well known that out of all program parts the *loops* are the hotspots to look for data parallelization opportunities. This is because under certain conditions each loop iteration can be assigned to a different thread. This perfectly fits with the GPU execution model and thus makes loops our main focus for the identification analyses.

Combining different types of analyses. When aiming for automatic program transformation, successful static analysis is to be preferred over dynamic analysis. This is because, even if not the most accurate, the results of a well behaved static analysis are always sound. Thus the transformation based on them is also sound. However, there are many cases when the static analysis cannot infer the program behaviour we are interested in, or the results it produces are too imprecise to be of any use. This usually happens when the program exhibits features that the static analysis cannot handle through approximations. In such situations a dynamic analysis of the program may produce the precise results that we are interested in. For example, it is well known that static dependency analysis is hard for code using indirect addressing, pointers, recursion, and indirect function calls [60, 55]. By contrast, dynamic analysis can handle this very well. Overall this implies that we often will have to trade soundness for precision. In such a case we can only discuss about guided program transformations.

Moreover, dynamic analysis is also essential when it comes to providing information about a particular program execution. Such information is mandatory in order to estimate the mapping performance.

Suggesting optimisations. In most cases additional GPU specific optimisations must be made in order to get a boost in performance [37], whether the GPU program was written manually or was the result of an automatic transformation. Because of the multiple tradeoffs involved in the process, choosing what optimisations should be performed is always a difficult task, especially if it has to be done automatically. This becomes even more complicated if we consider the fact that GPU programs are sensitive to small variations of the parameters (for example Ryoo et al. have shown in [48] that the effects of optimisations are usually non-linear).

Our goal is to automate the optimisation process with respect to a small set of optimisations that is likely to work for the majority of kernels. In this sense we focus on optimisations related to memory usage. More specifically we examine optimisations related to *global memory coalescing* and *shared memory usage*.

Symbolic approach. Mapping a loop to GPU usually implies creating new functions, copying data to and from devices and applying eventual optimisations. For a general program, these steps depend on the loop inbound variables and grid configuration parameters. This means that if we want to be generic and handle all classes of programs, not just the ones for which the values can be detected statically, we need to manipulate information in terms of the symbolic values of these parameters. Our approach is to construct an algebraic model of the program through static analysis. That is, we express the program behaviour such as array access bounds or memory access expressions, as algebraic expressions parametrised by the kernel parameters.

Flow overview. The overview of our approach and also the main flow of the designed system is depicted in Figure 3.1:

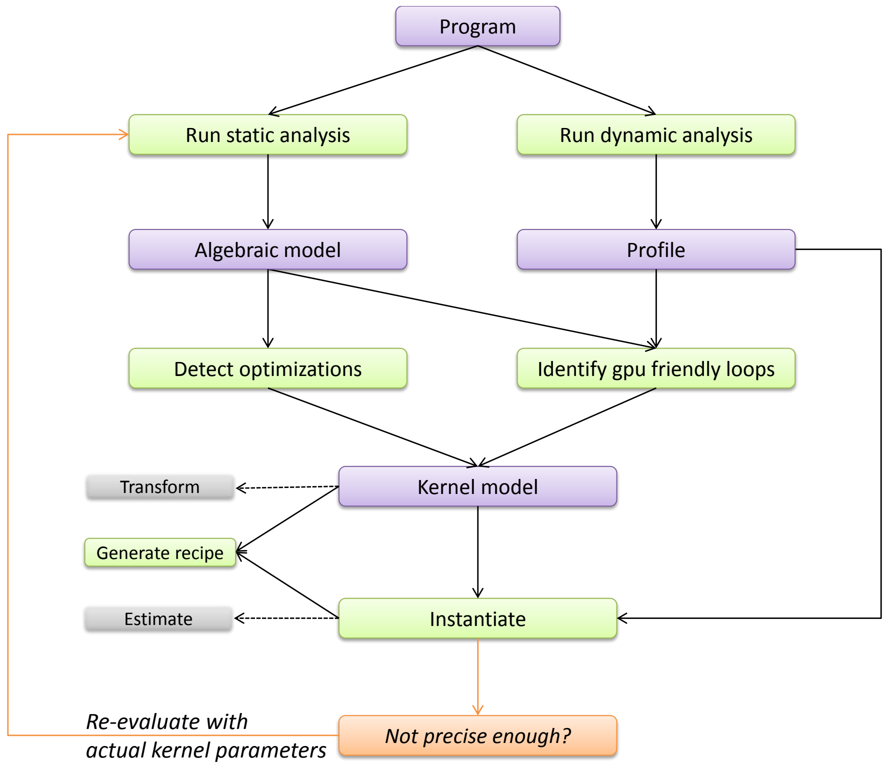


Figure 3.1: Approach overview

The starting point of the work flow is the program that we want to analyse. This is compiled into an intermediate representation that forms the basis for further program analysis. For more details about the representation we refer to Section 8.2. Next, we run static and dynamic analyses and collect various information about the program. Dynamic analysis is run on a representative set of test cases and creates a profile of the program. Based on the profile we extract program dependencies and loop counts. Using static analysis we create an algebraic model for the loop we want to parallelize. The algebraic model includes array access bounds, memory access expressions and loop invariants.

The information produced by both analyses are combined in order to identify GPU friendly loops. The identification process also takes care of discovering the inbound and outbound dependencies which will need to be copied to and from the device. After we have identified that a certain loop is feasible to be ported to a GPU we move forward and try to optimize the transformations. The process of detecting optimisations is solely based on the results of static analyses. Our main focus is on memory optimisations. Based on the result of the identification phase and optimisation phase we produce a symbolic kernel model. We call the model *symbolic* because it is based on the algebraic model of the loop and contains information in terms of inbound variables and grid parameters.

The kernel model is further combined with actual data values as they occur during program execution (e.g., the size of an image) and instantiated to create a detailed kernel report. This is referred to as the instantiation phase. The result of the instantiation can further be used to estimate the mapping performance. The symbolic report can be used to produce sound transformations of the program. Our approach right now is to generate recipes which will instruct the user how to transform and optimize the program to run on a GPU.

The reader should bear in mind that, even if we partially use dynamic analysis, our main source of information about the program is still static. There might be cases in which the results of static analyses are too imprecise to be of any use in practice. We compensate this by using dynamic analysis. The last step of the diagram refers to this case. Basically it involves re-evaluating part of the algebraic model in the context of the actual kernel parameters. This allows the system to avoid limitations of the static approach and produce precise results in the context of the execution that was analysed.

We note that for the dynamic analysis part we reused the existing infrastructure of vfEmbedded. The rest of the phases are personal contributions. The grey boxes from the flow diagram which are connected by dashed arrows are part of the future work.

4 Identifying GPU friendly loops

The identification phase consists of two steps. In the first step we investigate the GPU friendliness of the loop. This is done by checking whether certain conditions are met. The second step is applied only if the first one succeeded. Its objective is to gather essential information about the loop that will enable a first transformation towards a GPU kernel.

4.1 GPU friendliness test

In what follows we detail upon the conditions that a loop (1D or 2D) must satisfy in order to be labelled *GPU friendly*.

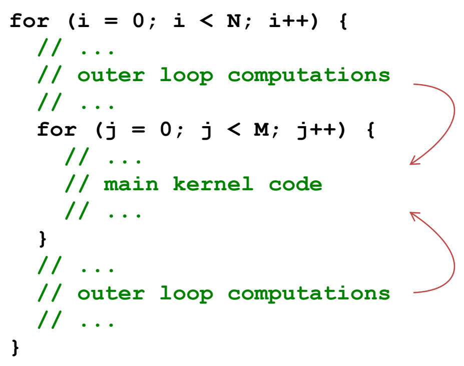
No loop carried dependencies. The loop must not have carried dependencies. That is, one iteration should not depend on data computed in previous iterations. This is needed in order to guarantee that the iterations can be executed independently of each other and that their execution order does not matter. This test is performed by using results from existing dynamic analysis. We note that we could also test for dependencies statically, by using the results of the range analysis (see Section 8.3) and employing a test like the Blume's range test [56]. Unfortunately the lack of a proper pointer alias analysis will severely limit the class of program that we can handle with such a method. Because of this we choose to use the available dynamic information.

Affine or normalizable loops. The loop must be affine. This means that the induction expression must be affine, the loop bound must be constant with respect to the loop body, and the iteration vector must not change inside the loop body. In other words, the loop must be normalizable: we should be able to get an equivalent version of the loop whose induction variable starts at 0 and has a unitary stride.

The normalization conditions are needed because the loop space has to be mapped to a grid of threads as described in Section 2.1.2. Without them, such a mapping is no longer possible in the general case.

For nested loops (2D patterns) additional restrictions are imposed on the inner loop:

- be an immediate child of the outer loop
- be the only inner loop at its level
- not have loop carried dependencies



```
for (i = 0; i < N; i++) {
  // ...
  // outer loop computations
  // ...
  for (j = 0; j < M; j++) {
    // ...
    // main kernel code
    // ...
  }
  // ...
  // outer loop computations
  // ...
}
```

Figure 4.1: 2D loops complications

In the context of 2D loops special attention must be paid to the outer loop computations. Figure 4.1 showcases this scenario. As suggested in the picture, the outer loop computations should be pushed inside the inner loop. Because there are no loop carried dependencies this can always be done in a safe way. However, pushing computations inside the inner loop will affect how often they are executed (in the example, `M` times instead of just once). Thus one might be concerned about the efficiency of such a transformation. In most cases this is not a problem because the loops will be mapped to hundreds of cores. Here, one should make the connection with scatter-gather algorithms which form a proven technique for parallelization [23]. From an implementation perspective the feasibility is tested with the help of an estimation module but a discussion on this subject is outside the scope of this thesis.

No writes to static or global variables. Writes to global variables are prohibited since they usually create loop carried dependencies. Moreover, static variables are not supported in current GPU kernels. Thus we do not allow for functions that are called from within the loop body to contain declarations of static variables. This test is performed statically by looking at variables types.

Limited number of library and system calls. We have to bear in mind that GPUs are not a full substitute for CPUs, thus not all library and system calls are available when executing the code. Nevertheless, some intrinsics are supported on GPUs. Such examples include mathematical functions (eg. *sin*, *cos*) and memory allocation functions (eg. *malloc*). To accommodate this we use a white-list which specifies which intrinsics can be used on the GPU.

4.2 Detecting kernel parameters

Besides applying the feasibility test, the identification phase concerns itself with recognizing kernel input and output parameters (i.e., loop inbounds and outbounds). It is important to realize that just reporting what program variable should be used as input/output is not enough. For array variables, the accessed sections are also a mandatory requirement. The reason for this is that data must be manually copied between the host (CPU) memory and the device (GPU) memory. In order to do so, the exact memory bounds must be known, either as exact numeric values or as symbolic values (in terms of other variables). Note that this issue does not occur in CPU multi-core parallelization. There, different threads have access to the same memory space.

The accessed size is computed by means of static symbolic range analysis (Section 8.3). In order to minimise the overhead of the data transfer between the host and the device we compute only how much is used of a given array and not how much was allocated. For example, if one allocates N bytes for an array, but during the loop only the first half is used we will report that only $N/2$ bytes need to be copied.

For a given array input variable, the actual report is made in terms of an accessed section - a closed interval of which the lower bound is the starting offset and the higher bound is the last accessed offset. The interval bounds are reported in bytes rather than in the actual size of the array data type. Although this might seem a bit strange, the motivation behind it is that it accommodates pointer casting. Recall that ANSI C includes cast operators which influence how pointer arithmetic is done. The following example clarifies the advantage of having the offsets in bytes:

```
int in[10];
int out[15];
for (int i = 0; i < 15; i++) {
  char *aux = (char*)in;
  out[i] = aux[i];
}
```

Even if the inbound array `in` is of type `int*` (4 bytes per element) it is accessed like a `char` array (1 byte per element). This means that only the first 15 bytes will be accessed instead of all 40. The corresponding access section will be $[0, 14]$. In contrast to this, the outbound array `out` will access 60 bytes and its access section will be $[0, 56]$. Having the offsets in bytes allow us to report on such cases in a very precise way.

A critical detail about array sections is that the expressions of the interval bounds are *algebraic symbolic expressions*. In particular, they also represent complex expressions that depend on other input parameters. The example below showcases this phenomenon. The left side presents a simple program to be analysed (matrix addition), and the right side presents the generated report with the input and output parameters.

```
typedef struct {
  int *data;
  int size
} matrix_t;

int add(int* a, int* b,
        matrix_t* c, int N) {
  for (int i = 0; i < N; i++)
    for (int j = 0; j < N; j++)
      c->data[i * N + j] =
        a[i * N + j] +
        b[i * N + j];
}
```

Inbounds:

- `a` $\Rightarrow [0, 4 * N * N - 4]$
- `b` $\Rightarrow [0, 4 * N * N - 4]$
- `c`
- `N`

Nested outbounds:

- `c->data` $\Rightarrow [0, 4 * N * N - 4]$

The *nested outbounds* from the above report represent a special case of output parameters. We will explain them with the help of the matrix addition example. It is easy to see that the only proper output parameter is actually `c`. However, in practice it is of little use to only know this; especially if we want to infer how much memory we should allocate on the device and how much we should copy from it. This information is mostly given by the `data` field of the structure. Unfortunately the field is "hidden" by its parent structure `c` and is not directly visible as an output parameter. When this happens we say that we encountered a *nested pattern*. The concept applies to both input and output parameters. Also, it is not specific to the use of structures. It applies to the general case where other forms of nesting are used (e.g., nested pointers).

Detection mechanism

All kernel parameters - inbound and outbound - as well as the nested patterns are detected statically. This gives us the advantage of being sound but also creates some precision issues in what concerns the set of outbounds. In what follows we present details about the detection procedure and discuss its limitations.

Input parameters. Input parameters are detected by looking at the scope of variables using *definition-use* chains. That is, we classify as inbound every variable that is used inside the loop body but defined outside it. This is always precise and will contain the exact set of input parameters. However, the set of inbounds sometimes might be larger than one would expect. To understand how and when this can happen recall that we perform the analyses on an intermediate representation of the program and not on the source code. The representation (see Section 8.2) is obtained after various compiler transformation. This means that from the source code to the representation we analyse, the compiler is free to reorder computation and/or introduce additional variables. When this happens our procedure can output more input variables than are actually visible in the source code. To better understand this, consider the transformation that the compiler does when normalizing loops. For clarity reasons we present the example in terms of source code.

```
for(i = -n; i <= n  i++){
  //...
}
```

Original loop - *inbounds: n*

```
aux = 2*n > 0 ? 2*n : 0
for(j = 0; j <= aux; j++){
  i = j - n;
  //...
 }
```

Normalized loop - *inbounds: n, aux*

During normalization the compiler replaces the original induction variable (`i`) with a new one (`j`) that iterates from `0` to the number of iterations of the original loop (`2*n`). The upper bound of the normalized loop is extracted in another variable (`aux`) that is further constrained to be greater or equal to zero. The original induction variable is reconstructed by adding its minimum value (`-n`) to the new induction variable. Because the extra introduced variable (`aux`) is defined outside the loop it will be considered an inbound variable.

Returning more inbound parameters is not a problem for an automatic transformation engine because they can be handled automatically. However, they might pose problems to the user, since they do not actually appear in his program. This is currently a limitation of our system.

Output parameters. In order to detect output parameters, we inspect the live variables at the loop exit. Unfortunately, this alone is not enough to determine the complete outbound set. It is possible for example that an array does not show up in the list of live variables because it is no longer referenced after the loop. Still, the memory it refers to could be accessed from other functions. This means that if a store operation is performed on the array inside the loop, the array becomes an outbound even if it is not in the live variables list. The following example illustrate this scenario:

```
void init_array(int* a) {
  for(i = 0; i < n; j++){
    a[i] = i;
  }
}
```

```
void main(){
  //
  init_array(a);
  // use a...
}
```

To accommodate this case, we consider as outbound parameters every array that has a store operation performed on it. The disadvantage of this is that it might catch a locally defined array which is not a real outbound. One might think that the set can be reduced by discarding all local arrays from it. However, this is not always possible because a local definition may be an alias for an inbound variable:

```
int **pp;
for (i = 0; i < n; i++) {
  int *p = pp[i];
  // use p
}
```

Nested patterns. Nested patterns are identified starting from a specific memory operation and reasoning about the origin of the corresponding pointer. If the pointer, say `p`, is obtained through a load operation (eg: `int* p = pp[i]` or `int *p = pp->pointer-field`) we mark it as a potential nested pattern. We then proceed with its parent, `pp` in our example, and apply the same procedure until we hit the root of the hierarchy. If the root is an inbound (outbound) pointer or a structure we mark the original pointer, `p`, as a nested inbound (outbound). Note that when we follow the definition chain we may encounter arbitrary pointer expressions. It is important to mention that these expressions are produced by the *expression reconstruction analysis* (Section 8.4).

Other information

Besides the above information we also report the functions which are called from within the loop body. The importance of this particular piece of informations is given by the fact that a GPU kernel cannot simply call a host function. First, the function must be ported to a GPU version. This is done by annotating the function declaration with special keywords.

The functions are detected statically, by traversing the call graph of the loop body. A limitation of the current infrastructure is that it cannot reason about functions pointers. We can at most report that function pointers are used, but we cannot derive the set of possible functions for a particular pointer.

5 Detecting optimisations

Most of the time, straightforward transformations of loops into GPU kernels will note achieve the "promised" speed-ups. The reason for this is that performance is greatly influenced by architectural details, and some of them are not directly exposed through the programming model. This section motivates our choice for looking into specific kinds of optimisations and presents how we achieve them in the context of the designed system.

Various recommendations [37, 47, 30] and experiments [49] suggest that memory optimisations are the most likely optimisations to improve program performance by a big factor. The goal is to maximise the use of the hardware resources by maximising the memory bandwidth. Bandwidth is best served by using as much fast memory and as little slow-access memory as possible. The former is satisfied by using for example *shared memory.* The latter can be tackled by grouping multiple memory request into one large request which will execute much faster; this is referred to as *memory coalescing.* We dedicate our efforts to identify and report these kinds of memory optimisation. But before going into depth and explaining how our system detects them, we need to clarify the architectural details behind them.

Global memory coalescing. NVIDIA states that the single most important performance consideration in programming for the CUDA architecture is coalescing global memory accesses [37]. The main reason for this is that access to global memory is very slow when compared to access to on-chip memory (by an order of magnitude). Thus reducing the number of accesses is critical, which is what coalescing is all about. Reports [47, 6, 37] and our own experiments showed that proper use of memory coalescing can lead to speedups of more than $8\times$.

Global memory loads and stores by threads of a warp are coalesced by the device into as few as a single transaction when certain access requirements are met. Figure 5.1 shows the impact of coalescing on the number of memory transactions. Based on the memory area that is accessed we can have from one up to the number of threads in a warp memory accesses. Grouping together the threads that manifest a good locality of memory accesses is thus a critical optimisation.

On-chip shared memory. Because it is on-chip, shared memory is much faster than local and global memory. This creates the opportunity to use it as an explicitly managed global memory cache. For example, when multiple threads in a block use the same data from global memory multiple times, shared memory can be used to manually cache the data and have only one access to global memory.

Aside from memory bank conflicts, there is no penalty for non-sequential or unaligned accesses by a warp in shared memory. As mentioned during the presentation of the GPU architecture, in order to achieve a high bandwidth for concurrent accesses the shared memory is split into different areas called *banks.* Each

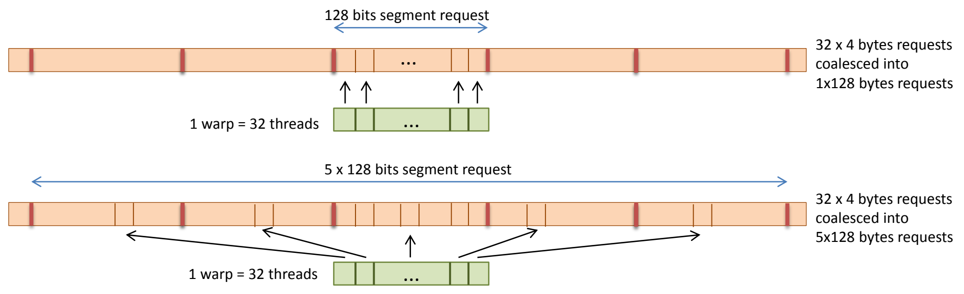

128 bits segment request
32 x 4 bytes requests coalesced into 1x128 bytes requests
1 warp = 32 threads
5 x 128 bits segment request
32 x 4 bytes requests coalesced into 5x128 bytes requests
1 warp = 32 threads

Figure 5.1: Memory coalescing impact on the number of requests.

bank can execute load and store operations independently of the other. Thus n simultaneous requests to memory that address n different banks will be served simultaneously. In contrast, if different requests fall into the same bank at different addresses they will be serialized. This degrades the performance by a factor equal to the number of request. However, if the requests fall into the same bank but at the same address then a single broadcast operation will be performed. Figure 5.2 illustrates bank conflicts by presenting two different kinds of accesses to the shared memory.

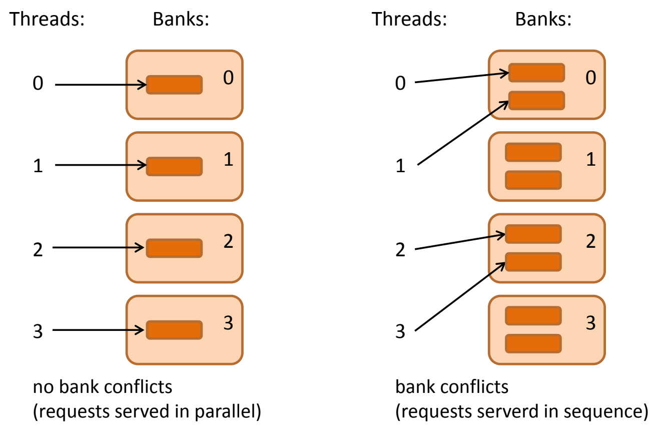


Figure 5.2: Different access patterns to the shared memory.

Shared memory can be helpful in several situations, such as coalescing and eliminating redundant access to global memory. Reports [47, 37] and personal experiments showed that bank-free use of shared memory can lead to speedups of more than 30×. However, the shared memory can also act as a constraint on occupancy (a measure of how many threads are handled by a SM at a given moment) and thus reducing the level of parallelism. This stems from the fact that shared memory is stored per SM and is shared by all blocks. In order to clarify this we will consider a concrete example. Suppose that you want to allocate 32KB of shared memory per block and that the total available amount of memory is 48KB. Since the shared memory is allocated per block, only one block of threads will be able to run on a SM. If the shared memory allocation is lowered to 16KB then 3 thread blocks will be able to run concurrently on the same SM, maybe leading to better resource usage. Usually, shared memory optimisations have to balance the need for fast memory and the need for high occupancy. Program transformations, such as statements reordering, might enable the possibility to reuse the memory; thus reducing the need for shared memory.

5.1 Coalescing analysis

Coalescing analysis statically computes how many memory requests (*coalescing factor*) will be issued per warp at runtime for a particular memory operation. The analysis is performed for each store and load operation from the loop body. The goal is to gather information about the coalescing behaviour of the kernel and report it together with optimisation advice to the user or the transformation engine. In order to compute the number of memory requests we make use of the access expressions of the memory operations. These are computed by the *expression reconstruction analysis*. Section 8.4 provides all the details about the reconstruction process; what is important to mention here is that the access expressions are symbolic expressions, parametrised by induction variables, inbounds and loop invariants.

The analysis returns for each memory operation its coalescing factor (i.e., the number of requests that are issued per warp). Each coalescing factor is an algebraic symbolic expression that will depend on: kernel input parameters, grid variables (e.g., thread index) and loop invariants. We will first explain the coalescing analysis informally, through an example, and then we will formalize the approach in an algorithm.

For now, consider again the square matrix addition algorithm. We are interested in mapping the two nested loops to a 2D GPU grid space.

```
void add(int* a, int* b, int* c) {
  for (int i = 0; i < N; j++) {
      for (int j = 0; j < N; j++) {
          int idx = i * N + j;
          c[idx] = a[idx] + b[idx];
  }
}
```

For reasons of space, we will only reason about the access to `a`. The rest proceeds in a similar way. The key idea is to compute the memory area accessed by a randomly chosen warp and then divide it by the memory request line size.

The offset expressions for the load operation on `a` is $i * N + j$. The expression depends on N, an inbound variable, and on i and j, induction variables that define the grid coordinates. Thus we can write it as a function of these parameters: $f_a(i, j, N) = i * N + j$. Since warps are computed based on threads and blocks ids it is useful to rewrite the expression in terms of actual grid coordinates: $f_a(Grid.x, Grid.y, N) = Grid.x * N + Grid.y$. Note that $Grid.x$ is not a valid CUDA variable. It is introduced by us for convenience and readability. We remind the reader that grid coordinates are computed based on block and thread coordinates: $Grid.x = ThreadIdx.x + BlockIdx.x * BlockDim.x$. It is important to note that this guarantees that adjacent threads will receive adjacent grid coordinates.

We can isolate a random warp by using the knowledge about how threads are assigned to warps. In short, warps are created by enumerating the threads over their x grid coordinate. Figure 5.3 depicts warp creation for a block of size 64×2.

Thus, we isolate a random warp w by fixing all parameters of the expression except the x coordinate. Doing this we get $f_{wa}(Grid.x) = Grid.x * N + Grid.y$. Note that N and $Grid.y$ are "downgraded" from variables to symbolic constants. It is straightforward to see that in the selected warp, w, the threads will access the following memory cells: $f_{wa}(Grid.x)$, $f_{wa}(Grid.x + 1)$, ... , $f_{wa}(Grid.x + warpSize - 1)$. Based on this observation we can compute the memory requirement for the entire warp. But instead of evaluating the expressions for each warp thread and then test for collisions, we compute the requirement based on monotonicity analysis. In our case, we analyse the monotonicity of f_{wa} with respect to $Grid.x$.

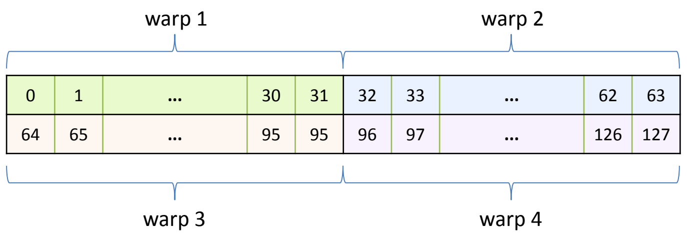


Figure 5.3: Warp distribution inside a thread block

We redirect the reader to Section 7.2 to discover how the monotonicity is determined. One can easily see that f_{wa} is increasing monotonically in $Grid.x$. Because of this the memory area accessed by the warp can be computed as: $warpArea_{wa} = f_{wa}(Grid.x + warpSize) - f_{wa}(Grid.x) = N * warpSize$. What remains to be done in order to compute the number of issued requests is to divide the warpArea by the request line size. For our example we get:

$$reqNr_a = \frac{warpArea_{wa}}{reqLineSize} = \frac{N * warpSize}{reqLineSize} = \frac{N * 32}{128} = \frac{N}{4}$$

Note that the computed number of requests ($reqNr_a$) can exceed the warp size ($warpSize$) which is actually the upper bound of requests per warp. In order to get the real number of requests we have to take the minimum of the two values. However, since this is a trivial step, we will omit it in further discussions and report only the number of requests as computed above.

The attentive reader probably spotted the following issues with the above approach:

1. the method is suitable only when the block size divides the warp size
2. monotonicity cannot be determined for every expression
3. even if monotonicity can be computed, expressions might still be non-monotonic
4. in the presence of conditionals multiple expressions are possible for the same memory operation

The first issue is not a major one. If the block size does not divide the warp size the result will be an optimistic approximation and will loose its soundness property. However, this is not a problem even if we consider automatic transformations that are based on it. The reason is that we use coalescing information to reason about performance and not about correctness.

The second issue is real and can be linked to limitations of our system. Indeed, because we work with symbolic expressions we cannot always compute their monotonicity. A simple example in this sense is $f(x) = x * a$. The monotonicity of f will depend on the sign of a which might not always be known. In particular, the result of monotonicity analysis greatly depends on the precision of the sign analysis (Section 8.5) which is responsible for inferring the signs of variables. Our approach in this scenario is to *increase precision by assumptions*. The approach is based on the fact that missing variables signs are the only reason for which we are not able to decide the monotonicity of a symbolic expression. Basically, when we cannot compute monotonicity we start making various assumptions about the missing signs (e.g., variable a is positive). We then use the assumptions to decide on the monotonicity and proceed with the next coalescing computations. We always report back the assumptions that were made during the analysis.

The third issue is handled through a brute force method. When the expression we analyse is non-monotonic (e.g., $f(x) = x^2$) we compute the memory offsets for each thread in the warp. If we detect that two threads access the same memory cell then we decrease the number of requests by one. At the end, we report the total number of different accesses.

The fourth and last issue refers to the following scenario:

```
int *p;
int idx;
if (...) {p = a; idx = x;}
else     {p = b; idx = y;}
... = p[idx];
```

At line 5 the expression for the load operation `p[idx]` can be either `*(a + x)` or `*(b + y)`. In general, at a given program point each variable can have multiple possible expressions. For a memory operation, the expressions may differ in two orthogonal directions. One is the pointer on which the operation is performed; the other is the offset. For each direction we take different actions. If the expression differs only in the offset, we apply the analysis for each expression and take the maximum result. If the expressions contains multiple pointers, we are conservative and return the maximum number of memory requests. It is important to mention that we are able to reason about the pointer values because the *expression reconstruction analysis* also serves as a basic pointer alias analysis. For more details about about what types of aliasing we are able to discover see Section 8.4.

The algorithm for computing the coalescing factor for a single offset expression is summarised below.

Algorithm 1 Coalescing algorithm

```
procedure GETCOALESCINGFACTOR($f, warpSize, reqLineSize$)
    $x \leftarrow$ the variable that corresponds to the $x$ grid coordinate
    $f_w \leftarrow f$ where all variables are fixed except $x$ grid coordinate
    $mono \leftarrow$ MONOTONICITY($f_w, x$)
    if $mono$ is unknown then
        $culprits \leftarrow$ variables in $f_w$ which do not have a sign        ▷ sign assumptions
        $f_w \leftarrow f_w$ where each variable in $culprits$ is assumed positive
        $mono \leftarrow$ MONOTONICITY($f_w, x$)
    end if
    if $mono$ is monotonic then        ▷ f is monotonic
        if $mono$ is increasing then
            $size \leftarrow f(x + warpSize) - f(x)$
        else
            $size \leftarrow f(x) - f(x + warpSize)$
        end if
        return $\frac{size}{reqLineSize}$
    else        ▷ f is not monotonic
        $offsets \leftarrow [f_w(x), f_w(x+1), \ldots, f_w(x + warpSize)]$
        $uniqOffsets \leftarrow removeDuplicates(offsets)$
        return $size(uniqOffsets)$
    end if
end procedure
```

Improving coalescing

Because the warps are computed based on the x grid coordinate the coalescing factor will be influenced by how the loop space is mapped to the grid space. To understand the relation between coalescing and mapping let us study again the matrix addition example. The coalescing factor for the straightforward mapping is $\frac{N}{4}$. In practice this is very bad since it will produce the maximum number of requests every time (usually $N > 128$). Consider now a reverse mapping of the loop space to the grid space. This implies that j becomes $Grid.x$ and i becomes $Grid.y$ (refer to Figure 5.4 for a graphical representation). Using the above algorithm we will have:

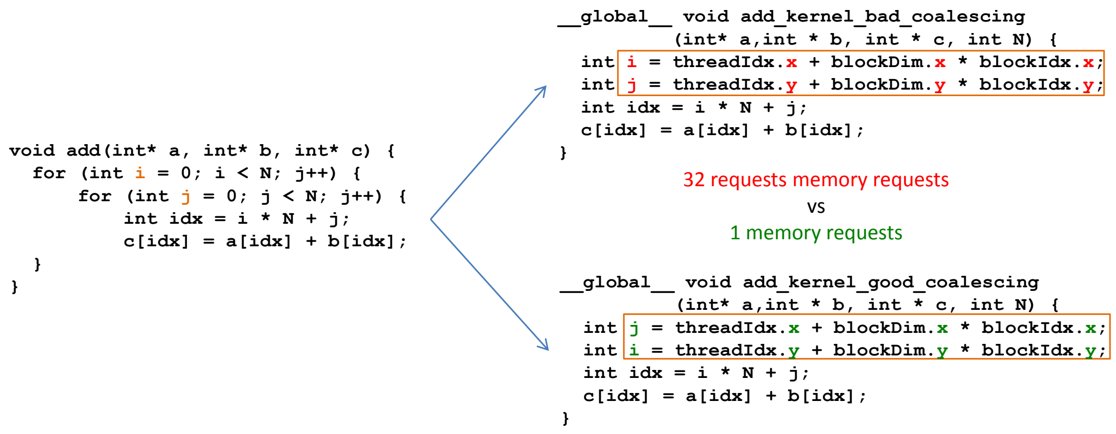
$$f'_{wa}(Grid.x) = Grid.y * N + Grid.x$$

$$warpArea_{wa} = f'_{wa}(Grid.x + warpSize) - f'_{wa}(Grid.x) = warpSize$$

$$nrReq_a = \frac{warpArea_{wa}}{reqLineSize} = \frac{warpSize}{reqLineSize} \approx 1$$

Thus, by reversing the mapping we get perfect coalescing (1 memory request) instead of perfect uncoalescing (32 memory requests). To stress the importance of good coalescing we mention that in this particular case, even if we have only 3 memory operations, the coalesced mapping obtains a 2× speedup over the uncoalesced one.

During the coalescing analysis we always perform extra *what if analyses* that tries to infer a better coalescing behaviour based on different mappings. If a better mapping is found then we report it, so that the user or the transformation engine can take actions based on it.



```
void add(int* a, int* b, int* c) {
  for (int i = 0; i < N; j++) {
      for (int j = 0; j < N; j++) {
          int idx = i * N + j;
          c[idx] = a[idx] + b[idx];
  }
}
```

```
__global__ void add_kernel_bad_coalescing
          (int* a,int * b, int * c, int N) {
  int i = threadIdx.x + blockDim.x * blockIdx.x;
  int j = threadIdx.y + blockDim.y * blockIdx.y;
  int idx = i * N + j;
  c[idx] = a[idx] + b[idx];
}
```

```
__global__ void add_kernel_good_coalescing
          (int* a,int * b, int * c, int N) {
  int j = threadIdx.x + blockDim.x * blockIdx.x;
  int i = threadIdx.y + blockDim.y * blockIdx.y;
  int idx = i * N + j;
  c[idx] = a[idx] + b[idx];
}
```

Figure 5.4: Impact of mapping on coalescing factor

Possible extensions

We note that the same techniques can be used to reason about other "hidden" architectural details that affect the runtime performance of the kernel. In particular, they can be applied to compute:

- bank conflicts for shared memory (NVIDIA and AMD specific)
- channel conflicts (AMD specific)

Due to space constraints we will not elaborate on these topics. The reader is invited to reason by himself in order to see how the techniques can be applied in the mentioned context. Also, the AMD manual [2] provides the necessary details to understand the above AMD specific concepts.

5.2 Shared memory analysis

Shared memory analysis detects memory areas that will benefit from being copied to the on-chip shared memory. There are two cases when this might happen. The first one is when the memory is accessed in an uncoalesced way. Because shared memory has very low latency, coalescing does not apply. Thus, one can first copy the accessed memory in a coalesced manner to the shared memory and then access it with the original pattern form there without penalties. The second case arises when different threads access overlapping memory areas. Because shared memory can act like a programmable cache, the overlapping areas can be copied to it and benefit from its low latency. We will further refer to these cases as *sharing opportunities*.

The analysis is initially done for each memory operation and is then refined in subsequent steps. Its input consists of the complete memory sections that can be accessed at each memory operation. Memory sections are expressed as closed intervals $[starting\_offset, ending\_offset]$ and are produced by the *symbolic range analysis* (Section 8.3). The interval bounds are defined as algebraic expressions.

The analysis associates to each memory operation the following pieces of information:

1. whether or not it exposes a sharing opportunity in the sense defined above
2. if the sharing opportunity is due to overlapping accesses then two additional things are reported:
    (a) how much memory is shared between two adjacent threads (the overlapping area)
    (b) how much memory should be allocated for a thread block and from where should that memory be copied (recall that shared memory is private per thread block)

The results are reported as symbolic intervals parametrised by inbound variables and grid variables (block index, block size).

Reporting how much memory is shared between two adjacent threads is critical in assessing the quality of the optimisation. Because shared memory is limited, one may have to choose between different sharing opportunities. In this case, among the most beneficial mappings will be the one that allows threads to share the most memory. The report gives exactly this piece of information. In other words, reporting overlapping areas helps estimating the performance of the different sharing opportunities.

As with coalescing analysis we will explain shared memory analysis with the help of a concrete example. But in order to do this we need to study a more complex example than matrix addition. For this purpose, we pick a classic image processing algorithm: 2D image convolution. We will focus only on sharing opportunities due to overlapping accesses. The coalescing case is just a restricted version of this and can be easily derived from it.

2D image convolution

The 2D convolution algorithm takes as input an image and a filter and produces a new image. Each output pixel is computed as a function of the input pixel, the filter and the neighbouring pixels. Figure 5.5 presents the idea of the algorithm.

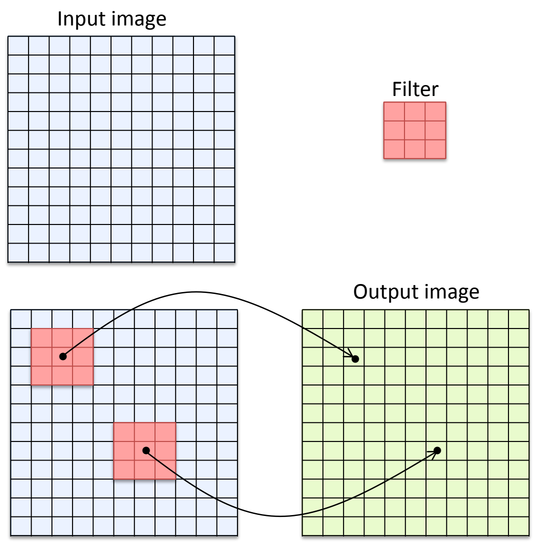


Figure 5.5: Convolution algorithm

Before continuing to the source code and the actual analysis it is important to mention what we are trying to discover. In a CUDA mapping of the algorithm the image will be divided into subimages and each subimage will be assigned to a different thread block. Basically, each pixel will be processed by an individual thread belonging to some thread block. Processing a pixel translates to iterating over the filter and the neighbouring pixels and accumulating the value for a new pixel. From a memory perspective this means that two adjacent threads will access the same filter memory area and also some part of the input image area. Our goal is to discover these memory areas. Figure 5.6 presents a graphical representation of the sharing. The green lines suggest how the image is divided into sub-images. For this example we considered a block size of 4×4.

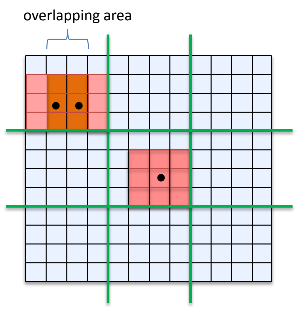


Figure 5.6: Sharing in a convolution grid

The C code for a 2D colour convolution is presented below. For brevity we skipped the bounds tests for border pixels as well as clamping the colours if they exceed the allowed values. In order to easily understand the code, it is important to mention that each pixel is represented with 3 bytes which correspond to the RGB spectrum.

```
1    for (y = 0; y < image_height; y += 1)
2    {
3      for (x = 0; x < image_width * 3; x += 3)
4      {
5        unsigned char *out_row = out_image[y * image_width * 3];
6        int fx, fy;
7        int red,grn,blu;
8        for (fy = 0; fy < filter_height; fy += 1)
9        {
10         int py = y + fy - (filter_height / 2);
11         unsigned char *in_row = in_image[py * image_width * 3];
12         for (fx = 0; fx < filter_width; fx += 1)
13         {
14           int px = x + 3*(fx - filter_width / 2);
15           int coeff = filter[fx + fy * filter_width];
16           // out of bounds tests skipped for brevity
17           red += in_row[px + 0] * coeff;
18           grn += in_row[px + 1] * coeff;
19           blu += in_row[px + 2] * coeff;
20         }
21       }
22       // clampping skipped for brevity
23       out_row[x + 0] = red * filter_gain;
24       out_row[x + 1] = grn * filter_gain;
25       out_row[x + 2] = blu * filter_gain;
26     }
27   }
```

Shared memory analysis on 2D convolution

We will analyse only the memory operations on `filter` (line 15). For the rest, similar computations are done. From the range analysis we get that the value interval for the access offset is $[0, filter\_height * filter\_width - 1]$. As in coalescing analysis, because we are interested in the behaviour with respect to the grid distribution, we abstract away the interval as a function of its symbolic variables and grid coordinates. Abusing mathematical notation we can re-write the interval to: $r_f(Grid.x, Grid.y, f_h, f_w) = [0, f_h * f_w - 1]$. We use r for the range function and the subscript f to suggest that it is associated to the filter variable. Also, for brevity we use f_h for $filter\_height$ and f_w for $filter\_width$. We further fix every variable except the grid parameters and get the interval function for a grid cell: $r_{gf}(Grid.x, Grid.y) = [0, f_h * f_w - 1]$ (subscript g suggest the restriction of r_f to grid coordinates). At this point it is trivial to see that the r_{gf} is actually a constant function. Thus, each thread will access the same memory area. However, we will continue with the reasoning to illustrate the entire procedure.

In order to compute the overlapping area between adjacent threads we instantiate r_{gf} with the neighbouring grid coordinates and intersect the resulted intervals. Because in general, the overlapping area depends on the axis, we compute independent values for each axis. Doing so, allow us to detect cases when threads share memory only across a single axis. We consider as adjacent threads for a given thread the natural neighbours on each axis. That is, for a thread identified by the coordinates (x, y) the adjacent threads will be $(x + 1, y)$ and $(x, y + 1)$. Thus, for x axis we get the following sharing:

$$share_{xf} = r_{gf}(Grid.x, Grid.y) \cap r_{gf}(Grid.x + 1, Grid.y) = [0, f_h * f_w - 1]$$

For the y axis a similar expression is obtained. Because the intersection is not empty it means that we have a sharing opportunity. The next step is to compute the memory requirement for an entire block. For this, we apply the same monotonicity technique that we used in the coalescing analysis. But first, we need to choose a random block. We do this by looking at the coordinates of the block first and last threads. In general, a random block can be identified by the coordinates of its first thread: $(BlockIdx.x * BlockDim.x, BlockIdx.y * BlockDim.y)$. For brevity we will use the notation $(Block.x, Block.y)$ to identify a random block. The block spans until the coordinate $(Block.x + BlockDim.x - 1, Block.y + BlockDim.y - 1)$. We can compute the memory bounds by evaluating r_{gf} for the block extremities. The final value is calculated from the extremities intervals based on the monotonicity of r_{gf} with respect to each of its parameters. In our example we have:

$$first\_interval = r_{gf}(Block.x, Block.y)$$

$$last\_interval = r_{gf}(Block.x + BlockDim.x - 1, Block.y + BlockDim.y - 1)$$
$$block\_mem_f(Block.x, Block.y) = [LB(first\_interval), UB(last\_interval]$$

where LB selects the lower bound of an interval and UB selects the upper bound. After performing the calculations we can summarise the results for the filter in the following formulas:

$$block\_mem_f = share_{xf} = share_{yf} = [0, f_h * f_w - 1]$$

The interpretation of this result is as follows: *It is recommended that the access to the memory area pointed to by* `filter` *from line* `15` *is to be done from shared memory. The motivation is that each thread within a block will reuse the designated memory. For each thread block the section determined by* $block\_mem_f$ *needs to be copied to shared memory.*

Increasing precision

The analysis gets more interesting, but also more complicated when the interval expressions depend on the grid variables. Consider for example the load operation from line 17. The corresponding interval function is:

$$\begin{aligned} r_{img}(Grid.x, Grid.y) \quad = \quad & [3 * (Grid.y - \tfrac{f_h}{2}) * i_w + Grid.x - 3 * \tfrac{f_w}{2}, \\ & 3 * (Grid.y + \tfrac{f_h}{2}) * i_w + Grid.x + 3 * \tfrac{f_w}{2}] \end{aligned}$$

where $i_w = image\_width$, $f_h = filter\_height$ and $f_w = filter\_width$. Applying the above procedure we get the following block requirement:

$$\begin{aligned} bloc\_mem_{img}(Block.x, Block.y) \quad = \quad & [3 * (Block.y - \tfrac{f_h}{2}) * i_w + Block.x - 3 * \tfrac{f_w}{2}, \\ & 3 * (Block.y + BlockDim.y - 1 + \tfrac{f_h}{2}) * i_w + \\ & Block.x + BlockDim.x - 1 + 3 * \tfrac{f_w}{2}] \end{aligned}$$

A closer look on the above expression will reveal a major precision issue. The real memory requirement for a given block is actually the subimage it has to process padded with filter size pixels. But instead of returning this, the approach so far returns a bit more than $BlockDim.y$ rows. Figure 5.7 highlights this problem.

The reason is that the accesses to the image are linearised and the 2D information cannot be obtained directly. We solve this issue by reconstructing the 2D access based on the access expression as returned by the expression reconstruction analysis. In this case the result will be a 2D pattern and will give the sub-matrix that should be copied to the shared memory. Section 5.2.1 presents details about the 2D reconstruction process.

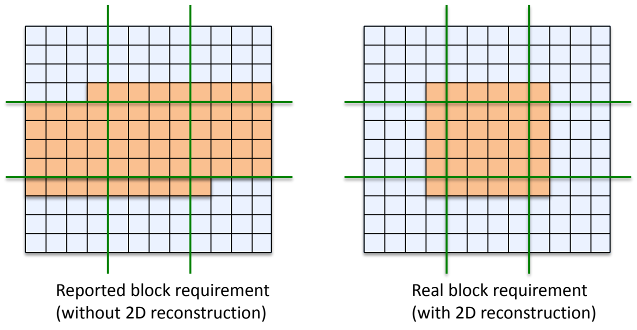


Figure 5.7: Impact of linearised accesses on shared memory analysis preciseness

Detection algorithm

The following algorithm summarises the procedure for detecting shared memory opportunities for a given memory operation. The arguments are the access interval section (r) and the access expression (f).

Algorithm 2 Shared memory algorithm

```
procedure GET_SHARED_MEMORY_INFO($r, f$)                                  ▷ $r$ = the access section interval
    if the offset expression $f$ is a 2D pattern then                     ▷ $f$ = the access expression
        compute sharing information from 2D pattern
    else
        $r_g \leftarrow r$ restricted to grid coordinates
        $share_x \leftarrow r_{gf}(Grid.x, Grid.y) \cap r_{gf}(Grid.x+1, Grid.y)$
        $share_y \leftarrow r_{gf}(Grid.x, Grid.y) \cap r_{gf}(Grid.x, Grid.y+1)$
        if $is\_empty(share_x)$ and $is\_empty(share_y)$ then return no sharing opportunity
        else
            $mono \leftarrow$ MONOTONICITY_BY_SET($r_g, Grid.x, Grid.y$)
            if $mono$ is monotonic then                                     ▷ f is monotonic
                $first\_interval \leftarrow r_{gf}(Block.x, Block.y)$
                $last\_interval \leftarrow r_{gf}(Block.x + BlockDim.x - 1, Block.y + BlockDim.y - 1)$
                if $mono$ is increasing then
                    $block\_mem_f(Block.x, Block.y) = [LB(first\_interval), UB(last\_interval]$
                else
                    $block\_mem_f(Block.x, Block.y) = [LB(last\_interval), UB(first\_interval]$
                end if
                return $(share_x, share_y, block_mem)$
            else                                                            ▷ f is not monotonic
                return $(share_x, share_y, unknown\ block\ requirements)$
            end if
        end if
    end if
end procedure
```

Special cases

As in coalescing analysis several cases require special attention:

1. monotonicity cannot be determined for every expression or the expression might be non-monotonic
2. in the presence of conditionals, multiple expressions are possible for the same memory operation

If the monotonicity of the interval bounds cannot be determined or they are non-monotonic, then the analysis cannot determine the memory requirements for the entire block. This is a known limitation of the current approach.

In contrast with coalescing analysis, multiple expressions do not pose any problems. Each possible expression is analysed individually and a separate sharing report is done for it.

5.2.1 Reconstruction of 2D access

Reconstructing the 2D accesses from linearised accesses is vital step in the shared memory analysis for two reasons:

conceptual reason There are a lot of cases where developers linearise a natural 2D access. As we have seen in the convolution example this can influence analysis precision.

practical reason If not linearised by developer, accesses will be linearised by compiler. Thus any intermediate representation will contain flat accesses. In particular, the representation we are working on expose this property (see Section 8.2).

The current system recognizes 2D patterns and sub-matrix patterns with arbitrary x-stride. We will present only the sub-matrix reconstruction process. Simple 2D patterns are recognized in a similar way.

The reconstruction algorithm takes as input the access expression, f, and tries to recognize the following pattern:

$$f = \mathit{offset} + \mathit{stride} * i_x + \mathit{stride} * i_y * \mathit{width} + \mathit{stride} * m_x + \mathit{stride} * m_y * \mathit{width}$$

where:

offset	an arbitrary offset, from which the given access starts to expose a 2D access pattern
i_x, i_y	the inner loops induction variables
m_x, m_y	the outer (main) loops induction variables
width	the width of the matrix
stride	the stride for the m_x induction variable

Basically m_x and m_y selects the element of the matrix from which the sub-matrix determined by the iteration space of i_x and i_y is computed. The following piece of code clarifies the detected pattern:

```
for (mx = 0; mx < width; mx += stride)
  for (my = 0; my < height; my += 1)
  {
    for (ix = 0; ix < inner_width; ix += 1)
      for (iy = 0; iy < inner_height; iy += 1)
      {
        ... = a[offset + stride * ix + stride * iy * width +
                stride * mx + stride * my * width]
      }
  }
```

The algorithm proceeds in two steps. The first step collects the induction variables from the given expression. The second step identifies which of the induction variables can play the roles of the above quadruple (i_x, i_y, m_y, m_y). It does this by considering each possible combination and applying a matching procedure against the 2D pattern. If no matching is found then the detection fails.

It is important to mention that given an arbitrary symbolic expression we can always identify which variables are induction variables. For more details on this capability please consult Section 8.4.

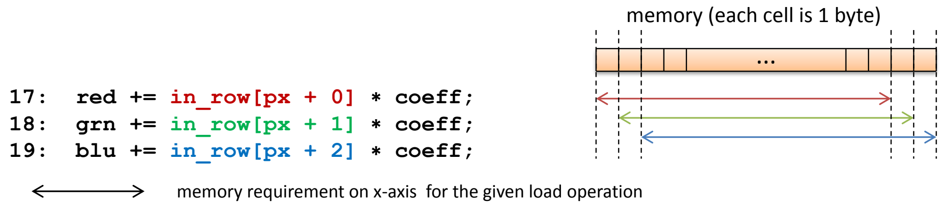
5.2.2 Reducing the need for shared memory

Because shared memory is limited and also affects occupancy it is important to allocate only as much as needed and re-use as much as possible. In order to satisfy this necessity and improve the overall shared memory usage our system does the following:

1. it reconstructs 2D access from linearised accesses
2. it groups shared memory by pointer variable according to some specified rules

We already discussed the necessity and benefits of 2D reconstructing in the previous section. During this section we focus on explaining what grouping means and how it is done.

The approach described so far focuses on detecting shared memory opportunities across threads. Basically it detects memory ranges that are accessed by adjacent threads and marks these regions as candidates for shared memory allocation. We can improve on this if we also look inside threads and take into account memory ranges that are accessed by different operations. Consider for example the memory loads from lines 17, 18 and 19 from the convolution example. The shared memory requirements for each of them is presented in Figure 5.8. For brevity the figure depicts only the requirement on the x-axis.



```
17:  red += in_row[px + 0] * coeff;
18:  grn += in_row[px + 1] * coeff;
19:  blu += in_row[px + 2] * coeff;
```

memory (each cell is 1 byte)
...
memory requirement on x-axis for the given load operation

Figure 5.8: Overlapping memory areas for operations inside a thread

It is easy to see that they mostly access the same area. They only differ in the first and last bytes which correspond to different colours. We can allocate much less shared memory if, instead of allocating three arrays with the same size but slightly different offsets, we allocate just one bigger array that will span across all three arrays. In this particular case we will reduce the need of shared memory by a factor of 3.

Our system implements two grouping schemes:

1. group two shared memory areas only if the difference between the starting offsets (ending offsets) is less then a given constant.
2. group all shared memory areas that belong to the same pointer. Although this method can add holes (unused areas) in the shared memory it is very useful when the starting offsets or ending offsets cannot be compared due to lack of symbolic information.

The grouping algorithm is based on the following idea. First, group the shared memory areas that belongs to the same pointer. Then, iterate through each group and accumulate a new memory interval based on the current value and on the selected scheme. If the current value breaks the scheme rule skip it but add it to a temporary queue, private per group. After the iteration finishes repeat the process starting form the temporary queues. Stop when there are no more elements in the temporary queues.

6 Instantiating the results

Where the previous two phases (identification and optimisation) strive for generality, the goal of the instantiation phase is to give precise numeric information about a particular program execution. It does this by instantiating the algebraic models created in the previous steps with information collected at run-time through dynamic analysis. The phase is important for two main reasons:

1. it helps estimating the efficiency of the transformation
2. it helps the developer to better understand his program

As we have already seen having algebraic information about the program is mandatory in order to reason about GPU transformations. But only with them, one could hardly make predictions about the performance of a given transformation. For example, in order to estimate the communication overhead one needs to know exactly how much data needs to be copied to the device. Another example is estimating how much time a memory operation will take. This depends on the coalescing factor and on the type of memory that is accessed. With only algebraic expressions one cannot estimate this precisely. Instantiating the results and getting numeric values out of symbolic expressions is thus the first step towards a performance estimation module. It is important to mention that the design and development of such a module is out of the scope of this research. Though, providing a way to instantiate the results sets up the starting point for such a module.

In what concerns the second reason, we note that sometimes the results can be too abstract for the user. Because the analysis may produce long and complicated symbolic expressions, their interpretation and consequences might be hard to understand. Consider for example deciding what data should go to shared memory. Shared memory analysis usually reports multiple opportunities and they cannot be compared with each other in every case. One solution is to insert checks and decide on what should go where at runtime. Another solution is to observe the behaviour of the program on a representative set of tests and collect the values of the parameters that influence shared memory requirements. Using this information one might decide based on a statistical analysis what opportunity is most likely to be the most beneficial. Being able to instantiate the algebraic results is a requirement for a such approach.

6.1 Instantiation procedure

Although evaluating a symbolic expression in a given context is a straightforward thing to do, finding the values for the context parameters is not. There are two main difficulties that we must overcome. First, in most cases it is impossible for a dynamic analysis to keep track of values for all program variables. Usually only a subset of values will be tracked and stored. The current infrastructure for example, can provide us only with the average loop counts. This means that we have to find a way to relate the loop counts with the actual parameters of the symbolic expressions. The second difficulty relates to the extra inbound variables that can be introduced by the compiler (see Section 4.2 for more details). The values of these parameters are not tracked by the current dynamic analysis. In order to infer their values, we also have to relate them with real inbound parameters.

We solve these problems by creating a system of equations that correlates the available information (the loop counts) with the missing information (inbound parameters and artificial variables). The instantiation procedure consists of the following steps:

1. Use dynamic analysis to get available loop counts. This step reuses the existing infrastructure of vfEmbedded.
2. Create a system of equations that relates the loop counts with the kernel parameters (the inbounds). The equations are created based on the information produced by the expression reconstruction analysis. For a better understanding of this step consider the example below.
3. Solve the system and get the parameters values. This is handled by the symbolic computation engine presented in Section 7. The exact solving procedure and its limitations are described in Section 7.2.
4. Based on the above values create an evaluation context to which add grid configuration parameters (i.e., block size).
5. Evaluate the results in the above context. This means evaluating the symbolic expressions associated with the following: inbound and outbound copy expressions, coalescing factors and shared memory requirements. Evaluating a symbolic expression is done through constant folding after all variables are replaced by the corresponding parameters values.

The rest of this section illustrate the procedure through a simple example. For clarity and conciseness we will present the example at the source code level. We will study a program that adds every fourth position of two vectors and stores the result in another array. Suppose that our goal in this context is to estimate the communication overhead to transfer data to and from the device. The orginal program is listed below:

```
int* a, b, c;
int n;
// initialize values...
for(int i = -n; i < n; i += 4) {
  a[i + 2*n] = b[i + n] + c[i + n];
}
```

After normalization the program is transformed to:

```
int tmp = n > 0 ? 0 : 2 * n / 4;
for(int k = 0; k < tmp; k += 1) {
  int i = k * 4 - n;
  a[i + 2*n] = b[i + n] + c[i + n];
}
```

After performing the analyses on this program variant we get the following information. The kernel parameters are: `a`, `b`, `c`, `n` and the compiler introduced `tmp`. The copy expression for the pointer parameters are: $copy_a = [4*n, 4*tmp+4*n-4]$ and $copy_b = copy_c = [0, 4*tmp-4]$. Recall that the copy expression are expressed in bytes, hence the factor of 4.

Let us suppose that we want to instantiate these results for an execution where the loop count is 100. The first step is to create an equation that relates the loop bound with the actual loop count: $tmp = 100$. The next step is to infer the relation between tmp and n. For this we look above the loop code that we analysed and reconstruct the following possible expressions: $tmp = 0$ or $tmp = \frac{2*n}{4}$. Since we already know that $tmp = 100$ it is clear that the option $tmp = 0$ is not a valid one. We discard it and form the following system:

$$\begin{aligned} tmp &= 100 \\ tmp &= \frac{2*n}{4} \end{aligned}$$

Solving this will yield $n = 200$. We can now evaluate the input and output copy expressions to $copy_a = [200, 596]$, $copy_b = copy_c = [0, 396]$. Having the precise memory bounds, estimating the communication overhead is just a matter of knowing the bandwidth from the host memory to the global memory.

There are two things that are worth mentioning with respect to the instantiation procedure. First, by having only the loop counts it is not always possible to fully instantiate the results. There might be input parameters that are not related in any way with the loop counts. For these, it is impossible to find values with the above procedure. Nevertheless, if they are provided, further re-instantiation is possible. Second, in practice we usually deal with more complicated expressions which are not always linear. To this end, our system is capable of solving pseudo non-linear systems. For the exact solving capabilities and limitations we redirect the reader to Section 7.2.

6.2 Trading soundness for precision

One should keep in mind that most of the information about the kernel is gathered through static analysis techniques. These provide only a safe and computable approximation of run-time values. Sometimes the approximation can be too imprecise and become useless in a practical situation. In our case, this can happen for example when expressions become too complex to efficiently represent. In such cases the static analyser uses conservative bounds that are usually too imprecise.

Take for example the program below. It applies a 1D signal convolution to an input signal. The specific feature is that in the case of outbounds access (for extremities) it adds the values from a special area stored at the end of the input array (line 6: `idx = size+k`).

```
1  for(i = 0; i < size; i ++) {
2    x = 0;
3    for(k = i - slice; k < i + slice; k++) {
4        int idx = 0;
5        if (k >= 0 && k < size)
6          idx = slice + k;
7        else
8          idx = size + k;
9        x += in[idx];
10   }
11   out[i] = x;
12 }
```

The main problem of the above program is that `size` and `slice` are incomparable as symbolic expressions. This, combined with some limitations in our symbolic computation engine (see Section 7.1), makes the symbolic range analysis to fail to compute the interval in which `idx` can take values at line 9. Thus, the analysis will not be able to provide the copy expression for the `in` variable and will also fail to provide the exact shared memory requirements. It will only report that `in` is a pointer variable and that there is a sharing opportunity for the load at line 9 but will not provide more details. Although this is a particular limitation in our system, cases like this make an inherent limitation of the static approach in program analysis.

When this kind of situations arise, we still want to be able to offer as much information as possible. Even if this means to lose soundness properties. To achieve this goal we designed the following procedure which will trade soundness for precision. The procedure makes possible to still get precise results when the static analyses fail. The trade-off is that the results are tied to a particular program execution.

The main idea is to reverse the order in which static analysis and instantiation is done. We can do this because computing values for inbound parameters given a concrete execution is independent of the static analyses that may fail. In particular, it depends only on expression reconstruction analysis which always provides exact expressions for variables. We redirect the reader to Section 8.4 to discover how this is done.

After inferring values for the inbound parameters we can re-run the static analyses to re-evaluate part of the algebraic model. Doing so will basically avoid the symbolic computation limitations. This will produce the precise results that we are interested in. To this end all of our static analyses support *injection* of values. During the analysis, *injected values* take precedence over the values that will otherwise be computed by the natural flow.

7 Symbolic computation

Symbolic computation is about expressing and solving mathematical problems the way one would think about mathematical problems - using variables, mathematical formulas, and mathematical functions. In more precise terms, it allows for manipulation of mathematical equations and expressions in symbolic form, as opposed to manipulating the approximations of specific numerical quantities represented by those symbols. The main advantage of symbolic computation is that variables can be kept as unknowns throughout the calculations. There is no need assign values to them unless one wants to get a numeric result out of the calculation.

In the context of GPU mapping, the need for symbolic computation arises from the necessity to work with expressions parametrised by the input and grid parameters. The following three cases illustrate this necessity. First, consider reporting how much data needs to be copied from host to device and the other way around. Only in very few cases this will be a precise number and most of the times it will depend on various input parameters. Second, think about determining how much shared memory needs to be allocated in a particular context. This will depend at least on the block size (which is a grid parameter) but most likely also on the input parameters. Third, in order to find if a memory access will be coalesced or not we have to investigate offset expressions which in most of the cases will depend on various induction variables.

It is clear from these examples that symbolic computation is a must in our context. To this end we developed a symbolic computation engine able to manipulate arbitrary arithmetic expressions. In what follows we will present the engine capabilities and detail on its algorithmic complexity.

7.1 Expressiveness

The domain of the symbolic expressions is the set of rational numbers. Besides the classical arithmetic operations the expressions also support $getMin$ and $getMax$ operations. These accept arbitrary expressions as input; but because not every two expressions are comparable the result may be an explicit min/ max expression. In particular we can represent expressions like $x * y + \frac{a}{2}$, $min(0, a)$ or $max(a * a, b * c)$. The complete set of basic operations is presented below.

Basic operation	Expression example
addition	$10 + a + b$
subtraction	$a - b$
multiplication	$5 * a + k * a + k * b$
division	$3 * \frac{a}{2}$
left shift	$a \ll 3 = a * 2^3$
right shit	$a \gg 3 = \frac{a}{2^3}$
get minimum	$min(a, b + c)$
get maximum	$max(0, 3 * d)$

An important aspect to be remarked from the examples above is that expressions are not restricted to affine or polynomial functions. This is a key element which makes our approach stronger than the polyhedral approach which can only handle affine expressions [5, 8, 9].

Restrictions. In order to allow efficient implementation for the supported operations we impose some restrictions on the operations. The restrictions are chosen in such a way as to balance the expressive power with the algorithmic and implementation complexity:

- In a division operation the divisor must be a numeric constant.
- In shifting operations the right hand side must be a numeric constant.
- min/max expression are allowed only at the top level. This means that if min/max expressions are used in any operation and the result cannot be expressed again as min/max top level only expression an exception will be thrown.

Normal form. In what concerns the representation we enforce a normal form which acts like an expression invariant during the computations. The normal form divides the expressions into two categories: *simple expressions* and *min/max expressions*. As mentioned above, min/max expressions cannot be nested. Simple expression are represented using a *sum of products* normal form. The sum is a list of products where each product is a list of variables and constants. Both, the sum and the products have their elements sorted lexicographically with constants coming first. The advantage of keeping the product and sum elements sorted is that it allows for efficient implementation of the *equiv_products* and *group_by* procedures described in the addition algorithm [4] ($O(n)$ instead of $O(n^2)$). At any time, an expression in normal form is maximally reduced. In other words, each product contains at most one numeric constant, and all min/max are pushed to the top.

It is important to mention that each expression carries a sign environment which associate to each variable present in the expression a list of possible signs. The sign environment for the result of an operation is obtained by merging the sign environments of the operands. At any time the sign environment of an expression can be overwritten allowing to make the expressions context sensitive with respect to the signs of their variables. This can make the difference between success and failure when it comes to applying certain operations in different contexts. To illustrate this, consider the example below. By setting n to be positive at line 2 we can further compute expression like $getMax(a, getMax(x, 1))$. By contrast, if we would not have this capability we would end up in a case similar to the one from line 8 where $getMax(a, getMax(z, 1))$ cannot be computed without breaking the normal form.

```
if (n > 0)
  y = n;      // n can be set to be strictly positive here

x = y + 1;    // getMax(x,1) = x
              //   because we know that in this context n is strictly positive
              // getMax(a,getMax(x,1)) = max(a,x) -> OK

z = n + 1;    // getMax(z,1) = max(z,1)
              //   because we don't know anything about n
              // getMax(a,getMax(z,1)) = max(a,max(z,1)) ->
              //   not OK, violation of the normal form
```

7.2 Operations and algorithms

Basic operations. The basic operation on expressions $+, -, *, /, \ll, \gg, getMin, getMax$ have the straightforward mathematical algorithms. What is worth mentioning is that only $+$, $*$, $getMin$ and $getMax$ are true operations. The others, can be implemented in terms of these. Subtraction is implemented based on addition and division, left shift and right shift in terms of multiplication. In what follows we highlight only the addition and $getMax$ algorithm.

Addition is important because it also serves as the normalization procedure. Adding 0 to any expression will normalize it without incurring any performance penalty. The algorithm proceeds in two steps. The first step is to group the equivalent products from the two operands. In the second step, each group of products is maximally reduced through constant folding. Two products are considered equivalent (*equiv_products* in the algorithm) if they differ only in their numeric constant (i.e., have exactly the same variables with the same cardinality). To clarify the grouping consider the next example. If $e = 1 + a$ and $f = 3 * a + b$ then the algorithm simulated the following steps: $e + f = (1 + a) + (3 * a + b) = (1) + (a + 3 * a) + (b) = 1 + 4 * a + b$.

Algorithm 3 Symbolic expression addition

```
procedure ADD($f$, $g$)
    $expanded\_sum \leftarrow concatenate(f, g)$
    $groups \leftarrow group\_by(equiv\_products, expanded\_sum)$
    for all $group$ in $groups$ do
        $sum\_elem \leftarrow$ reduce $group$ by constant folding
        put $sum\_elem$ in $sum$
    end for
    $sign\_env(sum) \leftarrow sign\_env(f) \cup sign\_env(g)$
    return $sum$
end procedure
```

The $getMax$ algorithm is based on the expression comparison algorithm which is explained later in this section. Though it is simple, special attention must be given to maintain the normal form. To this end, special patterns that otherwise will cause the operation to fail are recognized and handled.

Expression signs. As mentioned, each expression carries a sign environment for its variables. In the environment each variable has associated a set of possible signs. The sign for the entire expression is

Algorithm 4 Maximum of two symbolic expression

```
procedure getMax($f$, $g$)
    $max \leftarrow$ case $f, g$ of
        $(a, min(a,b)) \rightarrow b$
        $(a, max(a,b)) \rightarrow max(a,b)$
        $(min(a,b), max(a,b)) \rightarrow max(a,b)$
        $(a,b) \rightarrow \begin{cases} a & \text{if } a \geq b \\ b & \text{if } a \leq b \\ max(a,b) & \text{if } a \text{ and } b \text{ are incomparable} \end{cases}$
    return $max$
end procedure
```

computed by substituting variables with their possible signs and then resolving the expressions according to the rules below. We present only the simple rules involving the $\{+\}$ sign. The rest of the rules followed the same pattern and can be easily derived. All the rules are lifted to operate on sets by considering all possible combinations and then joining the results.

$$\begin{array}{ccccc} \{+\} & + & \{0\} & = & \{+\} \\ \{+\} & + & \{+\} & = & \{+\} \\ \{+\} & + & \{-\} & = & \{-,0,+\} \\ \{+\} & * & \{0\} & = & \{0\} \\ \{+\} & * & \{+\} & = & \{+\} \\ \{+\} & * & \{-\} & = & \{-\} \end{array}$$

Note that the above procedure may fail to determine the sign of an expression even if this is obvious from the context. Take for example $f = a - 1$ with $sign(a) = \{+\}$. By applying the rules we will get $sign(f) = \{+\} + \{-\} = \{-, 0, +\}$. Though, it is clear that since a is a positive integer its value is greater or equal to 1 and thus we should have $sign(f) = \{0, +\}$. This can make the difference between a precise value and a totally imprecise value (TOP) when static analysis is concerned. We handle these kinds of scenarios by computing the sign based on a *substitution method*. The general idea is to inspect the monotonicity of the expression with respect to one of its variables. Based on the monotonicity and on the variable sign we compute the interval in which the expression can take values by substituting the variable with its minimum or maximum integer value. The sign of the expression if derived from its value interval in a natural way. Since an expression may have multiple variables, the process is repeated for all of them and the results are merged. In general, if f is an expression and a is one of its variables, the rules that guide the substitution are detailed in the table below:

monotonicity of f with respect to a	$sign(a)$	$f(a)$ value interval
increasing	$\{+\}$	$[f(min(a)), +\infty]$
increasing	$\{-\}$	$[-\infty, f(max(a))]$
decreasing	$\{-\}$	$[f(max(a)), +\infty]$
decreasing	$\{+\}$	$[-\infty, f(min(a))]$

Expression comparison. Two expressions f and g are compared by subtracting one from the other and computing the possible signs of the result. If the result can have any sign then the expressions are incomparable.

Monotonicity. Expressions can be queried for monotonicity properties with respect to a specified variable. If $f(a)$ is an expression parametrised by a, the monotonicity of f with respect to a is computed based on the following decision algorithm:

Algorithm 5 Computes the monotonicity of f with respect to a

```
procedure MONOTONICITY($f$, $a$)
    $\delta \leftarrow$ a strictly positive fresh variable
    if $f(a+\delta)$ cannot compare with $f(a)$ then return unknown monotonicity
    else if $f(a+\delta) = f(a)$ then return constant monotonic
    else if $f(a+\delta) \geq f(a)$ then return increasing monotonic
    else if $f(a+\delta) \leq f(a)$ then return decreasing monotonic
    else return non monotonic
    end if
end procedure
```

The monotonicity decision algorithm is lifted to compute the monotonicity of an expression f with respect to to a set of variables $\mathcal{V}$ in the following way:

Algorithm 6 Computes the monotonicity of f with respect to the set $\mathcal{V}$

```
procedure MONOTONICITY_BY_SET($f$, $\mathcal{V}$)
    if $\forall u, v \in \mathcal{V}, monotonicity(f, v) = monotonicity(f, u)$ then
        $a \leftarrow choose(\mathcal{V})$
        return $monotonicity(f, a)$
    else if $\exists v \in \mathcal{V}, monotonicity(f, v) =$ unknown monotonicity then
        return unknown monotonicity
    else return not monotonic
    end if
end procedure
```

Note that the monotonicity algorithm requires expression comparison which in turn requires expression sign computation. If for the sign computation we use the *substitution method* (which needs again to compute monotonicity) we create a cyclic dependency. We break this cycle by computing the signs required by the monotonicity algorithm only through the straightforward method.

Equation solving. Although we can represent non-linear equations, the current system can only solve linear systems. The solving is done through the substitution method for which we present an example below.

1. Start with the system	$x + y = 3$ $x - y = 1$
2. Isolate x in the first equation	$x = 3 - y$
3. Replace x in the second equation	$3 - 2 * y = 1$
4. Solve and find that	$y = 1$ $x = 2$

Table 7.1: Example of solving a linear system by substitution method

We extend the basic substitution method to handle some non-linear equation systems through the following procedure. First, we partition the set of equations into linear and non-linear equations. We solve the linear equations and then substitute the variables in the non-linear equations based on the intermediate results. We repeat the procedure until no linear equations can be extracted. The following algorithm showcase this procedure.

Algorithm 7 Solve a system of equations

```
procedure SOLVE($eqs$)
    $linear\_system, nonlinear\_system \leftarrow$ partition the equation from $eqs$ based on linearity
    $inter\_result \leftarrow$ solve $linear\_system$ by the substitution method
    $new\_eqs \leftarrow$ substitute variables in $nonlinear\_system$ based on $inter\_result$ environment
    $result \leftarrow$ SOLVE($new\_eqs$)
    return $inter\_result \cup result$
end procedure
```

Sign equation solving. It proceeds in the same way as normal equation solving but all operations are done at the sign level instead of at the numerical level.

7.3 Algorithmic complexity

In this section we will analyse the algorithmic complexity for the main basic operations: addition, multiplication and maximum. For the others similar reasoning applies. During the complexity analysis we adopt the following notations:

$$\begin{array}{lcl}
e_1 + e_2 & = & \text{the operation to be analysed} \\
e_i & = & \text{the } i\text{th operand} \\
n_i & = & \text{the number of variables from } e_i \\
n & = & \sum_i n_i \\
n' & = & max_i(n_i) \\
m_i & = & \text{the number of sum terms (products) from } e_i \\
m & = & \sum_i m_i
\end{array}$$

Given this, the maximum length of a product can be $n_i + 1$ and there can be at most 2^{n_i} sum terms. In practice the actual number of sum terms is much less than 2^{n_i}. Because of this it is more useful to reason about complexity in terms of the number of products, i.e., m_i.

For addition, we first need to group the products of the two expressions based on the *equiv_products* predicate (products that have exactly the same variables but differ in their numeric constant). Because products are sorted, the equivalence of two products in terms of *equiv_products* can be tested in $O(max(n_1, n_2))$. Given that both expressions have their products sorted, the grouping can further be done in $O((m_1 + m_2) * max(n_1, n_2))$. What follows is the constant folding for all groups of products. For a given product this can be done in constant time because each product has at most one numeric constant in its composition. Since we know that there can be at most $m_1 + m_2$ products in a group this step amounts to $O(m_1 + m_2)$. Summing up the results we get the worst-case complexity for addition: $O(m_1+m_2)+O((m_1+m_2)*max(n_1, n_2)) = O((m_1+m_2)*max(n_1, n_2)) = O(m*n')$. Our experiments revealed that in practice m is usual a very small constant (< 10 on our examples). Thus we can argue that the complexity of addition is actually linear in the number of variables of the expressions.

For multiplication we basically need to distribute the product over the sum. We have m_1 product terms in the first operand and m_2 in the second. Thus, this will take $O(m_1 * m_2)$. Unfortunately doing this will break the order and thus the resulted sum needs to be sorted again. Because now we have $m_1 * m_2$ terms and a single comparison is done in $O(n')$ this will take $O(m_1 * m_2 * log(m_1 * m_2) * n')$. Grouping the equivalent products and applying constant folding can be done in $O(m_1 * m_2 * n')$, based on the same reasoning as above. This leads us to a total worst-case complexity of $O(m_1 * m_2 * (log(m_1 * m_2)) * n')$.

Getting the maximum out of two expression amounts to find the sign of their difference. The complexity for subtraction is the same as for addition, thus $O(m*n')$. Computing the sign for the resulted expression requires a traversal of the entire expression and thus requires $O(m * n)$ steps. Combining this we get $O(m * n') + O(m * n) = O(m * n)$.

The table below summarises the complexity for all available operations:

Operation	**Worst-case complexity**
addition subtraction	$O(m * n')$
multiplication division left shift right shit	$O(m_1 * m_2 * (log(m_1 * m_2) + n'))$
get minimum get maximum	$O(m * n)$
expression sign expression comparison	$O(m_1 * n_1)$
monotonicity	$O(m * n)$

7.4 Soundness discussion

Because the domain of symbolic expressions is actually $\mathbb{Q}$ and not $\mathbb{Z}$ some soundness issues arise when division is used. Consider the following expression:

$$\frac{a}{2} + \frac{a}{2}$$

The actual result of this expression depends on the domain set, $\mathbb{Q}$ or $\mathbb{Z}$, and on the parity of a:

$$\frac{a}{2} + \frac{a}{2} = \begin{cases} a & \text{if } a \in \mathbb{Q} \\ a & \text{if } a \text{ is even, } a \in \mathbb{Z} \\ a - 1 & \text{if } a \text{ is odd, } a \in \mathbb{Z} \end{cases}$$

In general, the results produced by symbolic computations are not sound when division is used and the divisor does not divide the dividend. Although this is a limitation of our approach it is important to note that it is not a major one. This is because we mainly use symbolic computations to reason about memory access operations. In practice it rarely happens that one uses division to compute the offsets for the memory accesses.

8 Analyses

8.1 Overview

This section presents the static analysis framework that powers up the top-level features. In what concerns the dynamic analysis framework, we reuse the existing infrastructure of vfEmbedded. For this reason we will not go into technical details. It suffices to mention that the data we collect dynamically are data dependencies and loop counts. Data dependencies are used in the identification process in order to reject the loops which have carried dependencies. Loop counts are used in the instantiation process in order to infer the values of the inbound variables. The integration of the two frameworks, dynamic and static, is done during the identification phase (Section 4) and the instantiation phase (Section 6).

The static analysis framework consists of four analyses. Table 8.1 presents their goals and usages while Figure 8.1 illustrate their dependencies with respect to each other.

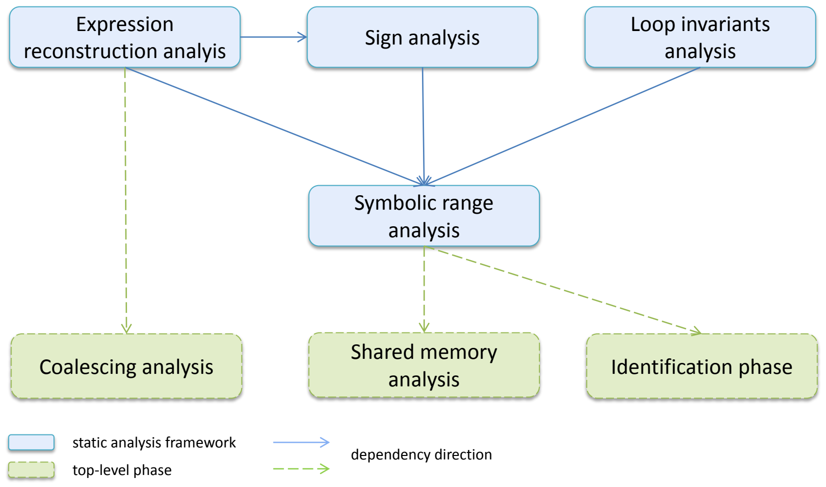


Figure 8.1: Static analyses framework

We can classify the analyses into two groups: *main analyses* and *support analyses*. The main analyses are mandatory for our approach and their results are used to compute mapping information. These are *expression reconstruction analysis* and *symbolic range analysis*. Support analyses are actually optional and do not affect the functionality of the system or of other analyses. Their main goal is to increase precision of other analyses. The support analyses are *sign analysis* and *loop invariant analysis*. We emphasis that although optional from a conceptual point of view, support analyses are a must in practice in order to achieve a decent level of precision.

Analysis	**Goal**	**Used to**
Symbolic range analysis	determines the minimum and maximum value for every variable	• compute inbound/outbound copy expressions • detect shared memory optimisations
Expression reconstruction analysis	reconstructs the symbolic expression of a given variable in terms of the other program variables (e.g., inbounds, loop invariants)	• detect coalescing issues • reconstruct matrix and submatrix access • support other analyses • basic aliasing analysis
Sign analysis	computes the possible signs that variables can have	• detect expressions sign and monotonicity • increase precision of other analyses
Loop invariant analysis	detects variables whose value does not change during the loop iterations	• substitute load operation with variables • increase precision of other analyses

Table 8.1: Static analyses goals and usages

Several features are common to all analyses. First of all they were designed using abstraction interpretation. Second, their specification is based on monotone frameworks. Next, they are flow-sensitive and intra-procedural. Ongoing work intends to lift the analyses to context-sensitive inter-procedural; Section 8.7 provides more details on this. And finally, the analyses are performed on an intermediate representation of the program (Section 8.2). Nevertheless it is important to mention that the techniques are representation agnostic. One can use the same ideas and have the analyses implemented at the source code level for example. Our choice for the intermediate representation was influenced by integration goals with respect to vfEmbedded infrastructure.

8.2 Intermediate program representation

The intermediate representation is a modified version of the control data flow graph (CDFG) of the program that is output by the compiler. The original CDFG does not have explicit control structure, thus making things more complicated for flow-sensitive static analyses. We fixed this by making *if then else* structures explicit in the graph.

The CDFG is also in *static single assignment form* (SSA) [13]. The main feature of SSA representation is that the definition-use chains of the program are explicit in the representation: each variable has at most one definition (this means that it is assigned only once). The transformation to SSA form amounts to renaming the variables and introducing special assignments at the points where control flow merges.

The special assignments are called ϕ-functions. Each argument of a ϕ-function identifies one of the program points from where the control flow might have come. The assigning statements take the form $x = \phi(x_1, \ldots, x_n)$ and the idea is the value of x will equal the value of x_i whenever the control flow comes from the ith predecessor. Figure 8.2 presents a SSA transformation example.

```
if (k > 0)
  x = y
else
  x = z
a = x
```

a) Normal form

$\Longrightarrow$

```
if (k > 0)
  $x_1$ = y
else
  $x_2$ = z
x = $\phi$($x_1$,$x_2$)
a = x
```

b) SSA equivalent form

Figure 8.2: Transformation to SSA form

Language

We can formalize the intermediate representation into a small language. This serves as a succinct description of the representation and helps us to explain the static analyses. The language abstract syntax is presented in Table 8.2. In order to increase readability we use specific operator symbols and sometimes redundant renaming.

We emphasise that the language is not intended to represent all features of ANSI C99. It is meant to be a simplification that helps us present the static analyses. In particular we do not represent features like structure type definitions or functions with variable numbers of arguments. They do not add value to our future explanations; thus we omit them. Also, one may notice that we do not represent control structure like switch-case structures or while loops. We only employ conditional and unconditional jumps. It is easy to prove that the other control structures can be defined in terms of them. We also do not concern ourselves to explicitly represent global and static variables. Operations on them are translated to loads (stores) from (to) specific memory areas. Because of this they are implicitly encoded within the language.

From a structure perspective a program is a list of functions. A function is represented in terms of basic blocks. A basic block is a group of statements that does not contain jump instructions. Jumping is allowed only the end of basic blocks. This contrast with how programs are usually represented in source code: in terms of blocks of statements that can have jumps at arbitrary points. Each basic block has an associated label that can be used to execute jumps.

There are tree types of statements: assignments, store operations and function calls. Expressions can be classified into trivial expressions (literals and variables) arithmetic/comparison expressions and complex expressions. In what concerns arithmetic expressions one may note that we did not include logical operators (i.e and, or, not) but only their bitwise relatives. The reason is that they can easily be expressed in terms of the bitwise operators. The complex expressions are the following. The ternary conditional expression (eg. *cond* `?` *a* `:` *b*) selects the first expression if the condition is evaluated to true or the second one otherwise. Cast expressions (eg. `cast (int*)` *p*) are used in order to convert between different types. Load expressions (eg. `load`(p+1)) fetch values from the specified address in memory (the address is an arbitrary pointer expression). A special instruction is used to perform pointer arithmetic: `ptr_add`. Valid arguments are a pointer expression and an arbitrary offset values. We use a special operation and not a simple add because the actual address computation depends on the pointer type. For example, `p + 1` will increase the address of `p` by 4 bytes if `p` is of type `int[10]` and by 40 bytes if `p` is of type `int[10][10]`.

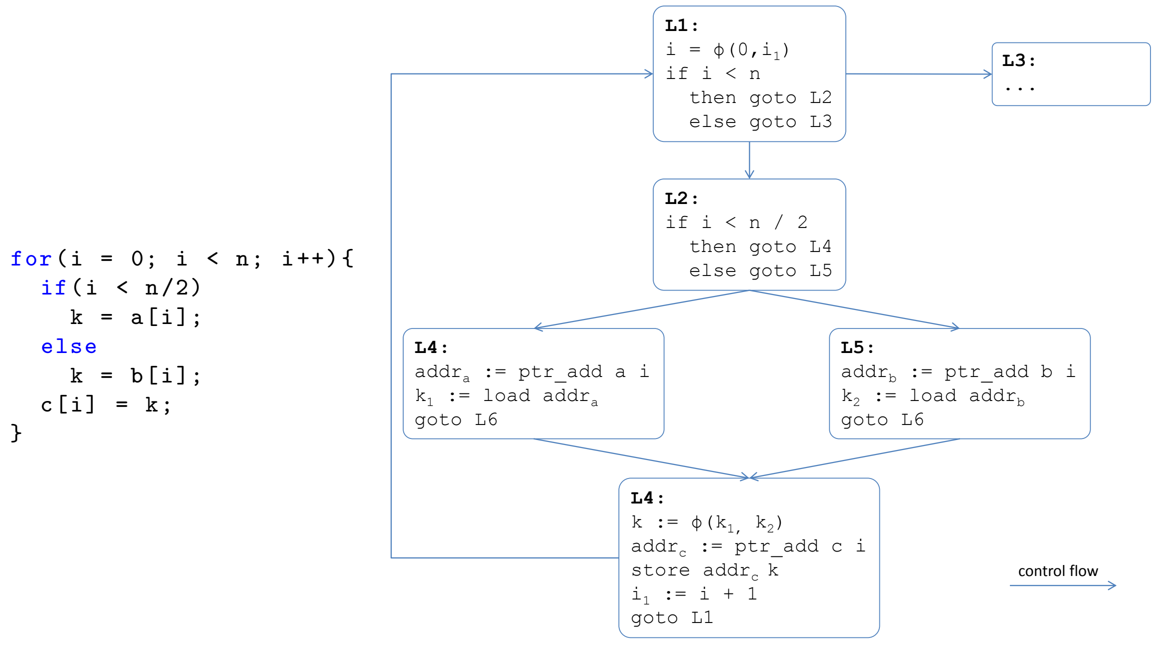
variable	::=	*string*
fun_name	::=	*string*
literal	::=	valid C literals
label	::=	*string*
type	::=	valid C type
op	::=	*arith_op* $\mid$ *bit_op* $\mid$ *cmp_op*
arith_op	::=	`+` $\mid$ `-` $\mid$ `*` $\mid$ `/` $\mid$ `%`
bit_op	::=	`&` $\mid$ `|` $\mid$ `^` $\mid$ $\ll$ $\mid$ $\gg$
cmp_op	::=	`==` $\mid$ `!=` $\mid$ `>` $\mid$ `>=` $\mid$ `<` $\mid$ `<=`
expression	::=	*literal*
	$\mid$	*variable*
	$\mid$	*expression op expression*
	$\mid$	*condition* `?` *expression* `:` *expression*
	$\mid$	`cast` *type expression*
	$\mid$	`load` *address*
	$\mid$	`ptr_add` *address expression*
	$\mid$	`field_sel` *expression expression*
	$\mid$	`phi` $args_{\phi}$
	$\mid$	*call_statement*
call_statement	::=	`call` fun_{id} $args_{fun}$
fun_{id}	::=	fun_{ptr} $\mid$ *fun_name*
fun_{ptr}	::=	*expression*
$args_{fun}$	::=	*expression**
$args_{\phi}$	::=	*variable+*
address	::=	*expression*
condition	::=	*expression*
statement	::=	*variable* `:=` *expression*
	$\mid$	`store` *address expression*
	$\mid$	*call_statement*
control_flow	::=	`return` *expression*?
	$\mid$	*cond_jump*
	$\mid$	*uncond_jump*
cond_jump	::=	`if` *condition* `then` $uncond_{jump}$ `else` $uncond_{jump}$
uncond_jump	::=	`goto` *label*
basic_block	::=	*label* `:` *statement** *control_flow*
function	::=	*type fun_name*`(`*fun_arg**`)` {*basic_block**}
fun_arg	::=	*type* `:` *variable*
program	::=	*function**

Table 8.2: Intermediate representation language syntax

Structure field selection is done with the `field_sel` instruction. Its first argument is an expression representing the structure, and the second argument when evaluated to an integer i, denotes the ith field. Phi expressions (`phi`) represent the special SSA assignments that are introduced at control flow merges. We note that we will write ϕ instead of `phi`. Function calls can be made either with the function name or with a pointer to that function.

The control flow is achieved through three special statements. The return statement (`return`) exits the current function and returns the specified value, if any. The conditional jump (`if` *cond* `then goto` L_1 `else goto` L_2) changes the control to another basic block according to the specified condition. The unconditional jump is represented with the `goto` statement.

We note that using the proposed syntax one may build non-valid programs of programs that are not in SSA form. It is the job of the compiler that will create valid intermediate representations. For the rest of the thesis we consider that we only work with valid programs in SSA form. Figure 8.3 illustrates the transformation from source code to intermediate representation of a simple program. The figure highlights the basic blocks and makes the control flow explicit.



```
for(i = 0; i < n; i++){
  if(i < n/2)
    k = a[i];
  else
    k = b[i];
  c[i] = k;
}
```

```
L1:
i = ϕ(0,i1)
if i < n
  then goto L2
  else goto L3
```

```
L3:
...
```

```
L2:
if i < n / 2
  then goto L4
  else goto L5
```

```
L4:
addra := ptr_add a i
k1 := load addra
goto L6
```

```
L5:
addrb := ptr_add b i
k2 := load addrb
goto L6
```

```
L4:
k := ϕ(k1, k2)
addrc := ptr_add c i
store addrc k
i1 := i + 1
goto L1
```

Figure 8.3: Transformation form source code to intermediate representation

8.3 Symbolic range analysis

Symbolic range analysis computes the minimum and maximum values for variables at each program point. The values are expressed in terms of a given set of input parameters, hence the *symbolic* qualifier.

The information is computed through an abstract interpretation procedure. That is, the algorithm executes the program by following its control flow paths, updating the current ranges to reflect the side effects of the statements encountered along these paths, until a fixed point is reached. As we have seen in Section 2.2.1 the key components of data flow algorithms are the *lattice* structure and the *transfer* *functions*. Section 8.3.1 motivates our choice for a particular lattice while Section 8.3.2 discuss discuss the overall algorithm and presents a selection of transfer functions.

We note that our approach was inspired by Blume et al.'s symbolic range propagation [57]. The similarities and differences between us and them are highlighted during the related work discussion (Section 10). Other inspiring resources were Bae's inter-procedural range propagation [3], Yong's pointer analysis [61], Verbrugge's generalized constant propagation [54] and Stephenson's bitwise analysis [51].

8.3.1 Lattice

There are many choices one can make for the structure which holds information about the values that a variable can have at run time. The difficulty of choosing the right one comes from the fact that we have to balance precision and computational costs. Moreover the choice has to be made at two different levels. The first level is choosing the right structure to hold the necessary data for a single program variable. Examples of such structures suitable for our needs are: a set of values, a single interval or an interval expression. The second level is about redefining the semantic to specify all executions "in parallel" and account for the entire program state. In other words, after we choose a structure to hold data for a given variable we have to lift it to a structure that can relate all variables data across all executions. This is usually refereed to as the *collecting semantics* [35]. An example for this is recording sets of execution histories in a relational or independent fashion.

Note that no matter what option is selected, for data flow analysis purposes the structure should form a lattice with ascending chain condition [35].

Choosing the right lattice

The most precise way to store the values that a variable can have is to keep all its possible values in a set. In this case, if the variables domain is $\mathcal{D}$ the corresponding lattice is $(2^{\mathcal{D}}, \subseteq)$. Unfortunately this is not an option in practice - the time and space complexity would simply be too high. Usually $\mathcal{D}$ is infinite or has a very large cardinality and thus it poses convergence problems. Using a widening operator [32, 35] does not help either since precision will be too low given the set representation.

An alternative is to represents the values as interval expressions. That is, instead of storing the data at the granularity of values, e.g., $v \mapsto \{1, 3, 20, 21, 23\}$, to allow some imprecision in the form of intervals: $v \mapsto [1, 3] \cup [20, 23]$. For this case, a reasonable complexity can be achieved if we use a widening operator that limits the number of intervals from expressions. This can be achieved by merging subintervals when the total number exceeds some constant. We can thus argue that the approach is a viable option in practice. However, it is appropriate only for concrete numeric intervals. Extending the approach to support arbitrary algebraic expressions as interval bounds is not easy. The reason is that for symbolic intervals we cannot easily distribute $\cap$ over $\cup$ and get an expression only in $\cup$. Expanding the representation to expressions with $\cup$ and $\cap$ is also not an option. The complexity associated with the symbolic computations needed to perform operations on such a form renders this approach impracticable. Though, being able to manipulate symbolic intervals is exactly what we need.

We can manage the extra complexity by applying one more restriction to the above representation. That is, the expression should contain only one interval at any time. In other words, we trade again precision for computation complexity. The main downside of this approach is that we cannot account for "holes" - a value set of $\{1, 2, 1000\}$ would be represented as $[1, 1000]$ instead of $[1, 2] \cup [1000, 1000]$. Figure 8.4 showcases the lattice for the case of plain numeric intervals.

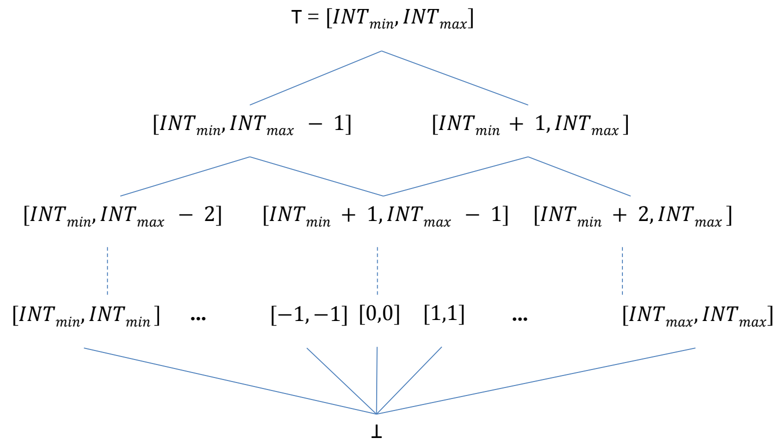


Figure 8.4: Classic lattice for integers intervals

The above lattice creates the starting point for the lattice that we actually use. As already mentioned we need to support intervals with symbolic bounds. Given a set of variables $\mathcal{V} = \{a, b, \dots\}$ we can extend the lattice to accommodate our symbolic needs as presented in Figure 8.5. The lattice elements are *symbolic ranges* whose bounds can be arbitrary symbolic expression that range over the variables in $\mathcal{V}$ (as defined in Section 7).

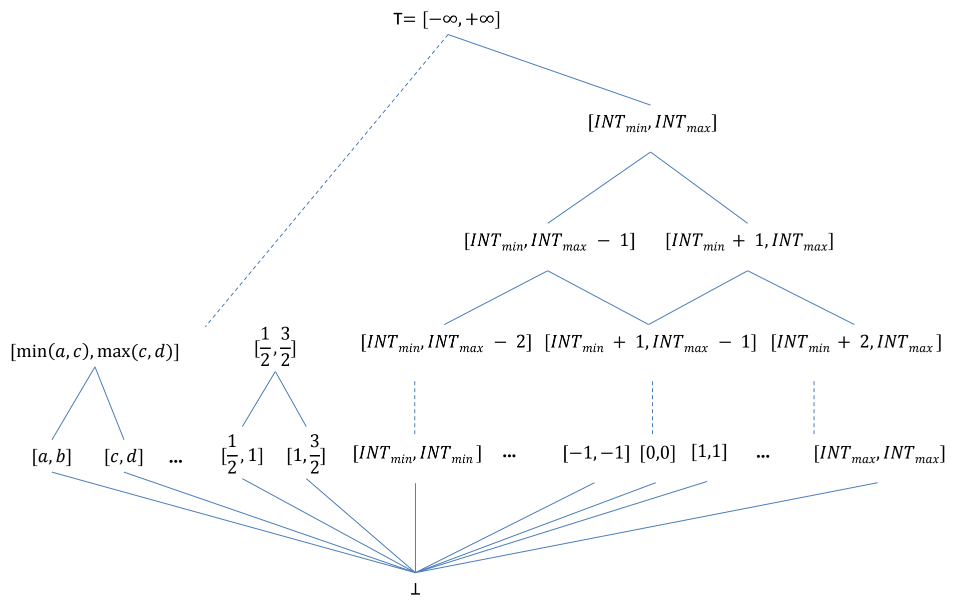


Figure 8.5: Lattice for symbolic intervals in $\mathbb{Q}$ domain

More formally, the lattice can be defined as $(\mathbf{SymRange}, \sqsubseteq)$ where $\mathbf{SymRange} = \{\bot\} \cup \{[a, b] \mid a \leq b, a$ and b symbolic expression over $\mathcal{V}\}$ and $\sqsubseteq = \subseteq$, the natural interval inclusion. We use $\bot$ to denote the uninitialized range. The join operator $\sqcup$ is the natural interval union.

It is important to mention that the "symbolic" part of the lattice does not have a proper statically defined semantic. In particular, there might be execution paths where $[a, b] \sqcup [c, d] = [a, d]$ (if $a \leq c$ and $d \geq b$) and paths where for the same variables we have $[a, b] \sqcup [c, d] = [c, b]$ (if $c \leq a$ and $b \geq d$). It also important to mention that the domain of the extended lattice is $\mathbb{Q}$ instead of $\mathbb{N}$.

In what concerns the collecting semantics we consider the following lattice for the program state:

$$\mathbf{RangeLattice} = \mathbf{Lab} \rightarrow \mathbf{Var} \rightarrow \mathbf{SymInterval}$$

where **Lab** is the set of program points, **Var** is the set of program variable that we track and **SymInterval** is the above defined interval lattice. This is the lattice that we ultimately use for the data flow algorithm. We can read it as: for each program point maintain a mapping from variable to their values interval.

8.3.2 Algorithm

In order to compute the ranges we employ a standard work-list algorithm. In what follows we highlight the general idea of the algorithm; the reader is invited to consult [35] for a precise specification. The algorithm iterates through the program flow edges in a predefined order, updating the information of each program point until a fixed point is reached. A flow edge is essential a tuple of two program points: (source, destination). At each step the algorithm processes an edge and applies a *transfer function* that will propagate information from the source program point to the corresponding destination. The transfer functions are defined based on the statement or expression that defines their corresponding program points. During the execution, the algorithm maintains a work list of edges that still need to be processed. Whenever the algorithm changes the value of the destination program point of an edge it adds its successors to the work list. The algorithm finishes when the work list becomes empty, i.e., the information of all program points stabilize.

The general algorithm is instantiated to compute ranges by using the lattice **RangeLattice** as the container for information about the program. Basically, for each program point we store a dictionary from variables to their values interval. We use the following initial ranges as the algorithm starting point. Program variable that correspond to inbound parameters (Section 4.2) and loop invariants (Section 8.6) have as initial range the symbolic expressions denoted by the variable itself. For all the other variables the initial range is $\perp$, the undefined range. This acts like a neutral element for interval union and intersection: $r \cup \perp = r \cap \perp = r$.

The transfer functions are as follows. At each control flow merge point we combine the ranges we got so far using the interval union operator as the join operator. The operator is applied point-wise to each variable in the dictionary. At control flow divergence points we add new constraints for the intervals using the interval intersection operator. For data flow program points we compute a new interval by performing interval arithmetic on the incoming operands. Table 8.3 presents the basic operators that are used to compute the ranges for control flow points. Table 8.4 presents the arithmetic operators used for data flow program points. Table 8.7 presents informally the most important transfer functions.

union	$[a, b] \cup [c, d]$	$=$	$[min(a, c), max(b, d)]$
intersection	$[a, b] \cap [c, d]$	$=$	$[max(a, c), min(b, d)]$
widening	$[a, b] \nabla [c, d]$	$=$	$[\textbf{if } a = c \textbf{ then } a \textbf{ else } -\infty,$ $\textbf{if } b = d \textbf{ then } b \textbf{ else } +\infty]$
narrowing	$[a, b] \triangle [c, d]$	$=$	$[\textbf{if } a \neq -\infty \textbf{ then } a \textbf{ else } c,$ $\textbf{if } b \neq +\infty \textbf{ then } b \textbf{ else } d]$

Table 8.3: Basic operations on ranges

$$[a,b] \oplus [c,d] \quad = \quad [a+c, c+d]$$

$$[a,b] \ominus [c,d] \quad = \quad [a-d, b-c]$$

$$[a,b] \otimes [c,d] \quad = \quad \begin{cases} [a*c, c*d] & \text{if } a \geq 0 \text{ and } c \geq 0 \\ [c*c, a*d] & \text{if } a \geq 0 \text{ and } d < 0 \\ [a*d, c*c] & \text{if } c < 0 \text{ and } c \geq 0 \\ [b*d, a*c] & \text{if } c < 0 \text{ and } c < 0 \\ [a*c, b*c] & \text{if } c \geq 0 \text{ and } c = d \\ [a*c, b*c] & \text{if } c < 0 \text{ and } c = d \\ [-\infty, +\infty] & \text{otherwise} \end{cases}$$

$$[a,b] \oslash [c,d] \quad = \quad \begin{cases} [a,b] \otimes [\frac{1}{d}, \frac{1}{c}] & \text{if } c \geq 0 \text{ or } d \leq 0 \\ [a,b] \otimes [\frac{1}{c}, \frac{1}{d}] & \text{if } c \leq 0 \text{ and } d \geq 0 \\ [-\infty, +\infty] & \text{otherwise} \end{cases}$$

$$[a,b] \, \texttt{mod}_r \, [c,d] \quad = \quad [c,d]$$

Note: At lower (upper) bounds computation, $+, -, *, /$ denotes symbolic expression operators (Section 7). If the operators fail for a given bound then the value is replace by the $-\infty$ $(+\infty)$. Moreover, although some comparison may seem redundant they are not - because we work with symbolic expressions we might have $a > b$, $a < b$ or $a\ ?\ b$.

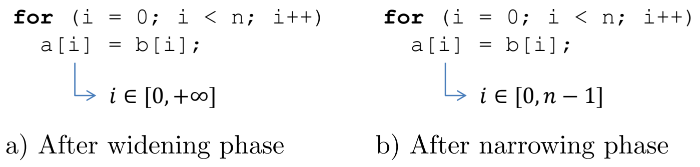
Table 8.4: Arithmetic operators on intervals

The attentive reader might have notice that the simple fix point algorithm described above might never converge given the lattice presented in the previous section. The reason is that the lattice **SymInterval** has infinite chains which is ultimately propagated to the actual lattice **RangeLattice**. A simple example for this is the chain $[0,0] \sqsubseteq [0,1] \sqsubseteq \cdots \sqsubseteq [0, MAX_{int}]$ when $MAX_{int} = +\infty$ (i.e., for integers with infinite precision). Note that even if we consider integers with finite precision, in practice MAX_{int} is too large to ever consider computing the above chain.

We solve this problem by using a widening operator and a narrowing operator at selective program points. Their definition is given in Table 8.3.

The widening operator is actually an overly conservative join operator. It replaces the join operator at loop headers when the incoming dictionaries define the same variables. This causes the algorithm to visit the loop twice before it applies the widening operator. We note that although the intermediate representation does not have explicit loops, the existing infrastructure of vfEmbedded provide analyses to reconstruct this information. The narrowing operator is used to recover some of the information that was lost when the widening operator was applied. Our definition for widening and narrowing operators was inspired by the operators defined by Blume et al. [57].

The actual algorithm proceeds in two phases: a widening phase, followed by a narrowing phase. During widening we compute an upper approximation of the least fix point. As discussed earlier the widening operator replaces the join operator at loop headers. During narrowing we try to regain some precision that we lost in the widening phase. The phase starts with the results of the previous phase. The join operator is replaced by the narrowing operator. Figure 8.6 illustrate the effects of each phase.



```
for (i = 0; i < n; i++)
  a[i] = b[i];
```

```
for (i = 0; i < n; i++)
  a[i] = b[i];
```

Figure 8.6: Algorithm phases

a)	$\begin{tikzcd} l_2: b \arrow[dr] & & l_3: c \arrow[dl] \\ & l_1: a = b\ \texttt{op}\ c \arrow[d] & \\ & {} & \end{tikzcd}$	$a^{l_1} = b^{l_2}\ \texttt{op}_r\ c^{l_3}$, where $\texttt{op} \in \{\texttt{+},\texttt{-},\texttt{*},\texttt{/},\texttt{\%}\}$, $\texttt{op}_r \in \{\oplus, \ominus, \otimes, \oslash, \texttt{mod}_r\}$
b)	$\begin{tikzcd} l_2: b \arrow[dr] & & l_3: c \arrow[dl] \\ & l_1: a = b \mathbin{\hat{}} c \arrow[d] & \\ & {} & \end{tikzcd}$	$a^{l_1} = \begin{cases} [0,0] \ominus b^{l_2} \ominus [1,1] & \text{if } c == -1 \\ [0,0] \ominus c^{l_2} \ominus [1,1] & \text{if } b == -1 \\ \top & \text{otherwise} \end{cases}$ ($b \mathbin{\hat{}} c$ denotes bitwise xor)
c)	$\begin{tikzcd} l_2: b \arrow[dr] & & l_3: c \arrow[dl] \\ & l_1: a = cond\ \texttt{?}\ b\ \texttt{:}\ c \arrow[d] & \\ & {} & \end{tikzcd}$	$a^{l_1} = b^{l_2} \cup c^{l_3}$
d)	$\begin{tikzcd} {} \arrow[d] \\ l_1: a = \texttt{load}\ addr \arrow[d] \\ {} \end{tikzcd}$	$a^{l_1} = \begin{cases} \texttt{sym\_var}_{range}(addr) & \text{if } addr \text{ is a loop invariant} \\ \top & \text{otherwise} \end{cases}$
e)	$\begin{tikzcd} l_2: p \arrow[dr] & & l_3: i \arrow[dl] \\ & l_1: q = \texttt{ptr\_add}\ p\ i \arrow[d] & \\ & {} & \end{tikzcd}$	$q^{l_1} = p^{l_2} + \texttt{size\_of}(p) * i^{l_3}$
f)	$\begin{tikzcd} {} \arrow[d] \\ l_1: a = \texttt{field\_sel}\ struct\ addr \arrow[d] \\ {} \end{tikzcd}$	$a^{l_1} = \begin{cases} \texttt{sym\_var}_{range}(struct, addr) & \text{if } addr \text{ is a loop invariant} \\ \top & \text{otherwise} \end{cases}$
g)	$\begin{tikzcd} l_2: b \arrow[dr] & & l_3: c \arrow[dl] \\ & l_1: a = \phi(b, c) \arrow[d] & \\ & {} & \end{tikzcd}$	$a^{l_1} = b^{l_2} \cup c^{l_3}$
h)	$\begin{tikzcd} & l_1: \texttt{if}\ a < b & \\ & l_2: \texttt{then goto}\ L_2 & \\ & l_3: \texttt{else goto}\ L_3 \arrow[dl] \arrow[dr] & \\ L_2: \ldots & & L_3: \ldots \end{tikzcd}$	$a^{l_2} = a^{l_1} \cap [-\infty, b_{lo} - 1]$ where $[b_{lo}, b_{hi}] = b^{l_1}$; $a^{l_3} = a^{l_1} \cap [b_{hi}, +\infty]$

Figure 8.7: Selected transfer functions for range analysis

Transfer function explanations. The left part of Table 8.7 illustrates the program flow and the right part presents the corresponding transfer function. Each statement and operation is prefixed by its program point label, e.g., $l_1:\ a = b + c$. The arrows suggests the flow direction. The transfer functions are expressed as equations in terms of exit values. That is, $a^{l_1} = b^{l_2}\ \texttt{op}_r\ c^{l_3}$ is read as: the interval for a at l_1 exit point is equal to the interval of b at l_2 exit point $\texttt{op}_r$ the interval for c at l_3 exit point.

a) performs classic interval arithmetic.

b) illustrates one of the special patterns that we recognize for bitwise operations. The compiler will replace normal operations with bitwise operations whenever possible; thus we often encounter such patterns.

c) handles the ternary conditional operator. Because we do not have an explicit control flow we are not able to impose constraints based on *cond*.

d) addresses the problem of loading a value from memory. If the address is proven to be loop invariant we replace the result by a singleton interval. $\texttt{sym\_var}_{range}$ generates a fresh symbolic variable that is used as lower and upper bound for the interval. It does this based on the *addr* such that subsequent loads from the same invariant address will receive the same symbolic variable. This emphasises the importance of having loop invariant analysis. Without it, we would be forced to always return top.

e) performs pointer addition. For each pointer p, `size_of` returns the size in bytes of the elements pointed to. Thus, the resulted interval will be the interval of accessed bytes. By using number of bytes and not number of elements we are able to have a fine control over pointer casting. For example, declaring a pointer of type `int* p` and then casting it to `char* q = (char*)p` will make each individual byte addressable. That is, if `p++` advances with four bytes, `q++` advances with only one byte. If we loop over such operations 100 times, the accessed size will be 400 bytes for `p` versus 100 bytes for `q`, even if `q` is just an alias for `p`. Thus, by counting bytes we are more precise about the actual accessed memory.

f) selects a field structure. The transfer is similar to the one for loading operations. If we can prove that the selection is loop invariant then we introduce a new singleton interval with a fresh variable. Otherwise we return top.

g) represents the transfer function for control flow merge points. The result will be the union of all incoming values.

h) handles control divergence points. Note that an `if` statement receives three different program points: for the condition, for the then branch and for the else branch. Based on which branch is taken a different constraint is imposed.

Table 8.7 presents just a subset of all transfer functions. For the rest of the statements ans expressions the functions are defined in a similar way.

Increasing precision. We can increase the analysis precision if we take better advantage of the constraints we generate at divergence points. We can do this by using the following observation. A variable is uniquely defined by its definition list. Thus, we can detect equivalent program variables by looking at equivalent ϕ nodes. For example if we have $l_1 : a_1 = \phi(b, c)$ and $l_2 : a_2 = \phi(b, c)$ we can infer that a_1 and a_2 represent the same variable even if they are defined at different program points, l_1 and respectively l_2. In this context if we add a constraint on a_1 we know that the constraint will also hold for a_2 (given that they are under the same constraint scope). Thus it would be beneficial from a precision point of view to propagate any constraint for a_1 to a_2. We do this by storing the definition of the variable on which constraints were added. If we later encounter another variable with the same definition we apply the corresponding constraints.

Besides this, it is important to note that the analysis precision can greatly benefit if certain transformations are applied before it. Two such transformations are induction variables substitution and ternary condition expansion. Induction variables substitution replaces computations from a loop body that are independent of the induction variable with computations that depend on it. Figure 8.8 illustrate such a transformation.

```
int k = 10;                          int k = 10;
for(int i = 0; i < n ; i++){         int k1 = k;
  ... = a[k] + b[i];           ⟹    for(int i = 0; i < n ; i++){
  k++;                                 ... = a[k1] + b[i];
}                                      k1 = k + i;
                                     }
```

Figure 8.8: Induction variable substitution transformation

The transformation is important because it enables us to add constraints on variables and thus we can be more precise. In the example above we are able to infer that `k`$_1 \in [10, n]$. In contrast, without transforming the code, we cannot impose restriction on `k`, and we will be force to return `k`$\in [10, +\infty]$.

Ternary condition expansion replaces the ternary condition operators with `if then else` control flow statements. This serves the same goal as the induction variable substitution. It allow us to impose more constraints on the intervals and be more precise.

8.3.3 Example

Figure 8.9 presents the result of the symbolic range analysis for a sample program. For readability reasons we use source code instead of intermediate representation. The example is artificially constructed to showcase most of the analysis features.

```
int* in  = // ...
int* out = // ...
int m = // ...
int sum = // ...
for(int i = 0; i < n; i++){
  char * aux = (char *)in ;
  int k = i * i;                  ⟶ k ∈ [0, n^2]
  out[i] = aux[k];                ⟶ aux ∈ [in, in + n^2], out ∈ [out, out + 4 * n^2]
  if (i < m) {
    // use i                      ⟶ i ∈ [0, m − 1]
  } else {
    int j = i*out[0];             ⟶ j ∈ [m * α, n^2 * α], α = out[0] = loop invariant
    int l = j * aux[i];           ⟶ l ∈ ⊤, aux[i] = loop variant
    // ...
  }
  int mux = m > 10 ? n : m;       ⟶ mux ∈ [min(m, n), max(m, n)]
  int w = mux + i;                ⟶ w ∈ [min(m, n), max(m + n, 2 * n)]
  // ...
}
```

Figure 8.9: Symbolic range analysis example

8.4 Expression reconstruction analysis

Expression reconstruction analysis computes for each program point the set of possible expressions that variables can have. The expressions are maximally expanded up to recursive definitions, predefined input variables, induction variables and unknown values loaded from memory. Figure 8.10 clarifies the goals of expression reconstruction through a simple example. For readability reasons we present the source code instead of the intermediate representation.

```
1  sum = 0;                              ⟶ sum = {0}
2  for(int i = 0; i < n; i++){           ⟶ i = {i}
3    int row_idx = i*n                   ⟶ row_idx = {i ∗ n}
4    int* row = img + row_idx;           ⟶ row = {img + i ∗ n}
5    for(int j = 0; j < m; j++){         ⟶ j = {j}
6      int val = row[j];                 ⟶ row = {α}, &α = {img + i ∗ n + j}
7      sum = sum + val;                  ⟶ sum = {0, sum + α}
8    }
9  }
```

Figure 8.10: Expression reconstruction example

For the first line it is easy to see that sum can only be 0. For line 2 we know that i is an induction variable and as mentioned its possible expressions are not expanded (similar for line 5 and j). The information that i and j represent induction variables is provided by existing vfEmbeded analyses (recall that in the intermediate representation we do not have explicit loops). Nevertheless, as we will explain later, we can also detect induction variables only using the reconstruction analysis. On line 3 it is again easy to see that row_idx can only be equal to $i * n$. The value for row at line 4 is obtained by expanding the value of row_idx. Line 6 features a memory load. Since its impossible to derive what value is located at the memory address, a new variable (α) is introduced. val becomes equal to α and a new expression is computed for the memory offset from where α originates ($\&\alpha$). At line 7 we have a recursive definition: sum is defined in terms of itself. When this happens, only the first recursive step is expanded. Taking into account that sum can also be 0 (from line 1) we get $sum = \{0, sum + \alpha\}$.

The attentive reader will have noticed that the goal of expression reconstruction analysis is similar to the one of available expression analysis [35, 1]. Available expression analysis determines for each program point, the set of expressions that must already have been computed, and not later modified, on all paths to the program point. The difference is that the reconstruction analysis computes the set of all possible expressions for all program variables (not only the ones that do not need to be recomputed). This leads to a *may* analysis, whereas the available expression analysis is a *must* analysis [35]. Moreover, as described in the rest of this section, the reconstruction of possible expressions is done in terms of other program variables and according to predefined rules.

As in the case of symbolic range analysis we specify the reconstruction analysis as a monotone framework. Also, a similar work list algorithm is used to compute the sets of possible expressions. The lattice in this case is **SymExp** $= (2^{\mathbf{Exp}}, \subseteq)$. **Exp** is the set of all possible symbolic expressions that can be constructed using the normal form described in Section 7. The join operator is the set union. A key difference from the range analysis is that for reconstruction analysis the flow-sensitiveness is encoded directly in the intermediate representation. The SSA form restricts variables to just one definition. This means that we cannot redefine a variable and assign a new expression to it. Multiple expressions can only be assigned to a variable through ϕ functions and ternary condition operators. This enables us to optimize the working lattice. That is, instead of maintaining for each program point a mapping from variables to their set of possible expressions we can store just one global mapping for the entire program. Thus our final lattice is:

$$\textbf{ExpLattice} = \textbf{Var} \rightarrow \textbf{SymExp}$$

Another difference with the range lattice is that although the lattice does not have ascending chain condition, in practice the chains are always finite and actually not very long. Thus, we do not need a widening operator, and a simple fix point iteration will rapidly converge. To understand why this happens consider the possible sources of infinite or very long chains. These are recursive definitions and loop iterations. But expressions are always expanded up to the first recursive definition and up to induction variables. Thus we will have a recursive definition chain of at most length two. Because of this a set of possible expression will not grow more than the longest argument list for a ϕ function.

The initial set of possible expressions of variables are as follows. For predefined variables (e.g., kernel parameters) and for loop invariants the only possible expressions is given by the symbolic expression defined by the variable itself. For all the other variables the initial set is empty.

The most important transfer functions are presented in Table 8.5. They closely follow the transfer functions defined for range analysis with the difference that the operations are performed on expressions rather than on intervals. Also, because of the SSA form, no constraints need to be applied at control flow points.

a)	$\begin{tikzcd} l_2: b \arrow[dr] & & l_3: c \arrow[dl] \\ & l_1: a = b \texttt{ op } c \arrow[d] & \\ & {} & \end{tikzcd}$	$a^{l_1} = \{e_b \texttt{ op } e_c \mid e_b \in b^{l_2},\ e_c \in c^{l_3}\}$, where $\texttt{op} \in \{\texttt{+},\texttt{-},\texttt{*},\texttt{/}\}$
b)	$\begin{tikzcd} l_2: b \arrow[dr] & & l_3: c \arrow[dl] \\ & l_1: a = b\ \hat{}\ c \arrow[d] & \\ & {} & \end{tikzcd}$	$a^{l_1} = \begin{cases} \{0 - e_b - 1 \mid e_b \in b^{l_2}\} & \text{if } c == -1 \\ \{0 - e_c - 1 \mid e_c \in c^{l_3}\} & \text{if } b == -1 \\ \{\texttt{sym\_var}_{exp}(b, c)\} & \text{otherwise} \end{cases}$ ($b\ \hat{}\ c$ denotes bitwise xor)
c)	$\begin{tikzcd} l_2: b \arrow[dr] & & l_3: c \arrow[dl] \\ & l_1: a = cond\ ?\ b\ :\ c \arrow[d] & \\ & {} & \end{tikzcd}$	$a^{l_1} = b^{l_2} \cup c^{l_3}$
d)	$\begin{tikzcd} {} \arrow[d] \\ l_1: a = \texttt{load}\ addr \arrow[d] \\ {} \end{tikzcd}$	$a^{l_1} = \{\texttt{sym\_var}_{exp}(addr)\}$
e)	$\begin{tikzcd} l_2: p \arrow[dr] & & l_3: i \arrow[dl] \\ & l_1: q = \texttt{ptr\_add}\ p\ i \arrow[d] & \\ & {} & \end{tikzcd}$	$q^{l_1} = \{e_p + \texttt{size\_of}(p) * e_i \mid e_p \in p^{l_2}, e_i \in i^{l_3}\}$
f)	$\begin{tikzcd} {} \arrow[d] \\ l_1: a = \texttt{field\_sel}\ struct\ addr \arrow[d] \\ {} \end{tikzcd}$	$a^{l_1} = \{\texttt{sym\_var}_{exp}(struct, addr)\}$
g)	$\begin{tikzcd} l_2: b \arrow[dr] & & l_3: c \arrow[dl] \\ & l_1: a = \phi(b, c) \arrow[d] & \\ & {} & \end{tikzcd}$	$a^{l_1} = b^{l_2} \cup c^{l_3}$

Table 8.5: Selected transfer functions for reconstruction analysis

It is important to mention that whenever we encounter an operation that is not supported by the symbolic computation engine and that cannot be translated to a supported one, we introduce a fresh variable that acts as a symbolic constant. This contrasts with range analysis where we would go to top. This is also the reason why the reconstruction analysis can never fail with top. The fresh symbolic constant is created using $\texttt{sym\_var}_{exp}$ function. This is in sync with $\texttt{sym\_var}_{range}$ - for the same expression they will generate the same symbolic constant.

a) performs arithmetic operations. The result is obtained by combining all possible expressions of the operands with the specified operator.

b) illustrates the translation from bitwise operations to arithmetic operations.

c) and g) illustrate the only cases that introduce multiple expressions. These are ternary operators and ϕ functions.

d) and f) always introduce new variables. The reason is that we cannot determine what will be loaded from memory and we cannot look inside structure fields.

e) presents pointer arithmetic. Similar to the range case, the expression is computed in bytes rather than in the number of elements. The motivation remains the same.

Note that the transfer functions presented in Table 8.5 do not account for recursive definitions. Because this is a recurring pattern that arise in multiple cases (e.g., a),b) and e)) we choose to explain it separately. In order to detect recursion we associate to each expression a list of program points that contributed to its creation. For example, in the case illustrated at point a), the program points that contribute to the possible expressions of a are l_1, l_2 and all the other points that contributed to b and c. Whenever we combine options from different sets of possible expression we first check if one of the expression was defined based on the current program point. If this is the case, then it means we detected a recursive pattern and we replace the expression with the recursive variable.

Using the analysis to detect induction variables

Currently, induction variables are detected using the existing infrastructure of vfEmbedded. We can also detect them using reconstruction analysis. Basically, every variable with a recursive definition is a potential induction variable. We can detect the real ones by checking if the given variable is involved in the testing condition for the loop exit. The downside of this approach is that in order to get the results that we need (expressions expanded up to induction variable) we will need to run the analysis twice. The first run will detect induction variables. The second run will compute the expressions we need using the detected induction variables as symbolic constants.

8.5 Sign analysis

Sign analysis computes at each program point the set of signs that variables can have. The analysis is vital for the precision of symbolic range analysis because it provides the necessary information to compare two symbolic expressions in the general case. The analysis is expressed as a monotone framework. It computes the variable signs with the help of same work list algorithm used in previous analyses.

The sign lattice is $\mathbf{Signs} = (2^{\mathbf{SignSet}}, \subseteq)$, where $\mathbf{SignSet} = \{\texttt{-}, \texttt{0}, \texttt{+}\}$. Figure 8.11 illustrates the lattice in graphical terms. The semantics is the natural one: `+` denotes a positive variable, `-` a negative one and `0` means that the variable is equal to 0. The working lattice follows the same pattern as the one from range analysis. It associates to each program point a mapping from variables to their possible signs:

$$\textbf{SignLattice} = \textbf{Lab} \rightarrow \textbf{Var} \rightarrow \textbf{Signs}$$

It is easy to see that the lattice has only finite chains and thus it satisfies the ascending chain condition. As a consequence we do not need to employ the extra widening operator.

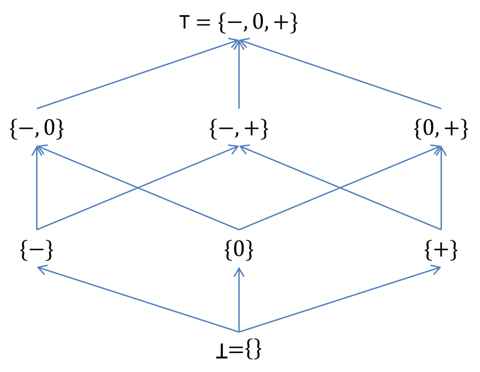


Figure 8.11: Sign lattice

Variables start with the undefined sign, represented by the empty set. The transfer functions are defined in Table 8.6. They proceed in a similar way with the transfer functions for expression reconstruction analysis. The difference is that all arithmetic computations are performed at sign level. The sign computation are subscripted with s (e.g., $+_s$) and are define in the natural way: i.e., $\texttt{+} \; *_s \; \texttt{-} = \{\texttt{-}\}$, $\texttt{+} \; +_s \; \texttt{-} = \{\texttt{-}, \texttt{0}, \texttt{+}\}$ etc.

a)	$\begin{tikzcd} l_2:\ b \arrow[dr] & & l_3:\ c \arrow[dl] \\ & l_1:\ a = b \texttt{ op } c \arrow[d] & \\ & {} & \end{tikzcd}$	$a^{l_1} = \bigcup\{s_b \ \texttt{op}_s \ s_c \ \mid \ s_b \in b^{l_2},\ s_c \in c^{l_3}\}$ where $\texttt{op} \in \{\texttt{+},\texttt{-},\texttt{*},\texttt{/}\}$, $\texttt{op}_s \in \{\texttt{+}_s,\texttt{-}_s,\texttt{*}_s,\texttt{/}_s\}$
b)	$\begin{tikzcd} l_2:\ b \arrow[dr] & & l_3:\ c \arrow[dl] \\ & l_1:\ a = cond \texttt{ ? } b \texttt{ : } c \arrow[d] & \\ & {} & \end{tikzcd}$	$a^{l_1} = b^{l_2} \cup c^{l_3}$
c)	$\begin{tikzcd} {} \arrow[d] \\ l_1:\ a = \texttt{load}\ addr \arrow[d] \\ {} \end{tikzcd}$	$a^{l_1} = \{\texttt{-}, \texttt{0}, \texttt{+}\}$
d)	$\begin{tikzcd} {} \arrow[d] \\ l_1:\ q = \texttt{ptr\_add}\ p\ i \arrow[d] \\ {} \end{tikzcd}$	$q^{l_1} = \{\texttt{+}\}$
e)	$\begin{tikzcd} {} \arrow[d] \\ l_1:\ a = \texttt{field\_sel}\ struct\ addr \arrow[d] \\ {} \end{tikzcd}$	$a^{l_1} = \{\texttt{+}\}$
f)	$\begin{tikzcd} l_2:\ b \arrow[dr] & & l_3:\ c \arrow[dl] \\ & l_1:\ a = \phi(b, c) \arrow[d] & \\ & {} & \end{tikzcd}$	$a^{l_1} = b^{l_2} \cup c^{l_3}$

g)	l_1: `if` $a < b$ l_2: `then goto` L_2 l_3: `else goto` L_3 $\swarrow \quad \searrow$ L_2: ... $\quad L_3$: ...	$a^{l_2} = a^{l_1} \cap (\{\texttt{-}\} \cup b^{l_1})$ $a^{l_3} = a^{l_1} \cap (\{\texttt{+}\} \cup b^{l_1})$

Table 8.6: Selected transfer functions for sign analysis

Increasing precision In order to increase the precision of variables signs we extend the basic analysis to detect three complex sign patterns. The necessity of discovering the patterns is increased by the fact that the compiler tends to transform the code in a way that artificially creates these patterns.

The first pattern tries to restrict more variables at control divergence points by looking at the definition of the comparison variable. The following code illustrates this:

```
a = 3*b;
if (a > 0) {
  //use b       ⟶ a is positive and so should b be
} else {
  //use b       ⟶ a is negative or zero and so should b be
}
```

We achieve this by inspecting the possible expressions for the comparison variable (as returned by the expression reconstruction analysis). If all the expressions depend on only one variable (called the definition variable) we move forward. Based on them we create sign equations which are solved one by one (see Section 7). The results are merged using the union operator. If by doing this we discover a more precise sign for the definition variable we update its sign.

The second pattern tries to detect relations between variables from the comparison expression. The example below illustrates the relations we aim to discover.

```
if (a + b > c){
  //use a             ⟶ if c - b > 0 then a should be positive
} else {
  //use a             ⟶ if c - b > 0 then a should be negative or zero
}
```

We recognize this pattern by using again the possible expressions of the comparison variables. Based on the expressions structure we create sign equations. If by solving them we get a more precise sign we update the variables accordingly.

The third and last pattern tries to increase precision for the sign of the expressions that result after the application of the ternary operator. Take for example the following statement $l_1 : a = b > 0 \ ?\ b \ :\ 2*b$. The possible expressions for a are $\{b, 2*b\}$. It is obvious from the statement that in this context the expression b is positive and the expression $2*b$ is negative or 0. Treating the operator as a compact expression makes detecting this case impossible. To this end, we simulate a conditional divergence control flow point by introducing new program points for the value expressions, i.e., $l_1 : a = b > 0 \ ?\ (l_1^1 : b) \ :\ (l_1^2 : 2*b)$. The newly created program points are integrated in the program flow in the natural way. In this way we are able to obtain the precision we need. Detecting this pattern is important, because it allows us to further compare the symbolic expressions associated with the definition of a.

8.6 Loop invariant analysis

Loop invariant analysis computes the set of variables and expressions that have the same value for all loop iterations. In other words, it detects the computations that are loop invariant and could be moved outside the loop body. The analysis is particularly helpful when it comes to memory operations. Under normal circumstances previous analyses will have to provide a conservative result for values that are loaded from memory. But if we can prove that such a value will be the same for all loop iterations then we can replace it by a symbolic constant during the computations (recall the range analysis example from Section 8.3.3). This becomes even more important when we consider how the compiler handles accesses to structure fields. In the intermediate representation the access to a structure field is translated to the special memory operation `field_sel` that accepts arbitrary operands. Proving that the operands are constant with respect to the loop body enables us to use the structure fields as symbolic constants in further computations. The following is valid code for the intermediate representation that the compiler might generate after optimisations:

```
field_1 := 0
a := field_sel str field_1     ⟶ field_1 is loop invariant, thus a is loop invariant
//...
field_2 := φ(0,1)
b := field_sel str field_2     ⟶ because field_2 is not loop invariant, neither is b
```

The analysis is performed for each loop of interest and follows the same pattern as the previous analyses. The lattice has only three elements $\mathbf{Inv} = (\{Uninitialized, Invariant, Variant\}, \sqsubseteq)$; the order relation is depicted in Figure 8.12.

$$\begin{tikzcd}[row sep=small]
\top = Variant \\
Invariant \arrow[u] \\
\bot = Uninitialized \arrow[u]
\end{tikzcd}$$

Figure 8.12: Lattice for loop invariant analyses

As the expression reconstruction analysis, the loop invariant analysis benefit from the SSA form of the intermediate representation. That is, because we are interested in the assignment statements, and a variable can only be assigned once, the flow sensitiveness is encoded directly in the representation. Thus, our working lattice maintains a global mapping from variables to invariant information.

$$\mathbf{LoopInvLattice} = \mathbf{Var} \rightarrow \mathbf{Inv}$$

The initial values for variables are set as follows. All loop inbound variables are considered $Invariant$s (refer to Section 4.2 for explanations about how these are detected). The rest of the variable are set to $Uninitialized$. The literals are implicitly considered $Invariant$s. All transfer functions are defined based on the same idea. An expression is loop invariant if all its operands are loop invariants. When an expression is evaluated the invariant information of its operands is combined with the following commutative and associative operator:

$$\begin{array}{llllll} Uninitialized & \&_{inv} & Invariant & = & Uninitialized \\ Uninitialized & \&_{inv} & Variant & = & Uninitialized \\ Variant & \&_{inv} & Invariant & = & Variant \\ Variant & \&_{inv} & Variant & = & Variant \\ Invariant & \&_{inv} & Invariant & = & Invariant \end{array}$$

Because of the simplicity we present only the transfer function for arithmetic operations.

$$\begin{tikzcd}[column sep=small] l_2:\ b \arrow[dr] & & l_3:\ c \arrow[dl] \\ & l_1:\ a = b \ \mathtt{op}\ c \arrow[d] & \\ & {} & \end{tikzcd} \qquad a^{l_1} = b^{l_2}\ \&_{inv}\ c^{l_3}, \text{ where } \mathtt{op} \in \{\mathtt{+,-,*,/,\%}\}$$

8.7 Adding context sensitiveness - ongoing work

The analysis framework described so far is context-insensitive. This means that when functions are analysed we do not take into account their call context. Considering the function context will give us more precise results but at the same time will incur heavy penalties on the analysis complexity [41, 35]. In general, before extending the analysis to a full inter-procedural context sensitive analysis, one has to carefully analyse the trade off between the need for extra precision and the added complexity. In this section we motivate why making the analysis context sensitive makes sense for us and how we actually do it. We note that the at the time of writing the report this is ongoing work. That is, the underlying framework and algorithms have been fully extended to support context sensitiveness, the sign analysis was already been updated with contexts, but the work on the other analyses is in progress.

8.7.1 Motivation

With context sensitive analyses we will be able to be more precise in the optimisation phase. The identification phase still needs global information about the kernel and thus does not benefit from contextual information. In contrast, coalescing and shared memory analysis benefit from such information and can generate more precise reports. We will explain this with the help of the following example:

```
int getIndex(int x) {
  return 2*x + 4;
}

void f() {
  int* a = //...
  int* b = //...
  for (int i = 0; i < n; i++) {   // to be mapped to GPU
    //...
    ... = a[getIndex(i)];
    //...
    ... = b[getIndex(i*n)];
    //...
    for (int k = 0; k < 10; k++) {
        ... = c[getIndex(k)];
16        }
17    }
18 }
```

It easy to see that the access to `a` will have good coalescing while the access to `b` will suffer from bad coalescing. However, this information can be computed only if we take into account the different contexts from which `getIndex` is called. Without doing this, the information about `x` will be merged across all calls and the coalescing of operations that use `getIndex` will be reported as being poor. This influences the quality of the mapping advice and the precision of future performance estimation.

In what concerns shared memory we have to focus on the memory operation from line 14. The access to `c` does not depend on the grid coordinates and thus the entire accessed section will be shared across all threads. However, we cannot detect this precise result because of the previous calls to `getIndex` which poisoned its possible return values. Thus, being context sensitive can make a big difference in terms of how much need for shared memory is reported.

It is also important to mention that in our context adding context sensitiveness does not incur big performance penalty. The reason is that we do not run the analysis on the entire program. We do it only for the program cone determined by the body of the loop that needs to be parallelized. This enable us to discard the issues associated with the complexity of context sensitive analyses.

8.7.2 Technique

We extend the analyses with context sensitive features based on the techniques for inter-procedural analysis presented by Nielson et al. in [35]. In this section we present only a short summary of the technique. For extensive details we direct the reader to [35].

Lifting an intra-procedural analysis to an inter-procedural one implies extending its monotone framework to an embellished monotone framework. Basically, this is done by:

- defining new transfer functions for calls, functions entry and exists and return statements
- lifting the existing transfer functions to include context by lifting the working lattices
- assuring that each function call and returns implies a context change

We add context by extending the working lattice such that the information stored for each program point depends on the context. For example the sign lattice is transformed from

$$\mathbf{SignLattice} = \mathbf{Lab} \rightarrow \mathbf{Var} \rightarrow \mathbf{Sings}$$

to

$$\mathbf{SignLattice}_{sensitive} = \mathbf{Lab} \rightarrow \mathbf{Context} \rightarrow \mathbf{Var} \rightarrow \mathbf{Sings}$$

We represent the context as call strings. Since a call is uniquely identified by its program point a call string is defined by a sequence of program points. Thus, we can write $\mathbf{Context} = \mathbf{Lab}^{\star}$. The context is bounded to an arbitrary length that can be set before the analysis is run.

Function entries and exists are artificial program points introduced for convenience. They allow us to have a single entry and a single exit point for each function and thus simplify the transfer functions. The transfer function for a call passes the information from actual parameters to formal parameters. At the same time it makes sure that the context is changed such that the computed information is associated with the right context. The transfer function for a return propagates information from inside the functions and from before the call back to the caller context. It also changes the context in order to avoid poisoning of the caller scope.

An alternative solution that can solve the problems related to context sensitiveness is function inlining. That is, replace each call to a non-recursive function with the actual function body. This eliminates the need for maintaining context information. Thus, it would enable us to reuse the same infrastructure we developed so far. Although it seems like an easy solution, inlining comes with several drawbacks. First of all, it cannot handle recursive functions. Since new GPU generations support recursive functions [36], this will limit the applicability of our system. Second, inlining increases the code size of the caller function. This will impact cache performance and may cause memory trashing. Because of this, inlining is recommended only for small functions. Third, it increases the number of registers that are required by the caller function. This overhead translates to a higher register pressure which affects the GPU occupancy [38]. Overall, even if beneficial for analyses function inlining may degrade performance if it is used careless. Thus, for best results, function inlining must be complemented by an explicit context-sensitive analysis.

9 Sample report

In this section we present a sample report as output by the system we designed. The goal is to provide a complete view of the data that is available after a given piece of code was analysed. We note that presenting the example in terms of the actual intermediate representation is not feasible. The representation is too verbose to fit in a reasonable space and not readable enough. Because of this we will stay at the source code level. Unfortunately, this implies that the report will be a bit off from the real one. The reason for this are the additional inbound variables that the compiler introduces during its own optimisation phase (refer to Section 4.2 for more details). They cannot be presented without going to the lower level of intermediate representation. Even so, the delta between the below presentation and the real report is minimal.

To maintain familiarity with the examples presented during the thesis we will present the full report for the 2D convolution explained in Section 5.2. The analysed code is presented below.

```c
// structure delcarations omitted for brevity
for (y = 0; y < image->height; y += 1) {
  for (x = 0; x < image->width * 3; x += 3) {
    unsigned char *out_row = out_image->pixmap[y * image->width * 3];
    int fx, fy;
    int red,grn,blu;
    for (fy = 0; fy < filter->height; fy += 1) {
      int py = y + fy - (filter->height / 2);
      unsigned char *in_row = in_image->pixmap[py * image->width * 3];
      for (fx = 0; fx < filter->width; fx += 1) {
        int px = x + 3*(fx - filter->width / 2);
        int coeff = filter->pixmap[fx + fy * filter->width];
        int offlimits = px < 0 || px >= image->width * 3 || py < 0 || py
            >= image->height;
        red += offlimits ? 0 : in_row[px + 0] * coeff;
        grn += offlimits ? 0 : in_row[px + 1] * coeff;
        blu += offlimits ? 0 : in_row[px + 2] * coeff;
      }
    }
    // clampping skipped for brevity
    out_row[x + 0] = red * filter_gain;
    out_row[x + 1] = grn * filter_gain;
    out_row[x + 2] = blu * filter_gain;
  }
}
```

Note that for brevity we skipped the structures definition and some of the irrelevant code. The analysis report is presented in the table below.

Kernel parameters

Inbound	*Nested inbound*
`image`	`image->pixmap` , accessed section = $[-\frac{3}{2} * f_h * i_w - \frac{3}{2} * f_w, -4 + \frac{3}{2} * f_h * i_w + \frac{3}{2} * f_w + 3 * i_w * i_h]$
`filter`	`filter->pixmap` , accessed section = $[0, f_w * f_h - 1]$

Outbound	*Nested outbound*
`out_image`	`out_image->pixmap` , accessed section = $[0, 3 * i_w * i_h - 1]$

Coalescing report

Line	*Memory operation*	*Alias for*	*Coalescing factor*	*Optimisation opportunity*
14:	`in_row[px + 0]`			
15:	`in_row[px + 1]`	`image->pixel + ...` [1]	$\frac{3}{4} * i_w$	Reverse loops order to get a coalescing factor of 1.
16:	`in_row[px + 2]`			
12:	`filter->pixmap[...]` [2]	none	1	none
20:	`out_row[px + 0]`			
21:	`out_row[px + 1]`	`out_image->pixel + ...` [3]	$\frac{3}{4} * i_w$	Reverse loops order to get a coalescing factor of 1.
22:	`out_row[px + 2]`			

Shared memory opportunities

Memory operations	`filter->pixmap[...]` [2]
Shared memory requirement	$[0, f_w * f_h - 1]$
Overlapping area size	$f_w * f_h - 1$, between both: x- and y-axis adjacent threads
Remarks	All threads will share the same memory area.

Memory operations	`in_row[px + 0]` , `in_row[px + 1]` , `in_row[px + 2]`
Shared memory requirement	2D requirement: x-axis : $[-\frac{3}{2} * f_w, 3 * BlockDim.x + \frac{3}{2} * f_w - 1]$ y-axis : $[-\frac{1}{2} * f_h, BlockDim.y + \frac{1}{2} * f_h]$
Overlapping area size	adjacent threads on x-axis : $f_w * f_h - f_w$ adjacent threads on y-axis : $f_w * f_h - f_h$
Remarks	Shared memory requirements were reduced by 2D reconstruction of linearised access and grouping of similar accesses by variable.

Table 9.1: Analysis report for 2D convolution example

[1]`in_row[px + 0]` $= -\frac{3}{2} * f_h * i_w - \frac{3}{2} * f_w + 3 * fx + 3 * fy * f_w + 3 * Grid.y + 3 * Grid.x * f_w$. Similar for the others. In the current coordinate mapping *Grid.x* corresponds to `y` induction variable and *Grid.y* to `x` induction variable.

[2]`filter->pixmap[fx + fy*filter->width]`

[3]`out_row[px + 0]` $= 3 * Grid.y + 3 * Grid.x * i_w$]. Similar for the others.

Some remarks must be made about the above report. First of all, in order to be concise we used aliases for variable names. That is, we use f_w, f_h to denote `filter->width` and `filter->height`, and i_w, i_h to denote `image->width` and `image->height`. Second, the reader might have noticed that when reporting the accessed section for the input image, `image->pixmap` we report negative offsets. The reason for this is that the out of bounds computations from line 13 uses complex logical expressions that cannot be analysed with the current infrastructure. The consequence is that we report more than needed. The extra reported space amounts to the padding of image with filter area and is added before and after the actual array. The third remark is about the factor 3 which appears in most of the expressions. One might think that this is an error, since for example it is natural for the size of the output to be equal to the original image size. This is caused by the mismatch between the actual pixmap size and the image size. The image size is expressed in pixels while the pixmap size is express in pixel colours (3 per pixel).

10
Related work

Automatic program analysis and transformation for program parallelization have been studied before and implemented in various compilers and tools. But until recently most of this work has been done in the context of multi-core CPUs. Only lately did the research community start to enter the GPU arena and propose specific solutions for this environment. In what follows we review the most important work that has been done in the GPU area and compare it with ours.

Leung et al. [27] propose an extension of the Java JIT compiler that executes suitable code on the GPU instead of the CPU. It employs both static and dynamic features to decide whether it is feasible and beneficial to off-load a piece of code to the GPU. Their GPU parallelization algorithm considers only simple affine loops and focuses on identifying loops without dependencies; optimisations or complex transformation schemes are not considered. The implemented extension makes use of the RapidMind framework [40] as a back-end for the generated code. Compared with them, we also look into GPU specific optimisations. Moreover, they require that every array index expression must be either loop-invariant or an affine expression in terms of a single induction variable. We do not have such constraints. They can also handle only numeric bounds whereas our approach is fully symbolic.

Kwiatkowski et al. [25] briefly describe a hardware independent tool that the authors claim can automatically parallelize sequential programs depending on the number of available processing units. The general methodology identifies three main stages for program parallelization: dependency analysis, program transformation and code generation. Still, there are no details about their actual strategies used in implementation and no benchmarking results. We also consider that they omitted the optimisation stage, which as we have seen is a critical step for GPU devices.

Baskaran et al. present in [5] what is probably the most complete approach to automatic GPU parallelization. They describe an automatic source-to-source transformation framework that can take an arbitrarily nested affine C program and generate an efficient CUDA program. In contrast with other mentioned approaches, they also apply various optimisations during the program transformation. Their focus is on memory optimisations as they exploit global memory coalescing, shared and constant memory usage and registers allocation. From a program analysis and transformation point of view they make use of the polyhedral model [8] and rely on two existing framework CLooG [10] and Pluto [9]. CLooG is a powerful open-source state-of-the-art code generator that transforms a polyhedral representation of a program with affine scheduling constraints into concrete loop code. Pluto is a source-to-source optimiser for end-to-end automatic parallelization and locality optimisation of affine programs. The framework is based on the authors prior work [7, 6] that developed some of the compiler optimisations. Among the limitations of the presented framework we mention: support only for affine loops, no control flow optimisations and no strategy to handle loops that have carry dependencies or which include calls to functions that cannot run on GPU. Compared with them, we approach GPU parallelization from a different angle.

They model the problem as a polyhedral framework whereas we model it as a data flow analysis problem with symbolic computations. We consider that although the polyhedral approach is powerful and efficient, it is too limited in the class of programs that it can handle. In particular the programs must have statically know control flow and employ only affine accesses to memory. We are not restricted by such conditions. Our symbolic framework can handle arbitrary symbolic expressions, not necessary affine or polynomial. Nevertheless, their approach has a better algorithmic complexity since they do not use fix point algorithms. Another key difference is that they perform the analysis on the source code whereas we use an intermediate representation.

Notable work in this area is also done by Ryoo et al. [48, 47, 49], Lee et al. [26], and Liu et al. [30].

Ryoo et al. [47, 49] present several experimental studies of how various programs perform on CUDA based GPUs. In [48] the authors propose performance metrics such as efficiency and utilisation to prune the optimisation search space on a pareto-optimality basis. All their work is performed by hand and does not involve a compiler or an analysis framework; they manually perform the optimisations and manually generate performance models for each test program. Even so, the reported results are insightful and very useful to us. They provide a starting point to reason about which optimisations are best suited to automation and after what program feature we should look in order to be able to generate them. Our choice for which optimisation to investigate in Section 5 is also based on their work.

Lee et al. [26] develop a compiler framework for automatic translation of standard OpenMP shared-memory programs to CUDA programs. The proposed translator converts the loop-level parallelism from the OpenMP programming model into data parallelism for the CUDA programming model. Their contribution consists of several translation strategies (e.g., kernel region identifying) and transformation techniques like matrix transpose and loop collapsing. However the system does not optimise access to global memory and also does not make use of on-chip shared memory. Also, as mention above, they target only existing OpenMP programs and do not consider general sequential programs. In contrast with them, our system does not need user annotation and is able to handle arbitrary programs. Also, as already mentioned, we do not stop after the initial kernel identification. We continue and search for more specific GPU optimisations.

Liu et al. [30] study the influence of program input on the optimisation of CUDA programs. They develop a compiler-based adaptive framework, G-ADAPT, which is able to extract optimisation space from program code and automatically search for the near-optimal configuration of parameters that affect its performance. The framework takes unoptimised CUDA code as input and traverse an optimisation space, searching for optimal parameters to transform the input into an optimised CUDA code. As mention, they only work with existing CUDA programs which they also require to be annotated with G-ADAPT-pragmas. Their work has a scope different from ours, but nevertheless it serves as a source of inspiration for the optimisation phase.

As a general observation, all the above approaches consider only the automatic transformation part of GPU parallelization. We make one more step, and allow the generated kernel models to be instantiated with run time values. This enables future accurate performance estimations. Moreover, we designed a "failover" mechanism which allow us to extract information about the program even when the analysis framework fails to provide relevant data. For this, we combine static and dynamic information about the program.

In what concerns the static analysis framework, the most notable analysis of our system is symbolic range analysis. Thus, we will only detail related work with respect to it. Range analysis has been addressed in several contexts over the last decades where it was used to eliminate arrays bound checks, detect arrays bound violation, compute bitwise information and detect memory dependencies. Our main use of range analysis is to determine array access sections.

Blume et al. [57, 58] propose a symbolic range analysis that can handle ranges with non affine bounds. They implement the analysis in the Polaris compiler [59]. But although they report good results, they mention that the algorithm convergence is slow. Our analysis was influenced by their approach but uses a different representation for the symbolic intervals. The algorithm we employ for propagating ranges is also slightly different. In particular, we start only with a predefined set of symbolic variables that will be used to express all the other ranges. Their approach considers every program variable as a possible symbolic variable. This creates the need for complex replacement policies when ranges are computed, from where the slow convergence of their algorithm. For our goals, using all program variables as the set of variables for symbolic expressions will only add overhead and no essential information. We also use more powerful symbolic computations backed by a complex sign analysis. For example, they compare expressions by computing the difference of their intervals and inspecting the resulting interval. Because this uses full interval computations, it is much more expensive than our approach.

Stephenson et al. [51] use range analysis to minimise the number of bits used to represent each operand for both integers and pointers in a program. They do not handle symbolic ranges, and are restricted to statically known numeric values. Despite this, they use a novel approach with respect to data flow propagation which increases the analysis precision. In particular they use a forward analysis followed by a backward analysis. The analyses resembles the application of widening and narrowing operators but differ in a key aspect. The backward analysis basically uses information about the statically allocated arrays in order to detect when an index is out of bounds. This information is then propagated backwards to reduce the original data range.

Rugina et al. [43] approach the problem with a novel perspective. They do not rely on classic abstract interpretation techniques. Instead, they formulate the analysis problem as a system of inequality constraints between symbolic bound polynomials. They reduce the constraint system to a linear program which is solved under the assumption that the sign of each coefficient is known. This basically eliminates the need for fix point iteration algorithms which is an excellent feature when compared with other approaches. They also show how the approach can be extended to inter-procedural analysis.

Other notable approaches to range analysis, symbolic or just numeric, are those of Yong et al. [61], Eigenmann et al. [3], Verbrugge et al. [54] and Padua et al. [52].

11 Conclusions

This thesis tackled the problem of GPU parallelization for existing sequential programs. We have shown three main things. First, why mapping programs to GPUs is important - because GPUs have tremendous power when it comes to data parallel processing. Second, why it is difficult to achieve a good GPU mapping - because GPGPU platforms are not yet very mature and require a lot of inside knowledge about the GPU architecture. Third and most important, how we can (semi)automate the mapping process - by automatically extracting information about the program that enables guided and automatic transformations.

We have designed and developed an automatic system which helps users get the most out of their data parallel kernels. The system takes as input an existing sequential program (written in ANSI C99) and is able to answer the following three main questions related to GPU mapping. Is this program part suitable to be mapped to a GPU? If yes, how it should be transformed in order to run on the GPU? Are there any GPU specific optimisation opportunities? The system is built on a complex automatic analysis framework that is able to provide enough information about the program to achieve guided or automatic GPU mapping. Guided means that developers can use the system to get information and advice about how they should port their program to run on a GPU. Automatic means that the information provided by the system can be used as input to an automatic parallelization engine that automatically transforms the code. We integrated the system with an existing production quality compiler and analysis tool, vfEmbedded [53]. It also worth mentioning that the system was developed using the Caml language [16].

From a top level perspective the system does the following. It identifies GPU friendly loops by applying a feasibility test. It extracts the information needed to realize a straightforward GPU mapping of the loop (i.e., inbound and outbound parameters and their accessed size). It further detects GPU specific optimisations opportunities; more specifically it looks for coalescing opportunities and shared memory optimisations. It assembles all the information obtained so far into an algebraic kernel model. The model can further be used for automatic parallelization. It further combines the model with actual data values as they occur during program execution to produce a detailed kernel report. This can be used to estimate the mapping performance. And last, it tries to recover information that was lost due to analysis imprecision by using data extracted during the program execution. In the context of vfEmbedded, we use the kernel model to produce recipes that guide the user in the transformation process.

Under the hood, the system relies on automatic program analyses. To this end we have designed a static analysis framework that complements the dynamic analyses of the existing system. The static analysis framework is based on symbolic computations and aims to compute algebraic information about the program. It consists of four intra-procedural, flow-sensitive analyses: symbolic range analysis, expression reconstruction analysis, sign analysis and loop invariant analysis. Work in progress aims at lifting the analyses to context-sensitive inter-procedural.

The techniques we proposed and developed, although targeting the CUDA platform and ANSI C99 programs, are largely independent of the actual GPU platform and the language that is analysed. Up to present day, most of the GPUs that are suitable for general purpose computing are separate devices with their own memory. Because of this, the identification part which deal with recognizing the kernel is architecture agnostic. Also, every GPU has special on chip memories (the equivalent of shared memory) and employ similar techniques such as coalescing to speed up the access to the slower global memory. Because of this, the memory optimisations we investigated apply outside the CUDA scope.

Future work

This thesis is the first step towards an automatic parallelization framework for GPUs. In this context, there are multiple directions for future work that can improve or extend the current work.

The next obvious step is to perform the actual transformation. For this, the necessary information is already present in the system. Because our analyses are performed on an intermediate representation of the program, reconstructing the original source code is hard and imprecise. Thus, the transformations will ultimately aim to modify the intermediate representation or to directly generate CUDA binary.

Another research direction is to investigate more about coordination transformations. That is, how the loop space is mapped to the grid space. With respect to this, the current system is limited to permutations of the original loop space. More complex mappings can help at improving data locality for both coalescing and shared memory.

A critical research direction is to extend the class of programs that can be handled. For example, the current system discards any loop with carried dependencies. Nevertheless, there are plenty of loops whose dependencies are caused just by a small fraction of instructions. If we detect this fraction, then we could extract it into a different loop. Basically we would get two loops: one without carried dependencies and one with. We could then map the first one to the GPU and keep the second one on the CPU. The communication between loops can be done by vectorizing the data that was present in the original loop. If the first loop performs heavy computations then we would definitely achieve better performance. This amounts to detecting map-reduce patterns [15] where the heavy computation is mapped to the GPU, and the reduction is performed on the CPU.

Another research direction relates to discovering more optimisation opportunities. For example detecting branch divergence is particularly important because when it happens, both branches are executed sequentially. One approach to this is to extend the symbolic capabilities of the system to handle logical symbolic expressions. Based on these, we could detect what input variables cause the divergence. The next step would be to search for alternative coordinate mappings for which divergence does not occur. Another possible optimisation relates to multiple kernel launches. Consider for example two different kernels that are executed one after the other and that use the same input data. In this case we can keep the data on the GPU and not copy it again for the second kernel. The current system can be easily extended to support such optimisations.

Future work can also improve the quality of the reported optimisations. For example, in order to reduce the need for shared memory the system currently reconstructs linearised 2D accesses. A next step is to improve on this and recognize arbitrary nD accesses.

One other important research area is to extend the system to consider specific features of other GPU architectures (e.g., AMD). Although the presented techniques are general and can easily by applied to other architectures there are also fundamental differences that require a different approach (e.g., AMD uses a VLIW architecture while NVIDIA uses a SIMT one).

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison Wesley, us ed edition, January 1986.

[2] Amd. *Accelerated Parallel Processing Programing Guide*, 2011.

[3] Hansang Bae and Rudolf Eigenmann. Interprocedural Symbolic Range Propagation for Optimizing Compilers. In Eduard Ayguadé, Gerald Baumgartner, J. Ramanujam, and P. Sadayappan, editors, *Languages and Compilers for Parallel Computing*, volume 4339 of *Lecture Notes in Computer Science*, chapter 28, pages 413–424. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2006.

[4] Gogul Balakrishnan and Thomas Reps. Analyzing Memory Accesses in x86 Executables. In Evelyn Duesterwald, editor, *Compiler Construction*, volume 2985 of *Lecture Notes in Computer Science*, chapter 2, pages 2732–2733. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2004.

[5] Muthu Baskaran, J. Ramanujam, and P. Sadayappan. Automatic C-to-CUDA Code Generation for Affine Programs. In Rajiv Gupta, editor, *Compiler Construction*, volume 6011 of *Lecture Notes in Computer Science*, chapter 14, pages 244–263. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2010.

[6] Muthu M. Baskaran, Uday Bondhugula, Sriram Krishnamoorthy, Jagannathan Ramanujam, Atanas Rountev, and Ponnuswamy Sadayappan. A Compiler Framework for Optimization of Affine Loop Nests for GPGPUs. In *Proceedings of the 22nd annual international conference on Supercomputing*, ICS '08, pages 225–234, New York, NY, USA, 2008. ACM.

[7] Muthu M. Baskaran, Uday Bondhugula, Sriram Krishnamoorthy, Jagannathan Ramanujam, Atanas Rountev, and Ponnuswamy Sadayappan. Automatic Data Movement and Computation Mapping for Multi-Level Parallel Architectures with Explicitly Managed Memories. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 1–10, New York, NY, USA, 2008. ACM.

[8] Cedric Bastoul. Code Generation in the Polyhedral Model Is Easier Than You Think. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, PACT '04, pages 7–16, Washington, DC, USA, 2004. IEEE Computer Society.

[9] Uday Bondhugula, Albert Hartono, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 101–113, New York, NY, USA, 2008. ACM.

[10] CLoog. Code Generator in the Polyhedral Model. http://www.cloog.org/.

[11] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM.

[12] Béatrice Creusillet and François Irigoin. Interprocedural Array Region Analyses. In *Proceedings of the 8th International Workshop on Languages and Compilers for Parallel Computing*, LCPC '95, pages 46–60, London, UK, 1996. Springer-Verlag.

[13] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, October 1991.

[14] Brian A. Davey and Hilary A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 2 edition, May 2002.

[15] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, page 10, Berkeley, CA, USA, 2004. USENIX Association.

[16] Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The Objective-Caml system, release 3.12*. Institut National de Recherche en Informatique et en Automatique, June 2010.

[17] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon System for Dynamic Detection of Likely Invariants. *Sci. Comput. Program.*, 69:35–45, December 2007.

[18] Michael Garland, Scott Le Grand, John Nickolls, Joshua Anderson, Jim Hardwick, Scott Morton, Everett Phillips, Yao Zhang, and Vasily Volkov. Parallel Computing Experiences with CUDA. *IEEE Micro*, 28(4):13–27, July 2008.

[19] Michael P. Gerlek, Eric Stoltz, and Michael Wolfe. Beyond Induction Variables: Detecting and Classifying Sequences Using a Demand-driven SSA Form. *ACM Transactions on Programming Languages and Systems*, 17:85–122, 1995.

[20] Dominik Grewe and Michael O'Boyle. A Static Task Partitioning Approach for Heterogeneous Systems Using OpenCL. In Jens Knoop, editor, *Compiler Construction*, volume 6601 of *Lecture Notes in Computer Science*, chapter 16, pages 286–305. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2011.

[21] Khronos OpenCL Working Group. *The OpenCL Specification*, September 2010.

[22] Mary Hall, Jacqueline Chame, Chun Chen, Jaewook Shin, Gabe Rudy, and Malik Khan. Loop Transformation Recipes for Code Generation and Auto-Tuning. In Guang Gao, Lori Pollock, John Cavazos, and Xiaoming Li, editors, *Languages and Compilers for Parallel Computing*, volume 5898 of *Lecture Notes in Computer Science*, chapter 4, pages 50–64. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2010.

[23] Bingsheng He, Naga K. Govindaraju, Qiong Luo, and Burton Smith. Efficient Gather and Scatter operations on Graphics Processors. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, SC '07, pages 1–12, New York, NY, USA, 2007. ACM.

[24] Stefan Holdermans and Jurriaan Hage. On the Role of Minimal Typing Derivations in Type-Driven Program Transformation. In *Proceedings of the Tenth Workshop on Language Descriptions, Tools and Applications*, LDTA '10, New York, NY, USA, 2010. ACM.

[25] Jan Kwiatkowski and Radoslaw Iwaszyn. Automatic Program Parallelization for Multicore Processors. In *Proceedings of the 8th international conference on Parallel processing and applied mathematics: Part I*, PPAM'09, pages 236–245, Berlin, Heidelberg, 2010. Springer-Verlag.

[26] Seyong Lee, Seung J. Min, and Rudolf Eigenmann. OpenMP to GPGPU: A Compiler Framework for Automatic Translation and Optimization. *SIGPLAN Not.*, 44:101–110, February 2009.

[27] Alan Leung, Ondřej Lhoták, and Ghulam Lashari. Automatic Parallelization for Graphics Processing Units. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, PPPJ '09, pages 91–100, New York, NY, USA, 2009. ACM.

[28] Amy W. Lim. *Improving Parallelism and Data Locality with Affine Partitioning.* PhD thesis, 2001.

[29] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro*, 28(2):39–55, March 2008.

[30] Yixun Liu, Eddy Z. Zhang, and Xipeng Shen. A Cross-Input Adaptive Framework for GPU Program Optimizations. In *IPDPS '09 Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–10. IEEE, May 2009.

[31] Francesco Logozzo and Manuel Fahndrich. On the Relative Completeness of Bytecode Analysis Versus Source Code Analysis. In *Proceedings of the Joint European Conferences on Theory and Practice of Software 17th international conference on Compiler construction*, CC'08/ETAPS'08, pages 197–212, Berlin, Heidelberg, 2008. Springer-Verlag.

[32] Kim Marriott. Abstract Interpretation: A Theory of Approximate Computation. In Pascal Van Hentenryck, editor, *Static Analysis*, volume 1302 of *Lecture Notes in Computer Science*, chapter 28, pages 367–378. Springer Berlin / Heidelberg, Berlin/Heidelberg, 1997.

[33] Mario Méndez, Jorge Navas, and Manuel V. Hermenegildo. An Efficient, Parametric Fixpoint Algorithm for Analysis of Java Bytecode. In *In ETAPS Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE'07), Electronic*, 2007.

[34] Nicholas Nethercote and Julian Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM.

[35] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis.* Springer, corrected edition, December 2004.

[36] Nvidia. *Fermi Compute Architecture Whitepaper.*

[37] Nvidia. *CUDA C Best Practices Guide*, August 2010.

[38] Nvidia. *CUDA C Programming Guide*, September 2010.

[39] Jens Palsberg. Type-Based Analysis and Applications. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, PASTE '01, pages 20–27, New York, NY, USA, 2001. ACM.

[40] RapidMind. http://www.rapidmind.net/.

[41] Thomas Reps. On the Sequential Nature of Interprocedural Program-Analysis Problems. *Acta Informatica*, 33(5):739–757, August 1996.

[42] Xavier Rival. Abstract Interpretation-Based Certification of Assembly Code. In *Proceedings of the 4th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI 2003, pages 41–55, London, UK, UK, 2003. Springer-Verlag.

[43] Radu Rugina and Martin Rinard. Symbolic Bounds Analysis of Pointers, Array Indices, and Accessed Memory Regions. *SIGPLAN Not.*, 35:182–195, May 2000.

[44] Silvius Rus, Maikel Pennings, and Lawrence Rauchwerger. Sensitivity Analysis for Automatic Parallelization on Multi-Cores. In *Proceedings of the 21st annual international conference on Supercomputing*, ICS '07, pages 263–273, New York, NY, USA, 2007. ACM.

[45] Silvius Rus, Maikel Pennings, and Lawrence Rauchwerger. Implementation of Sensitivity Analysis for Automatic Parallelization. In José Amaral, editor, *Languages and Compilers for Parallel Computing*, volume 5335 of *Lecture Notes in Computer Science*, chapter 22, pages 316–330. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2008.

[46] Silvius Rus, Lawrence Rauchwerger, and Jay Hoeflinger. Hybrid Analysis: Static & Dynamic Memory Reference Analysis. *Int. J. Parallel Program.*, 31:251–283, August 2003.

[47] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen mei. Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 73–82, New York, NY, USA, 2008. ACM.

[48] Shane Ryoo, Christopher I. Rodrigues, Sam S. Stone, Sara S. Baghsorkhi, Sain Z. Ueng, John A. Stratton, and Wen mei. Program Optimization Space Pruning for a Multithreaded GPU. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '08, pages 195–204, New York, NY, USA, 2008. ACM.

[49] Shane Ryoo, Christopher I. Rodrigues, Sam S. Stone, Sara S. Baghsorkhi, Sain-zee Ueng, and Wen-mei W. Hwu. Program Optimization Study on a 128-Core GPU. In *In The First Workshop on General Purpose Processing on Graphics Processing Units*, 2007.

[50] Olin G. Shivers. *Control-flow Analysis of Higher-order Languages or Taming Lambda.* PhD thesis, Pittsburgh, PA, USA, 1991.

[51] Mark Stephenson, Jonathan Babb, and Saman Amarasinghe. Bidwidth Analysis with Application to Silicon Compilation. *SIGPLAN Not.*, 35(5):108–120, May 2000.

[52] Peng Tu and David Padua. Array Privatization for Shared and Distributed Memory Machines (extended abstract). *SIGPLAN Not.*, 28:64–67, January 1993.

[53] VectorFabrics. vfEmbedded. http://www.vectorfabrics.com/products/vfembedded.

[54] Clark Verbrugge, Phong Co, and Laurie J. Hendren. Generalized Constant Propagation: A Study in C. In *Proceedings of the 6th International Conference on Compiler Construction*, pages 74–90, London, UK, 1996. Springer-Verlag.

[55] Norman Wilde. Understanding Program Dependencies. Technical report, Software Engineering Institute, Carnegie Mellon, 1990.

[56] Blume William and Rudolf Eigenmann. The range test: a dependence test for symbolic, non-linear expressions. In *Proceedings of the 1994 ACM/IEEE conference on Supercomputing*, Supercomputing '94, pages 528–537, New York, NY, USA, 1994. ACM.

[57] Blume William and Rudolf Eigenmann. Symbolic Range Propagation. In *Proceedings of the 9th International Symposium on Parallel Processing*, IPPS '95, pages 357–363, Washington, DC, USA, 1995. IEEE Computer Society.

[58] Blume William and Rudolf Eigenmann. Demand-Driven, Symbolic Range Propagation. In *Proceedings of the 8th International Workshop on Languages and Compilers for Parallel Computing*, LCPC '95, pages 141–160, London, UK, 1996. Springer-Verlag.

[59] Blume William, Rudolf Eigenmann, Jay Hoeflinger, David Padua, Paul Petersen, Lawrence Rauchwerger, and Peng Tu. Automatic Detection of Parallelism: A Grand Challenge for High-Performance Computing. *IEEE Parallel Distrib. Technol.*, 2:37–47, September 1994.

[60] Michael Wolfe. *High-Performance Compilers for Parallel Computing.* Addison Wesley, facsimile edition, June 1995.

[61] Suan H. Yong and Susan Horwitz. Pointer-Range Analysis. In Roberto Giacobazzi, editor, *Static Analysis*, volume 3148 of *Lecture Notes in Computer Science*, chapter 12, pages 19–21. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2004.