A framework for data-access strategies in GPGPU programs

Mallens, L.F.C.C.

Award date:
2013

Technische Universiteit
Eindhoven
University of Technology

Department of Mathematics and Computer Science
Den Dolech 2, 5612 AZ Eindhoven
P.O. Box 513, 5600 MB Eindhoven
The Netherlands
http://w3.win.tue.nl/nl/

Series title:
Graduation report,
Embedded Systems

Commissioned by Professor:
prof. dr. H. Corporaal

Group / Chair:
Electronic Systems

Date of graduation symposium:
March 25, 2013

A framework for data-access strategies in GPGPU programs

by

Author: L.F.C.C. Mallens

Internal supervisor: prof. dr. H. Corporaal, Technische Universiteit Eindhoven

External supervisor: Drs. S. Holdermans, Vector Fabrics B.V.

Abstract

In recent years, graphics processing units (GPUs) became more and more popular as high performance processing units. Due to the availability of hundreds of cores, code fragments speed up significantly when they are transformed from CPU functions to GPU kernels.

The transformation process is non-trivial and therefore error prone. Developing correct and efficient GPU accelerated programs is time consuming.

One of the aspects that has to be considered is the logically and physically separation of a GPU's address space from the CPU. In order to enable kernel execution on the GPU, data transfers between the CPU and GPU need to be explicitly scheduled in the code. Since the CPU-GPU communication bandwidth is relatively low, compared to the computational power of the GPU, the number of transferred bytes should be minimised.

A GPU is equipped with an advanced hierarchy of scratchpad memories, that is different from CPU memory hierarchies. When these differences are not taken into during the development of the GPU kernel, the computational power of the GPU is not fully exploited. Memory access patterns need to be adapted to take full advantage of this memory hierarchy.

In order to decrease development time, there is an increasing interest in tools that can assist in the transformation process from CPU programs to GPU accelerated programs.

In this thesis, we propose a set of analysis tools and code optimisations to aid the transformation of CPU code fragments to GPU kernels. The tools focus on optimising the CPU-GPU communication strategy and optimising the kernels for the GPU's advanced memory hierarchy.

Contents

Chapter 1

Introduction

Nowadays, hardware vendors resort to the development of multi-processor platforms, to satisfy the ever increasing hunger for processing power. Multi-processor platforms have a better performance-to-watt ratio compared to single processor platforms [1]. Programs need to be adapted to these new platforms. Hereto they need to be divided in tasks that are assigned to threads. Threads are assigned to different processing units to be executed in parallel.

Graphical processor units (GPUs) become more and more popular as a multi-processor platform. GPUs evolved from devices that could only render three-dimensional graphics to *accelerators* that can execute a wide range of computational tasks. An accelerator is a device that executes tasks in parallel to a *host* (device containing the CPU). GPUs that can execute non-graphical tasks are called General Purpose GPUs (GPGPUs). GPGPUs have a large number of processing units that together offer high computational performance (hundreds of GFLOPS). Large speedups are achieved when CPU code fragments that expose a high level of data-parallelism are transformed into GPU *kernels*. GPU kernels are code-fragments that are executed on a GPU. Unfortunately, the transformation process is often non-trivial and far from easy. Therefore, transformation frameworks are developed that can aid programmers to generate GPGPU accelerated programs, i.e. code fragments are transformed into GPU kernels. This thesis presents a set of tools that fits in such a framework.

1.1 Challenges in GPGPU-programming

Programmers face a number of challenges in order to design correct and efficient GPGPU accelerated applications. Some of these challenges are listed in this section.

The majority of GPUs have a logically separated memory space from the CPU, that is explicitly managed by software. Therefore a programmer has to state explicitly which data-structures need to be transferred between the host and the GPU. In the case that a CPU program is transformed into a GPGPU accelerated program, generating a correct data transfer strategy can be a time consuming task. Indirect addressing, pointer modification and aliasing can make it challenging for a programmer to observe which data-structures need to be transferred.

When the programmer has designed a correctly behaving GPGPU accelerated program, the resulting performance may be disappointing due to the following two reasons.

First, the overhead in CPU-GPU data communication can be considerably large. The memory hierarchy of a GPU is typically physically separated from the CPU's memory hierarchy. Communication is performed over a low-bandwidth system bus. Therefore, the contribution of data transfers in the overall execution times can be significantly large. Hence, effort needs to be made to reduce this overhead. Figure 1.1 shows the effect of optimising data transfer strategies. In this example, a naive data-transfer strategy is shown, where required data is transferred to the GPU before the start of the kernel and all data-elements that are produced in the kernel are transferred back to the host after the kernel has ended. Two optimisations can be identified to reduce the overhead. The first one is the reduction of redundant copy actions by considering multiple kernels together. By applying this step, the copy-in actions for E and F are removed. These arrays are already available on the GPU since they were produced in previous kernels. Allocation and free actions have to be scheduled accordingly to guarantee correct behaviour. The second optimisation is to schedule copy actions asynchronously to the host and GPU. In this way they are overlapped with computations. By carefully scheduling all required copy and allocation actions throughout the code, the transfer and allocation overhead is minimised.

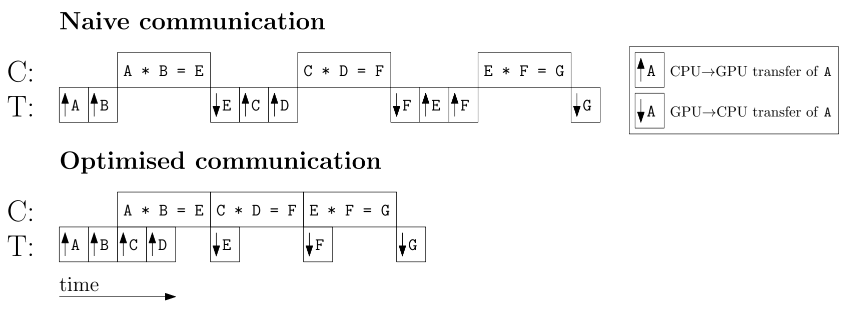


Figure 1.1: Computation (C) and data transfers (T) involved in 3 consecutive matrix multiplications executed on the GPU.

Secondly, a typical GPU has an advanced memory hierarchy of scratchpad memories and caches, each with different performance characteristics. Inefficient utilisation of the GPU's memory hierarchy significantly reduces overall performance. The basis of the GPU memory hierarchy is the GPU's off-chip memory. This explicitly managed memory is the largest memory available on the GPU and can be accessed by all threads on the GPU and by the host. However, accessing the off-chip memory is an expensive operation in terms of latency, therefore available bandwidth should be utilised most efficiently. In order to increase bandwidth utilisation, GPUs group memory operations of multiple threads into larger memory transactions. This technique of grouping is called *coalescing*. However, coalescing is not always possible.

Another way to reduce kernel execution time is to cache data on the on-chip scratchpad memory. The on-chip memory provides significantly better access latencies and communication bandwidths. Though, this memory has a small size (kilobytes) and can only be accessed by a subset of all threads throughout the kernel.

As can be concluded from previous issues, a large burden is placed on the programmer to design a correct and efficient GPGPU accelerated application. Development of GPGPU accelerated programs takes more time than the development of its CPU equivalent. Therefore, the need arises to develop tools that can aid programmers in solving these issues in order to decrease development time. The ultimate goal is to come to a framework that can automatically generate GPGPU accelerated programs from sequential CPU programs, without any involvement of the programmer.

1.2 Contributions

This thesis presents a set of analysis tools that can aid programmers or automatic transformation frameworks to transform a CPU program to a GPGPU accelerated program. The tools provide information to optimise the data-access strategies within GPU kernels and to generate efficient CPU-GPU data-transfer and GPU allocation strategies. The main goal is to optimise GPGPU accelerated programs for execution time.

The tools use the dynamic analysis framework as is provided by Vector Fabrics. Dynamic analysis uses execution traces of the application for input. Where a typical static analysis is unable to infer properties from applications that have a complex program-structure (containing pointer aliasing, pointer modification, indirect addressing, etc.), these properties can be inferred with dynamic analysis. However, the tools can easily be transformed to make use of static analysis.

The main contributions of this thesis are methods to:

1. generate efficient copy and allocation strategies in GPGPU accelerated programs (Chapter 4);
2. observe linear memory access patterns from program execution traces (Section 5.1);
3. map loops over the GPU's thread configuration, such that the level of memory coalescing is maximal (Section 5.2.1). Whenever memory operations are still uncoalesced, a code transformation is applied to optimise these operations for coalescing (Section 5.2.2);
4. discover reuse of data within thread blocks. This enables a code transformation to cache data on the GPU's on-chip memory (Section 5.3)

While this thesis restricts the programming model to CUDA and OpenACC, the presented tools are applicable to all GPGPU programming models.

1.3 Organisation of thesis

The remainder of this thesis is organised as follows. Chapter 2 gives all required background information on the GPU's hardware and software architecture and the program analysis techniques that are used by our tools. Chapter 3 gives a summary of the related work. The next two chapters elaborate on our contributions. Chapter 4 discusses the generation of efficient copy and allocation strategies. Chapter 5 focusses on optimising the GPU kernel (coalescing and reuse analysis). In Chapter 6 the proposed tools are evaluated for a set of applications. Chapter 7 touches the topic of future work. Finally, Chapter 8 gives a conclusion on the work that is presented in this thesis.

Chapter 2

Background

Our research is related to the following two fields: *GPGPU programming* and *program analysis*. In order to understand the issues and solutions that are discussed in the remainder of this thesis, this chapter gives an introduction to each field. Furthermore, this chapter serves as a foundation for the notions needed in later chapters.

Although our research can be applied to every GPGPU platform, we use the NVidia CUDA (Compute Unified Device Architecture) platform [2] in this thesis. The design of other GPGPU platforms is similar. Section 2.1 elaborates on this platform and its corresponding programming model.

In addition to the CUDA platform, the OpenACC programming model [3] is considered in this thesis (Section 2.2). The OpenACC programming model defines a set of compiler directives to indicate loop nests in a CPU program that have to be transformed into GPU kernels by an OpenACC compiler. This compiler is able to transform the indicated loop nests into kernels for a range of accelerators (including CUDA GPUs). OpenACC provides portability over different accelerator platforms and ease-of-programming since the programming model abstracts away from platform dependent details. This comes at the cost of decreased fine-tuning possibilities.

Finally, Section 2.3 addresses the program analysis techniques that are used in this thesis.

2.1 CUDA platform

This thesis focusses on the NVidia CUDA (Compute Unified Device Architecture) platform [2]. The CUDA platform breaks down into two layers: the CUDA hardware architecture and the CUDA programming language (as defined by the CUDA programming model). The hardware architecture is discussed in Section 2.1.1. Section 2.1.2 elaborates on the CUDA programming model. Finally, Section 2.1.3 discusses the CUDA execution model.

2.1.1 Hardware architecture

The computational power of CUDA GPUs is formed by an array of computational components called Streaming Multiprocessors (SMs). In each SM, computations are performed by a set of execution units called symmetric processors (SPs) or CUDA cores. Each CUDA core consists of a fully pipelined integer arithmetic logical unit (ALU) and floating point logical unit (FPU).

For example the NVidia GeForce 640 GT has 2 SMs, each consisting of 192 CUDA cores which amounts to a total of 384 CUDA cores. Thus, the device is capable of sustaining 384 parallel hardware threads.

The SMs and CUDA cores communicate and synchronise with each other through an advanced memory hierarchy (Figure 2.1). The hierarchy is composed of registers, per-SM L1 caches, a GPU-common L2 cache and off-chip DRAM memory. The architecture of the memory hierarchy varies over different versions of the CUDA hardware architecture. In the description that follows, we focus on the most recent versions of the architecture, codenamed *Kepler* and *Fermi*.

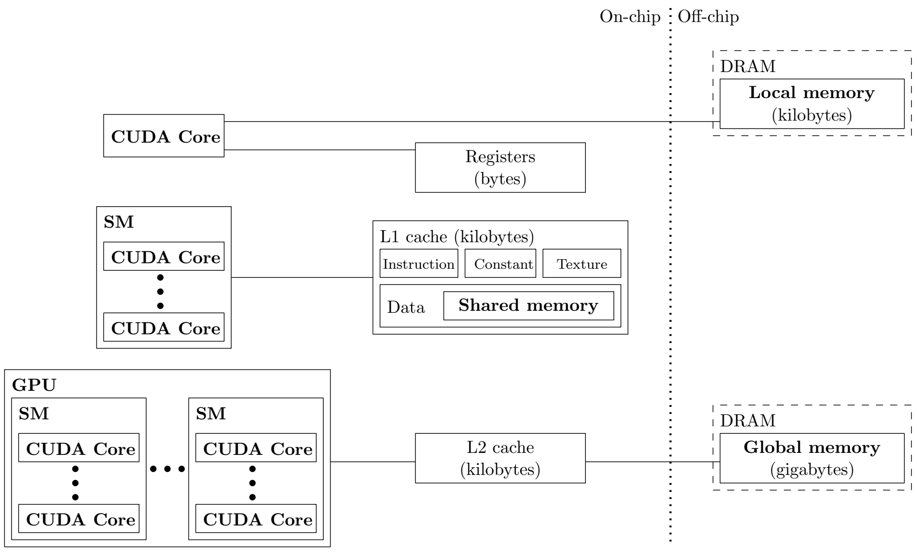
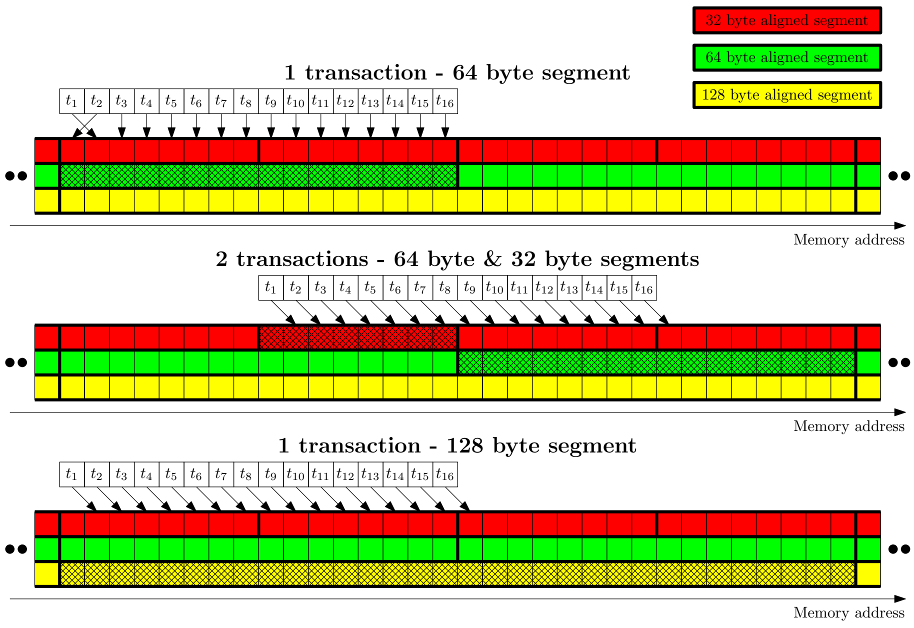


Figure 2.1: CUDA memory hierarchy

At the base of the memory hierarchy, there is the global memory. This memory has the largest capacity among all memories on the GPU and can be accessed by all CUDA cores and the CPU. It is located on the off-chip DRAM of the GPU, which implies large access latencies. In order to reduce the average access costs, the GPU features an on-chip L2 cache which caches recently used data and instructions. In addition to this cache, a GPU relies on the technique of memory coalescing to optimise the bandwidth utilisation. Groups of threads (organised in warps, Section 2.1.3) execute instructions in a single instruction multiple data (SIMD) fashion. Whenever a warp reads neighbouring data-elements on the global memory, these accesses are grouped (coalesced) into larger aligned memory transactions by the memory access unit of the SM. Figure 2.2 shows the coalescing capabilities in different situations.

Each SM has four types of caches: instruction, data, constant and texture cache. All four caches together form the L1-cache of the GPU. In this thesis, we focuss on the data cache. The data cache is configured in two parts. The first part is used for implicit caching of data by hardware [1]. The other part of this cache is called *shared memory*. The shared memory is explicitly managed by software.

[1]NVidia GPUs with older architectures then Kepler or Fermi do not have this hardware managed data cache.



32 byte aligned segment
64 byte aligned segment
128 byte aligned segment
1 transaction - 64 byte segment
t_1 t_2 t_3 t_4 t_5 t_6 t_7 t_8 t_9 t_{10} t_{11} t_{12} t_{13} t_{14} t_{15} t_{16}
Memory address
2 transactions - 64 byte & 32 byte segments
Memory address
1 transaction - 128 byte segment
Memory address

Figure 2.2: Memory accesses within a warp are grouped into memory transactions of 32, 64 or 128 byte aligned memory segments. In this example, warps consist of 16 neighbouring threads.

Access latencies to the L1 cache are significantly lower compared to global memory accesses. In order to decrease average access latencies, the L1 cache is organised in banks. Accesses to addresses that fall into different banks are served simultaneously. However, if the accesses are within the same bank, a *bank conflict* occurs and the accesses are serialised.

Each SM has a set of registers that are shared among all CUDA cores. If all registers are occupied, data is spilled to the off-chip local memory that is located on the DRAM of the GPU.

2.1.2 CUDA programming model

The CUDA programming model is a heterogeneous programming model that hides the complexity of the GPU hardware architecture from the programmer. This is achieved through a set of abstractions concerning the organisation of threads, the memory hierarchy and synchronisation primitives. The CUDA programming language is implemented in the C language (ANSI C99 standard). Figure 2.3 emphasises important concepts of the CUDA programming model by giving the native C implementation of an array-multiplication function and its CUDA equivalent kernel.

A CUDA program is composed of a *host part* (executed by the CPU) and a *GPU part* (executed by the GPU). The GPU part is composed of a set of computational units (kernels) which are invoked by the host.

The host specifies the thread configuration for each invocation of a GPU kernel, stating the number of GPU threads that are launched. The main idea of GPGPU programming is to divide the work over a large number of threads, preferably more as the number of available CUDA cores. In this way CUDA cores can always be occupied and memory access latencies are hidden with computations. Kernels are launched asynchronously to the host.

CPU

```
void eltMult(int n, int *a, int *b, int *c)
{
    int i;
    for (i = 0; i < n; i++)
        c[i] = a[i] * b[i];
}
```

CUDA

```
__global__ void cuda_eltMult(int n, int *a, int *b, int *c)
{
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    if (i < n)
        c[i] = a[i] * b[i];
}

void foo(int n, int *a, int *b, int *c)
{
    int *da, *db, *dc;
    dim3 blockSize(8);
    dim3 gridSize((n/8) + 1);
    cudaMalloc(da, n * sizeof(int));
    cudaMalloc(db, n * sizeof(int));
    cudaMalloc(dc, n * sizeof(int));
    cudaMemcpy(da, a, ...);
    cudaMemcpy(db, b, ...);
    cuda_eltMult<<<gridSize, blockSize>>>(n, da, db, dc);
    cudaMemcpy(c, dc, ...);
}
```

🟥 Host code 🟩 Device code 🟦 CUDA call 🟫 Kernel call

Figure 2.3: CUDA code example

Threads are organised in a grid of *thread blocks* (groups of threads). The grid of thread blocks and the individual thread blocks can have up to three dimensions. Threads are identified by means of its corresponding thread block identifier and thread identifier within the thread block. With this information a programmer can coordinate the work that is done by individual threads. Figure 2.4 gives an example of the CUDA thread configuration.

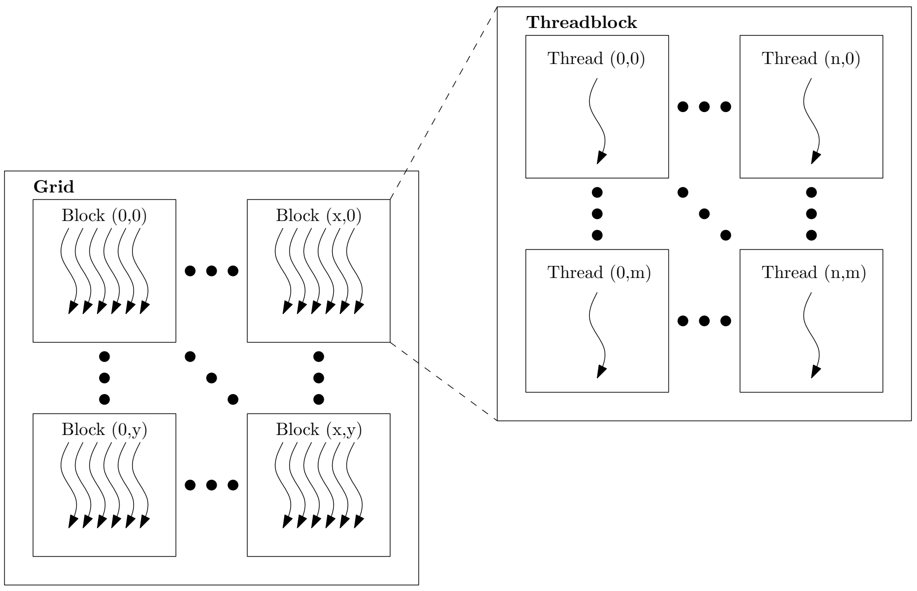


Figure 2.4: Example CUDA thread configuration. This thread configuration consists of a two-dimensional grid with two-dimensional thread blocks.

The programming model assumes that both the host and the GPU have their own separate memory spaces, and the host part manages the GPU memory through calls to the CUDA runtime. Copy actions between the CPU and GPU can either be executed synchronously or asynchronously to the host. For a synchronous copy action, the CUDA runtime returns the call after the copy action has been completed. For a asynchronous copy action, the CUDA runtime returns the call directly after the copy action has been started.

In the CUDA programming model, host memory can be allocated in two ways: *pageable* and *pinned*. By default, host memory is allocated as pageable memory. Pageable memory can be swapped out from the CPU's DRAM memory to the hard drive to facilitate infinite address spaces on limited sized memory. When memory is allocated in a pinned fashion, the allocated memory segment always resides in the CPU's RAM memory. The GPU fetches pinned memory through direct memory access (DMA) which implies higher CPU-GPU communication bandwidths, compared to pageable memory. Pageable memory cannot be fetched through DMA. Furthermore, pinned memory facilitates asynchronous communication. Allocating pinned memory takes significantly more time compared to allocating pageable memory. By experiments, we found that allocating 100 megabyte of pinned CPU memory takes more then 100 ms of time. Allocating pageable memory takes less then 5 ms. In this report we use pageable memory by default, unless pinned memory is required for asynchronous data transfers.

Copy actions or kernel launches can be divided over multiple *streams*. A stream is a sequence of operations that are executed on the GPU in-order. Streams may be executed concurrently.

To make sure that asynchronous actions of the CUDA runtime have been finished at certain points in the host code, synchronisation barriers can be called.

In the GPU kernel, threads within a thread block can be synchronised by using barrier synchronisations at indicated moments in the kernel's code. Barrier synchronisation between thread blocks can not be performed, due to the freedom of the CUDA execution model to schedule thread blocks freely over the SMs (Section 2.1.3).

As discussed in Section 2.1.1, a GPU has a heterogeneous memory hierarchy. Figure 2.5 shows the range of memories that is available through software. Data that is stored in the texture/constant and global memory persists across multiple kernel invocations. Other memories are not persistent.

2.1.3 CUDA execution model

Upon the host invoking a GPU kernel, all thread blocks are distributed over the SMs with available compute capacity by the GPU's scheduling unit. On the SMs, the thread blocks are split into groups of threads which are called *warps*. A warp is a group of consecutive threads in the X-dimension of the thread block. The size of a warp is fixed by the GPU's architecture. Warps are in one of the following three states: *ready*, *active*, *waiting-for-memory*. Warps in the ready-state are ready to execute a set of instructions. The SM's warp scheduler assigns a warp to a group of available CUDA cores. The warp becomes active. The CUDA cores execute instructions in a Single Instruction Multiple Data (SIMD) fashion. When memory is being accessed, execution of the warp is blocked and the warp is in the waiting-for-memory state. When data is available, the warp will go to the ready state again to execute new instructions.

Threads occupy registers and memory space. Since a SM is equipped with limited memory and registers, there is a maximum on the number of warps that is scheduled simultaneously on a SM. The ratio of active warps to the maximum number of warps that is supported by the warp

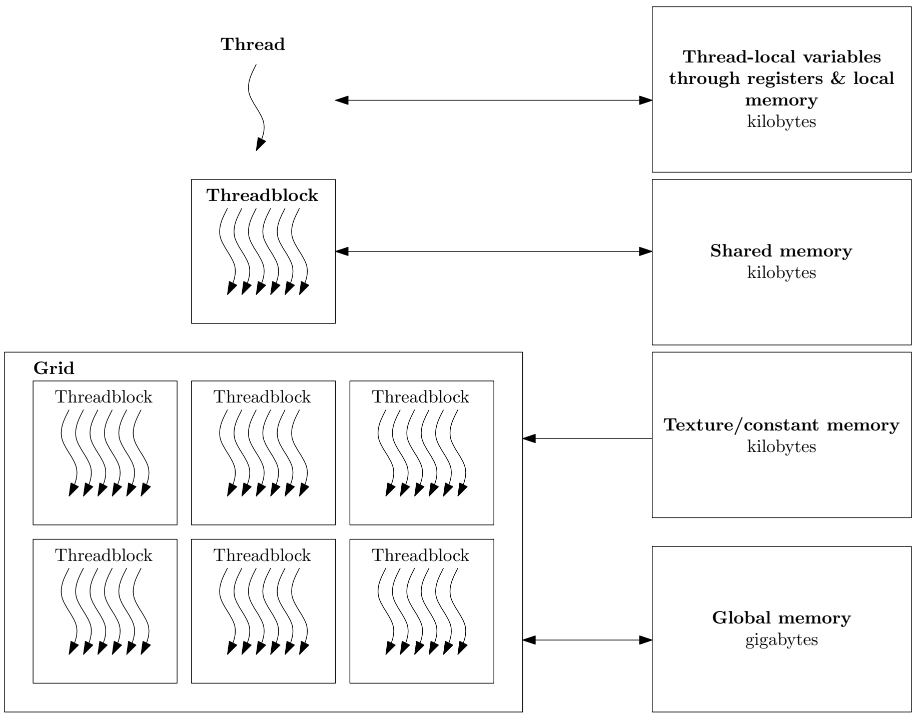


Figure 2.5: Threads to memory mapping

scheduler is called the *occupancy factor*. When the occupancy factor decreases, the number of active warps decrease and the kernel's execution time increases.

2.2 OpenACC programming model

As discussed in Section 2.1.2, GPU programming models expose GPU specific concepts to the programmer (advanced memory hierarchy, the concept of thread blocks and synchronisation primitives). Therefore, generating GPGPU accelerated programs is a time consuming task for programmers who are no expert in the domain of GPU programming. Furthermore, every GPU programming model has its own architecture specific artefacts. Hence, porting code from one programming model to the other is non-trivial.

In order to abstract these issues away, the OpenACC programming model [3] is developed by CAPS, CRAY, PGI and NVidia. The OpenACC programming model consists of a set of compiler directives. With these directives, programmers can annotate loop nests in a CPU program. These loop nests are transformed into GPU kernels by an OpenACC compiler. Listing 2.1 shows an example of an OpenACC annotated program.

If an annotated loop nest can be parallelised, the OpenACC compiler generates the GPU kernels and the control code fragments that are executed by the host. Furthermore, the compiler schedules copy and allocation actions for the data-structures that are used in the GPU kernel and searches for an optimal thread configuration.

The majority of OpenACC compilers performs static analysis (Section 2.3.1) of the program during compilation. These compilers have limitations in the code constructions that can be analysed (Section 2.3.1). A typical OpenACC compiler is unable to handle pointer arithmetic; therefore compilation fails. The compiler cannot determine upfront the value of pointers. Different pointers may map to the same data-structure, therefore imposing dependencies (Section 2.3.4) that limit parallelism in the GPU kernel. In order to succeed, programmers have to add extra directives to aid the compiler in solving these issues. For example, a programmer can add the `independent` directive to annotated loops to ensure an OpenACC compiler that a loop nest contains no obstructing dependencies and can therefore safely be parallelised.

By default, copy actions and kernel launches are performed synchronously with the host. However, a programmer can add directives to launch kernels and copy-actions asynchronously. Synchronisation barriers can be inserted to synchronise GPU activities with the host.

The programmer can insert directives to optimise or correct the implicit data-management strategy chosen by the compiler. By adding directives like `create`, `copy`, `copy-in` and `copy-out`, he is able to control the moment where data-structures should be copied or allocated, together with the exact size of the data-structures.

Listing 2.1: OpenACC example

```
void eltMult(int n, int *a, int *b, int *c)
{
  int i;
#pragma acc kernels loop independent
  for (i = 0; i < n; i++)
    c[i] = a[i] * b[i]
}
```

One of the main benefits of the OpenACC programming model is the abstraction from GPU specific parameters like thread configurations and the GPU's advanced memory hierarchy; the responsibility for these low-level GPU programming issues is delegated to the compiler. This decreases development time at the cost of less optimisation possibilities. Another benefit is portability over different accelerator platforms. Currently, OpenACC compilers are capable to translate OpenACC annotated programs into CUDA and OpenCL [4] compatible kernels or multithreaded CPU code if no GPU is available. This makes the programming model suitable for a wide range of accelerators.

Currently, CAPS, CRAY and PGI have commercially available OpenACC compilers. In this thesis, the OpenACC compiler of PGI is used.

2.3 Program analysis

Program analysis is the process of inferring properties by analysing the behaviour of a computer program. Examples of properties that can be inferred are accessed address ranges of a memory operation, data dependencies or the life-span of variables. Applications that rely on program analysis techniques are for example compilers (using program analysis for code improvement) and software validation frameworks (using program analysis for error detection). Two main approaches for program analysis can be distinguished: static analysis and dynamic analysis. The framework that is designed in this thesis uses dynamic analysis to infer properties of a program. However, the tools can easily be transformed to make use of static analysis.

The remainder of this section is organised as follows. In Section 2.3.1, we touches the topic of static analysis techniques. Section 2.3.2 focusses on dynamic analysis techniques and describes the analysis framework of Vector Fabrics that is used as a basis for our tools. Section 2.3.3 discusses the *extended call graph*. An extended call graph gives a representation of a program's execution flow and is used throughout this thesis. Finally, in Section 2.3.4 we discuss the topic of dependency analysis.

2.3.1 Static program analysis

With static analysis, properties are inferred by analysing the source (either source code or object code) of a program, with formal methods like data flow analysis [5] and the polyhedral model [6]. The properties that are inferred by static analysis are sound [5]. That is, the properties can always be proven correct, regardless of the program's input.

However, static analysis may fail on complex programs. For example, it is hard to infer data dependencies statically if the code uses indirect addressing, pointers recursion and indirect function calls. This is in line with Rice's theorem [7] which states that only trivial properties of programs are algorithmically computable.

2.3.2 Dynamic program analysis

Dynamic analysis uses execution traces for analysis. By observing these traces, dynamic analysis can solve issues like indirect addressing, pointer recursion and indirect function calls, since the analysis has knowledge about the actual program's execution flow. These issues are hard to solve by static analysis approaches.

The properties as inferred by dynamic analysis may not be sound. The set of observed execution traces is a subset of all program flows that are possible. Therefore, the inferred properties may not hold for those cases that are not observed during analysis. However, it is programmers best practice to have a large set of test cases that cover all practical situations.

Base-line analysis The tools that are proposed in this thesis are using dynamic analysis. The base-line framework for this analysis is implemented by Vector Fabrics and is schematically shown in Figure 2.6. The analysis framework takes a program's source code as an input. The source

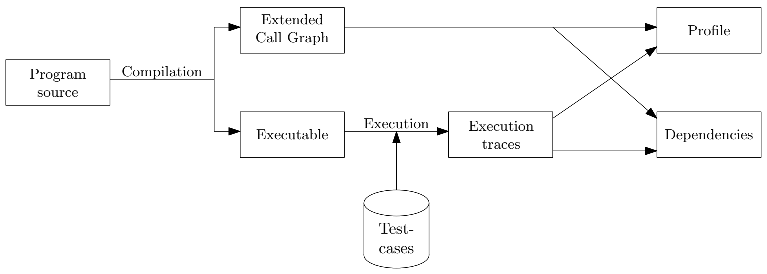


Figure 2.6: Base-line analysis framework

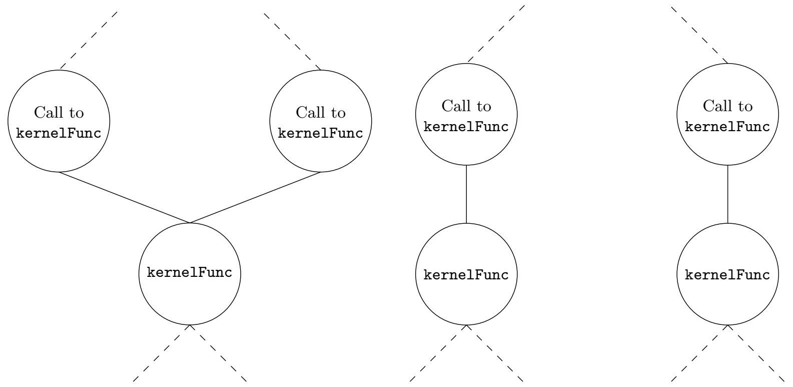
code is compiled to an executable. As a side effect of this, the extended call graph (ECG) (Section

2.3.3) is created, representing the execution flow throughout the program. The ECG is lacking information about issues like the number of iterations that are executed for each loop or the number of times a branch is taken. To retrieve this information, the program is executed for a set of test cases. Based on the observed execution traces and the ECG, the base-line analysis framework generates a profile that contains execution specific information for each operation or code fragment that is represented by a node in the ECG.

The base-line analysis framework uses the ECG and execution traces to detect data dependencies (Section 2.3.4) that impose constraints on the execution order of operations.

2.3.3 Extended call graph

An extended call graph (ECG) is a tree-like structure that represents the execution flow throughout the program. Each node in the graph represents a function, a sequence of instructions, a loop, a call, a load or a store operation. This graph is an unfolded version of a call graph [8] extended with extra information like the number of invocations and executed iterations. In traditional call graphs, different invocations of the same code fragment map to the same node in the call graph (Figure 2.7(a)). In the extended call graph, different invocations are mapped to different nodes (Figure 2.7(b)). Each statement in the original source code maps to a set of nodes in the ECG. Each node in the ECG maps to a set of statements in the source code. As an example, Figure 2.8 shows the resulting ECG for a given input program.



(a) In traditional call graphs, nodes are shared over different invocations.

(b) In the extended call graph, nodes represent individual invocations of code fragments.

Figure 2.7: A extended call graph is an unfolded version of a traditional call graph

Each node is indicated by a key. With this key the position of two nodes within the ECG can be compared. Based on this key, information about the order, in which nodes are executed, is found. When the key of a node X is smaller than the key of a node Y, it means that the code fragment of node X is invoked earlier in the program than the code fragment of node Y.

```
void main (void)
{
  int *A = malloc(sizeof(int) * 100);

  for (int i = 0; i < 100; i++)
    A[i] = i;

  for (int i = 0; i < 100; i++)
    printf("%d\n", A[i]);

  free(A);
}
```

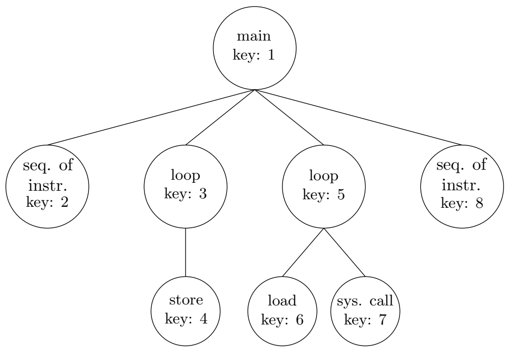


Figure 2.8: resulting ECG for input program

2.3.4 Data dependency analysis

Data dependency analysis is the technique of finding data dependencies in a program. Data dependencies impose constraints on the execution order of instructions. In this thesis we use data dependencies to reveal the data flow throughout the program. A data dependency is defined by a group of source operations S and a group of destination operations D. All operations in S are invoked before the operations in D. Both groups of operations are related to each other since they operate on the same registers or memory locations. Table 2.3.4 shows the three types of data dependencies together with the function of the source S and destination D nodes.

Table 2.1: Types of data dependencies

Data-dependency	S	D
Write-after-write	Write	Write
Write-after-read	Read	Write
Read-after-write	Write	Read

The tools of Vector Fabrics are able to observe these three kinds of data-dependencies. However, only the read-after-write data dependencies are relevant within the scope of this thesis, since these dependencies reveal data flow throughout the program.

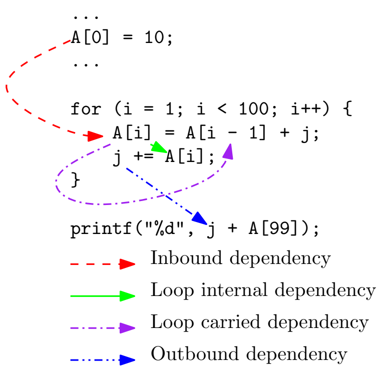
A read-after-write data dependency d is defined by a set of operations P_d that produces the related value and the operations C_d that consume the value after C_d is finished (2.1). All operations in P_d and C_d relate to different nodes in the ECG.

$$d = P_d \times C_d \tag{2.1}$$

Figure 2.9 shows the four types of read-after-write dependencies that can be identified in relation to a code fragment.

- **Inbound dependencies** Data is produced by P_d before the code fragment and is consumed by C_d inside the code fragment.
- **Outbound dependencies** Data is produced by P_d inside the code fragment and is consumed by C_d after the code fragment finished.

- **Internal dependencies** Data is produced and consumed within the same iteration of the code fragment.
- **Loop-carried dependencies** This type of dependencies can only occur in loops. Data is produced and consumed in different iterations of the loop. Hence, data needs to be carried over to later iterations of the loop.



```
...
A[0] = 10;
...

for (i = 1; i < 100; i++) {
    A[i] = A[i - 1] + j;
    j += A[i];
}

printf("%d", j + A[99]);
```

Figure 2.9: Different types of data dependencies

The dependencies carry information on the data-structures and the data-elements that are involved.

Chapter 3

Related work

The idea of designing tools to aid programmers or automatic frameworks in the generation process of GPGPU accelerated programs is not new. Work has been done on both the generation of efficient data-transfer and allocation strategies and optimising GPU kernels. However, the majority of related work uses a static analysis approach, due to the soundness properties. The analysis capabilities of these approaches are more limited compared to dynamic approaches (that are used in this thesis). Code fragments that use indirect addressing, pointer aliasing or pointer arithmetic are difficult to analyse. Parts of the proposed techniques can also be applied in a dynamic analysis context.

We present the related work in two parts: In the first part we focus on the work that is related to the generation of efficient data-transfer and allocation strategies. In the second part we present the work that is related to optimising GPU kernels. This is a broad field of research. We only discuss work that proposes techniques for coalescing analysis and data caching on on-chip memory.

Data transfer and allocation strategies To the best of our knowledge, this is the first work that uses a fully dynamic analysis approach for the generation of data transfer strategies between a host and an accelerator.

Custers et al. [9] propose a framework to classify loop nests for parallelisation opportunities. Classification is based on matching code fragments to predefined patterns. The classification framework observes the set of data-structures that needs to be transferred between the host and the accelerator in order to execute the application correctly. This information is used to construct *memory-regions*. A memory-region for a data transfer action t is defined by a pair of points in the source code $pos_{start} \times pos_{end}$ between which t is to be executed. At pos_{start} the transfer action can be started asynchronously to the host. At pos_{end} the data transfer should be finished. A set of optimisations is proposed to position the memory regions such that the data transfer overhead is minimised. The performance of the framework of Custers in generating data transfer and allocation strategies is limited by the number of patterns that are supported by the framework for classification. Whenever a loop nest cannot be classified for parallelisation by the framework, the memory regions cannot be generated. Hence, no data transfer strategy is generated. Our framework depends on information that is extracted from observing dependencies and is therefore decoupled from the parallelisation classification.

Amini et al. [10] propose a compiler step to generate an efficient data transfer strategy. The control and data flow graph of the application is used for analysis. The tool of Amini uses two simple heuristics in scheduling transfer actions: schedule transfer actions to the GPU as early as possible and schedule transfer actions back to the host as late as possible. Redundant communications due to data reuse between kernel executions are avoided. Allocation and free actions of the corresponding data-structures are managed by a run-time library that uses the global memory as a cache memory. Data-structures are preserved on the GPU as long as there is enough memory on the GPU. When a kernel execution requires more memory than is available, the run-time library frees memory ranges that are the least recently used. The compiler step of Amini is unable to deal with issues like indirect addressing and pointer aliasing since static analysis is used. Since we use dynamic analysis for our tools, we can solve these issues.

Jablin et al. [11, 12] introduce a CPU-GPU communication framework that uses a combination of static and dynamic analysis to generate a data transfer and allocation strategy. The framework consists of two parts, a run-time library and an optimising compiler. The compiler schedules copy and allocation actions efficiently throughout the program in the same way as is done by Amini. Data transfers to the GPU are scheduled early in the program and retrieving it only when necessary. However, questions that are difficult to answer (like the size and shape of copied data-structures) are postponed to the runtime library. The runtime library tracks GPU memory allocations and transfers data between the CPU memory and GPU memory. By dividing responsibilities over a compiler step and a runtime library, CPU-GPU communication does not need to depend on the strength of static compile-time analysis or on programmer-supplied annotation. Compared to Amini et al., more complex code structures can be analysed. However, this comes at the cost of extra overhead due to the runtime library. Our work can solve the issues that cannot be solved by the compiler of Jablin. Therefore, there is no need for adding runtime libraries.

Optimising the GPU kernel In literature, a wide set of GPU kernel optimisations is proposed. Mostly they are implemented as compiler steps that use static analysis. In what follows next, research is highlighted that is related to the work as presented in this thesis: coalescing analysis (contribution 3 of Section 1.2) and caching on on-chip memories (contribution 4 of Section 1.2).

Yang et al. [13] propose a set of optimisations to improve the performance of naive GPU kernel implementations. One of these optimisations is to use buffers in the shared memory to allow threads to read from global memory in a coalesced way. In order to use this optimisation, access patterns of the read operations are reconstructed by using static analysis. The analysis of Yang is unable to deal with indirect addressing. Whenever array indices (e.g. `x` in `a[x]`) are loaded from memory, the analysis for optimisation is skipped. The static analysis cannot calculate the value of these indices. Our tools reconstruct access patterns by observing execution traces and can therefore reconstruct access patterns where indirect addressing is used.

Baskaran et al. [6] present a framework to efficiently map data segments over an advanced memory hierarchy of DRAM memory and scratchpad memories. By caching data on high bandwidth and low latency scratchpad memories, performance increases. The polyhedral model is used for analysis. Baskaran focusses in his work on the GPU's memory hierarchy. More specifically, opportunities are explored to use the GPU's on-chip scratchpad memory. Baskaran states that since processing units can use both the GPU's off-chip and on-chip scratchpad memory throughout the program, mapping data to the scratchpad memory is only beneficial when there is sufficient reuse between threads in a thread block. In order to determine the level of reuse for a data-structure, overlap is determined in the range of data-elements that is accessed by neighbouring threads. To obtain this information the shapes of the accessed data-structures should be known. However, with dynamic analysis these shapes cannot be observed. Dynamic analysis can only observe accesses in a linear address space. Only by using symbol information (as can be extracted with static analysis), shapes with a higher dimensionality can be reconstructed.

Within Vector Fabrics, a series of projects were done on the topic of GPGPU programming. The following project relates to the kernel optimisations that are proposed in this thesis.

Juravle et al. [14] propose a set of analysis tools that enable automatic and guided transformation of sequential programs to GPU kernels. Part of this is a tool to explore kernel optimisations (coalesced memory access and shared memory exploration). The toolset checks for coalescing behaviour of a GPU kernel based on expression reconstruction analysis. Juravle states that a memory read operation is only coalesced when its corresponding access function has the property of monotonicity. However, monotonicity cannot be determined for every symbolic expression due to the static analysis techniques. Based on the coalescing analysis, a thread configuration is generated that maximises the coalescing capabilities. Furthermore, Juravle also presents a tool to check for reuse of data between threads in a thread block by using static symbol range analysis. The tools of Juravle use static analysis and do therefore not fit into the dynamic analysis framework of Vector Fabrics.

Chapter 4

GPU data-transfer and allocation

This chapter proposes a tool to generate efficient data-transfer and allocation strategies in GPGPU programming. By considering memory dependencies (as generated by the dynamic analysis of Vector Fabrics), the tool schedules a minimal set copy and allocation actions in a GPGPU program; i.e. the amount of transferred data and allocation overhead is minimised.

As stated in Chapter 2, the memory space of a GPU is logically separated from the CPU's memory space. OpenACC compilers can schedule transfer and allocation actions automatically. However, this is only successful for simple programs. The static analysis fails, if pointer arithmetic is involved. For example, the OpenACC compiler of PGI cannot parallelise a matrix multiplication function (`matrixMult(A, B, C)`) into a GPU kernel when `A`, `B` and `C` are pointers to dynamically allocated data-structures. The compiler can not calculate the addresses of pointers `A`, `B` and `C`. `A`, `B` and `C` may point to the same data-structure, and can therefore impose dependencies that obstruct parallelisation. A programmer must help the compiler by adding extra compiler directives. In the example of the matrix multiplication kernel, the programmer indicates that `A`, `B` and `C` do not point to the same data-structures. In the CUDA programming model, all transfer actions have to be scheduled by the programmer. For both programming models, manually generating a correct data transfer and allocation strategy takes time, especially when sequential programs need to be ported to GPU kernels. Data-flow patterns, as generated by other programmers may be complex. Understanding these patterns is a time-consuming task.

Besides the logical separation of memory hierarchies, the memory hierarchy of a typical GPU is also physically separated from the host. Data-transfers between the CPU's RAM memory and the GPU are executed over a system bus that has relatively low bandwidth compared to the GPU's computational power. Figure 4.7 shows the achieved bandwidths of the PCI-express 2.0 and PCI-express 3.0 system busses of the systems we use for evaluation (Chapter 6). The PCI-express 3.0 bus achieves a maximal bandwidth of 11 GB/s for pinned memory and 5 GB/s for pageable memory. The results for a PCI-express 2.0 bus are worse (2.5 GB/s for pinned memory and 1.5 GB/s for pageable memory). Bandwidth decreases dramatically when smaller data-segments are transferred. The low bandwidth of the system bus can annihilate all gain obtained by using a GPU for computations.

Because of the large costs of transferring data between the CPU and GPU, the amount of bytes that is transferred between CPU and GPU should be minimised. A typical OpenACC compiler

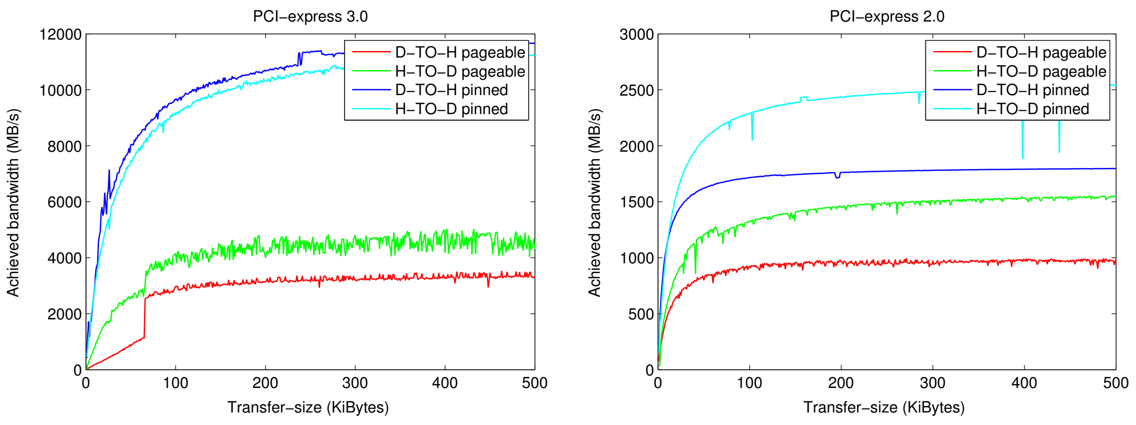


Figure 4.1: Achieved bandwidth in CPU-GPU communication. H-to-D: Data-transfers from the host (CPU) to the device (GPU). D-to-H: Data-transfers from the device (GPU) to the host (CPU).

will always schedule data transfers around the kernel's boundaries. This is a naive approach that causes significant overhead. Data-structures may be transferred between the CPU and GPU redundantly (as was seen in Figure 1.1). The data transfer strategy can be optimised by adding compiler directives throughout the code to modify the position of transfer and allocation actions. Finding the optimal position for data transfer and allocation actions is a time consuming process for programmers in both the CUDA and OpenACC programming model.

Besides optimising the position of transfer actions, memory transfers can be performed asynchronous to the CPU and GPU to hide them with computations. This is possible since GPUs are equipped with Direct Memory Access controllers to directly access the memory spaces of the GPU and the CPU, without intervention of the CPU. To enforce correct behaviour a programmer needs to insert synchronisation barriers, to ensure data-transfers have completed at certain moments.

This chapter describes a tool to generate optimised data-transfer strategies. In order to generate a transfer and allocation strategy the framework performs the following three steps:

1 Find all variables that should be transferred, allocated and freed on the GPU for each individual kernel function (Section 4.1)

2 Reduce the number of redundant copy, allocation and free actions by considering all GPU kernels together (Section 4.2)

3 Schedule data-transfer, allocation and synchronisation actions throughout the code (Section 4.3)

As a running example through this chapter we use the program as is shown in Listing 4.1. It is an extended version of the matrix multiplication application that is shown in Figure 1.1. In the application a series of calculations are performed in a loop. During each iteration of the loop, matrices `A`, `B`, `C` and `D` are filled with random floating point numbers. Matrix-multiplication is applied to matrices `A`, `B` and `C`, `D`. The results are stored in `E` and `F`. Matrices `E` and `F` are again multiplied and the results are stored in `G`. As a final step matrices `E`, `F` and `G` are consumed in a CPU function (`totalSum`). Each invocation of the matrix multiplication function is transformed in a separate GPU kernel. Listing A.1 shows a naive data-transfer strategy as is generated by a typical OpenACC compiler. All required data-structures are copied in to the GPU before a kernel starts and all data-structures that are used after the kernel execution are transferred back to the CPU after the kernel ends. In this strategy redundant copy actions are produced. An example is the copy-in action for matrices `E` (line 39) and `F` (line 40) before `matrix_mult(E, F, G)`. Listing A.2 shows the data-transfer and allocation strategy as is generated by our framework. The redundant copy-in actions for matrices `E` and `F` are removed. All remaining copy actions are scheduled at an optimal (i.e. the number of transferred bytes is minimal) position and executed asynchronously to overlap with computations. The allocation and free actions of the corresponding data structures are scheduled outside the loop. The chapter shows how the optimal strategy is generated step by step.

Listing 4.1: Input program

```
void main(void)
{
  ...
  for (i = 0; i < N; i++) {
    randomize_array(A);
    randomize_array(B);
    randomize_array(C);
    randomize_array(D);
    matrix_mult(A, B, E);
    matrix_mult(C, D, F);
    matrix_mult(E, F, G);
    total_sum(E, F, G);
  }
  ...
}
```

4.1 Individual kernels

In the first step (of the three steps) in generating an efficient data transfer and allocation strategy, all information is gathered on the set of data-structures that is used in each individual GPU kernel. More specifically, this step gathers all information to generate the allocation and data-transfers actions that are required for an individual kernel in order to be executed correctly.

The tool retrieves the set of read-after-write dependencies D_k that is related to a kernel k. These dependencies describe the flow of data in relation to k. As explained in Section 2.3.4, three types of dependencies can be related to a kernel: inbound, outbound and internal dependencies. For every dependency d, symbol information can be retrieved, to retrieve the data-structure x that is involved in a dependency.

All dependencies in D_k are grouped in the set of inbound ($D_{inbound,k,x}$), outbound ($D_{outbound,k,x}$) and kernel internal ($D_{internal,k,x}$) dependencies for every data-structure x. These groups of dependencies contain all information to generate all transfer and allocation actions that need to be executed for k.

This step generates a initial set of *kernel actions* for every kernel k. A kernel action is a data-transfer or allocation action that needs to be done in relation to k. A kernel action is represented by a set of dependencies that causes this action to be scheduled.

The tool generates four types of actions for a data-structure x in a kernel k.

- **Copy-in** For data-elements that are produced before an invocation of kernel k and are used inside k, a copy-in action ($a_{copy-in,k,x}$) is generated to transfer these data-elements from the CPU to the GPU. This copy-in action is described by the set of inbound dependencies (4.1).
- **Copy-out** For data-elements that are produced inside k and are used afterwards, a copy-out action ($a_{copy-out,k,x}$) is generated to transfer these data-elements from the GPU to the CPU. This copy-out action is described by the set of outbound dependencies (4.2).
- **Allocation & Free** Data-elements that are somehow used inside k, since they are either consumed or produced, should be allocated ($a_{allocation,k,x}$) before the kernel and freed ($a_{free,k,x}$) after the kernel. The data-elements that are used in the kernel are reflected by the inbound, outbound or internal memory dependencies (4.3) (4.4).

$$a_{copy-in,k,x} = D_{inbound,k,x} \tag{4.1}$$

$$a_{copy-out,k,x} = D_{outbound,k,x} \tag{4.2}$$

$$a_{allocation,k,x} = D_{inbound,k,x} \cup D_{outbound,k,x} \cup D_{internal,k,x} \tag{4.3}$$

$$a_{free,k,x} = D_{inbound,k,x} \cup D_{outbound,k,x} \cup D_{internal,k,x} \tag{4.4}$$

The kernel actions for data-structures $x_1, \dots, x_n$ that are used inside kernel k are grouped according to the 4-tuple as shown in (4.5). All 4-tuples that are generated for the individual GPU kernel are taken to the next step.

$$\begin{aligned} A_k =& \{a_{copy-in,k,x_1}, \dots, a_{copy-in,k,x_n}\} \times \\ & \{a_{copy-out,k,x_1}, \dots, a_{copy-out,k,x_n}\} \times \\ & \{a_{allocation,k,x_1}, \dots, a_{allocation,k,x_n}\} \times \\ & \{a_{free,k,x_1}, \dots, a_{free,k,x_n}\} \end{aligned} \tag{4.5}$$

Table A.1 shows the actions that are generated for the example program.

4.2 Intra-kernel optimisations

The resulting 4-tuples as generated by the previous step (Section 4.1) consider GPU kernels separately. However, when this set of kernel actions (as represented by the set of corresponding dependencies) are scheduled around each kernel, this may result in redundant data-transfers.

The optimisation step that is explained in this section reduces the number of redundant kernel actions when multiple kernels exist in the input program. In the naive transfer and allocation strategy which is shown in Listing A.1, the copy-in actions of matrices `E` and `F` (lines 38 & 39) are redundant since the data is not modified by the host between the producing GPU kernels (lines 16 & 29) and the consuming GPU kernel (`matrix_mult_kernel(E, F, G)` on line 40). Furthermore, matrices `E` and `F` are allocated and freed multiple times (lines 13, 20, 25, 29, 35, 36, 37, 42, 43, 44) for each iteration of the surrounding loop (lines 4-49).

Copy-in & Copy-out In order to reduce the number of copy-in actions, each non-empty $a_{copy-in,k,x}$ is considered for kernel k and data-structure x. Each dependency $d \in a_{copy-in,k,x}$ is examined in the following way. When the complete set of producing operations P_d for d is located inside GPU kernels, this means that all data-elements that are related to this dependency are produced by GPU kernels. Hence, the related data-elements already reside on the GPU's memory. Therefore d can be removed from $a_{copy-in,k,x}$ in the further process of generating a transfer and allocation strategy.

In the case of copy-out actions, the consumer operations C_d of each related outbound memory dependency $d \in a_{copy-out,k,x}$ are considered. When all instructions in C_d are located inside GPU kernels, this means that all data-elements that are related to the copy-out action are never consumed by the CPU. d is left out from $a_{copy-out,k,x}$.

In the running example, the copy-in actions of matrices `E` and `F` (lines 38 and 39 in Listing A.1) are removed. The tool observes that all inbound data-dependencies for `matrix_mult(E, F, G)`, related to `E` and `F`, have all producing instructions in the GPU kernels. Therefore these dependencies are removed from further analysis. This situation is depicted in Figure 4.2.

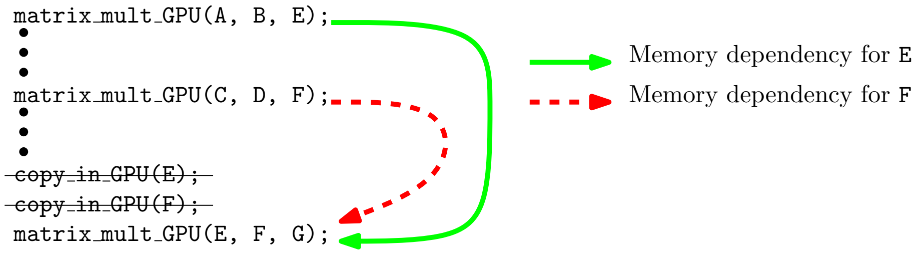
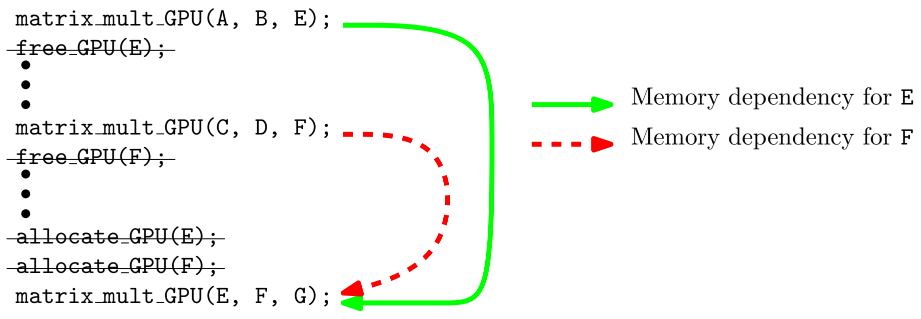

`matrix_mult_GPU(A, B, E);`
•
•
•
`matrix_mult_GPU(C, D, F);`
•
•
•
~~`copy_in_GPU(E);`~~
~~`copy_in_GPU(F);`~~
`matrix_mult_GPU(E, F, G);`

Figure 4.2: `copy_in_GPU(E)` and `copy_in_GPU(F)` are removed

Allocate & Free In order to reduce the number of allocation actions for data-structure x, all non-empty actions $a_{allocation,k_1,x}, \ldots, a_{allocation,k_m,x}$ are considered for all GPU kernels $k_1, \ldots, k_m$. All dependencies that are related to these allocation actions are transferred to $a_{allocation,k_i,x}$, where kernel k_i is the first kernel in the program that uses data-structure x. The first kernel has the lowest key in the extended call graph, and will therefore be invoked first. k_i is the first moment in the program where data-structure x should be allocated. All dependencies are transferred to $a_{allocation,k_i,x}$, to ensure that all data-elements of x that are used in at least one of the GPU kernels are allocated at once.

A same approach holds for free actions. In this case all non-empty actions $a_{free,k_1,x}, \ldots, a_{free,k_m,x}$ are considered for all GPU kernels $k_1, \ldots, k_m$. All dependencies that are related to these free actions are transferred to $a_{free,k_i,x}$, where kernel k_i is the last kernel in the program that uses data-structure x. After this kernel is invoked for the last time, x can be freed from the GPU.

In the running example, the tool observes that matrices `E` and `F` are used across multiple GPU kernels. Therefore, the number of required allocation and free actions for these matrices can be reduced. It is even required since redundant copy-in actions (`copy_in_GPU(E, N * size)` and `copy_in_GPU(F, N * size)`) are removed. These matrices are required to be resident on the GPU's global memory. Therefore, the allocation actions for matrices `E` and `F` before kernel `matrix_mult_GPU(E, F, G)`, the free action for matrix `E` after `matrix_mult_GPU(A, B, E)` and the free action for matrix `F` after `matrix_mult_GPU(E, F, G)` are removed. This situation is depicted in Figure 4.3.



`matrix_mult_GPU(A, B, E);`
~~`free_GPU(E);`~~
•
•
•
`matrix_mult_GPU(C, D, F);`
~~`free_GPU(F);`~~
•
•
•
~~`allocate_GPU(E);`~~
~~`allocate_GPU(F);`~~
`matrix_mult_GPU(E, F, G);`

Figure 4.3: `allocate_GPU(E)`, `allocate_GPU(F)`, `free_GPU(E)`, `free_GPU(F)` are removed

Table A.2 shows the modified actions that result from the second step to generate efficient copy and allocation strategies.

4.3 Location of copy, allocation and synchronisation actions

In the final step of the generation process of efficient data transfer and allocation strategies, each kernel action is scheduled in the program code. The actions are scheduled such that the number of redundant data-transfer and allocation actions is minimised. Consider for example the situation of Listing 4.2 where a GPU kernel is nested in a loop that is executed on the host. Scheduling the copy-out action for `A` (that is produced by the GPU kernel `GPU(A)` and consumed by `X(A)`) directly after line 3 (inside the loop) results in redundant invocations of this action (Figure 4.4).

Listing 4.2: GPU kernel `GPU(A)` is nested in a loop. `A` is produced by `GPU(A)` and is consumed by function `X(A)`

```
1  ...
2  for (i = 0; i < N; i++) {
3    GPU(A);
4  }
5
6  ...
7  X(A);
8  ...
```

To reduce the transfer overhead even further, memory transfers are launched asynchronously to the host code and the GPU kernels. In this way data-transfers can overlap with computations. Synchronisation barriers have to be scheduled throughout the program to guarantee that copy actions have ended at a certain point in the code. An example situation is shown in Figure 4.5.

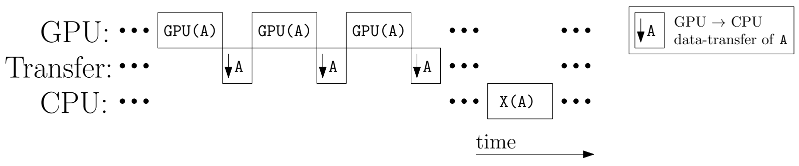


Figure 4.4: Scheduling the copy out action of `A` right after `GPU(A)` results in redundant invocations.

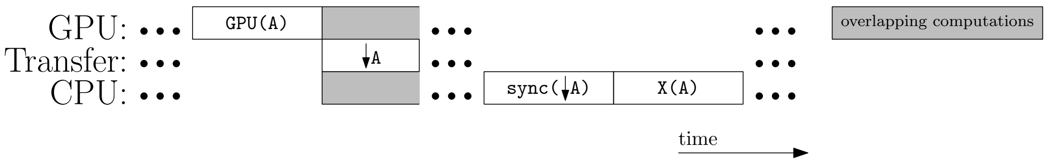


Figure 4.5: The copy-out action for `A` (`↓A`) is performed asynchronous to computations on the GPU and host. A synchronisation barrier is inserted (`sync(↓A)`) to ensure that `↓A` has finished before the first consumption on the host in `X(A)`.

To schedule all kernel actions as generated by the previous step, we propose Algorithm 1. The algorithm schedules a kernel action a in the program in relation to kernel k. A kernel action is represented by a set of related dependencies. The next paragraphs explain this algorithm in more detail. In the final paragraph we focus on the size of the copy and allocation actions.

Get the boundary operations (line 1) In the first step of the algorithm, the *boundary operations* are retrieved in the analysed program. Boundary operations, together with the GPU kernel, define a window in the extended call graph where action a has to take place. The calculated window depends on the context of the action (`copy-in`, `copy-out`, `allocation` or `free`).

- **Copy-in (line 2)** The function GetLastProducerOperations retrieves the latest producing operation p_l among all producing operations P_a that are related to a. After p_l has been executed for the final time, all data related to a is ready to be transferred to the GPU. p_l has the highest key among all instructions P_a in the extended call graph.
- **Copy-out (line 4)** The function GetFirstConsumingOperation retrieves the first consuming instruction c_f among all consuming instructions C_a that are related to a. When c_f is executed for the first time, all data related to a should be copied out from the GPU to the CPU. c_f has the lowest key among all instructions C_a in the extended call graph.
- **Allocation & Free (line 6)** The set of retrieved instructions consists of the final producing instruction p_f and the first consuming instruction c_l over all dependencies that are related to a. This set of operations represents the lifespan of the related data-structure over the set of kernels.

Calculate the least-common-ancestor (line 10) For copy actions, a relationship is identified between the position of an action in the extended call graph (i.e. the depth in the extended call graph) and the number of invocations of the action that is executed. At the extremes, the copy actions can be positioned close to the kernel boundary or close to the operations that are assigned to $Operations$; copy actions are scheduled in the deepest level of the extended call

Algorithm 1 Scheduling of copy, allocation and synchronisation actions

```
$\textsc{ScheduleAction}(a, k)$
▷ Get the boundary operations
if $\textsc{Context}(a) = \textsc{copy-in}$
    then $Operations \leftarrow \{\textsc{GetLastProducerOperation}(\text{a})\}$
elseif $\textsc{Context}(a) = \textsc{copy-out}$
    then $Operations \leftarrow \textsc{GetFirstConsumerOperation}(\text{a})$
    else ▷ $\textsc{Context}(a) = \textsc{allocation} \vee \textsc{Context}(a) = \textsc{free}$
            then $Operations \leftarrow \{\textsc{GetFirstProducerOperation}(a),$
                 $\textsc{GetLatestConsumerOperation}(a)\}$

▷ Calculate the least-common-ancestor
$LCA \leftarrow \textsc{GetLCA}(Operations, k)$

▷ Schedule the action
if $\textsc{Context}(a) = \textsc{copy-in}$
    then $\textsc{ScheduleCopyInAction}(a, \textsc{GetProducingChild}(LCA))$
elseif $\textsc{Context}(a) = \textsc{copy-out}$
    then $\textsc{ScheduleCopyOutAction}(a, \textsc{GetConsumingChild}(LCA))$
elseif $\textsc{Context}(a) = \textsc{allocation}$
    then $\textsc{ScheduleAllocationAction}(a, LCA)$
    else ▷ $\textsc{Context}(a) = \textsc{free}$
            then $\textsc{ScheduleFreeAction}(a, LCA)$
```

graph as possible. Figure 4.4 shows a situation where this strategy implies non-efficient data-transfer strategies. Data elements are copied redundantly. By scheduling the copy actions around surrounding loops, the number of invocations of the copy actions reduces. Allocation and free actions are to be scheduled accordingly, such that memory is allocated when a related copy-in action starts and memory is not freed before corresponding copy-out actions are completed.

The highest level in the extended call graph where a data copy action can be scheduled is defined by the *Least-Common Ancestor* of the indicated GPU kernel and the operations that reside in $Operations$ (Figure 4.6). The Least-Common Ancestor (LCA) is the deepest node in the extended call graph that shares both the GPU kernel and the operations of $Operations$.

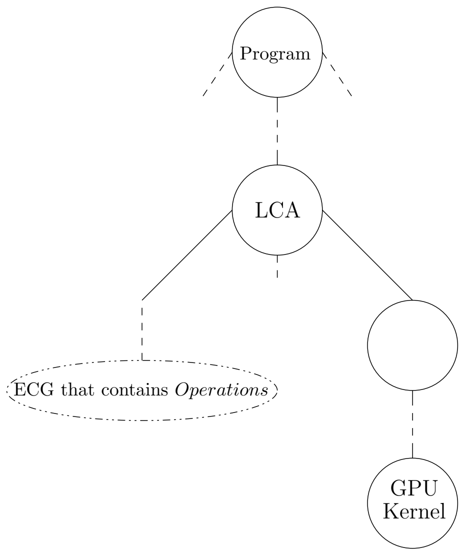
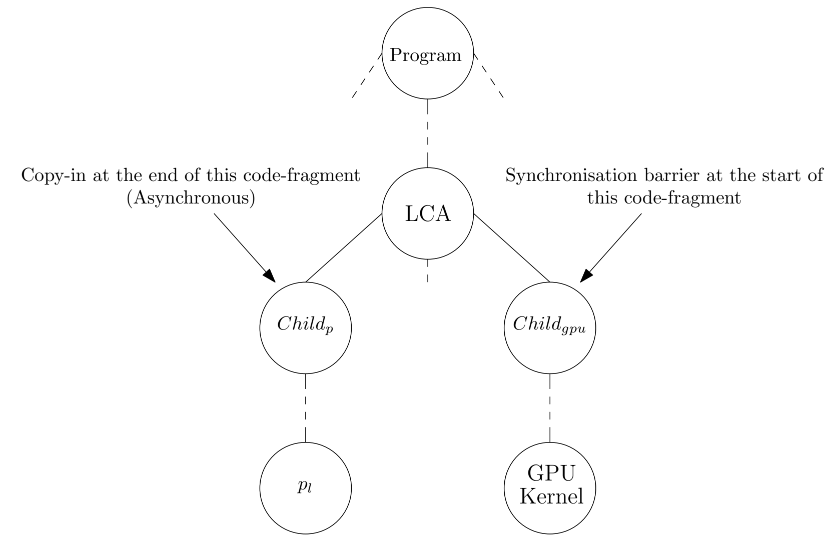


Figure 4.6: Least Common Ancestor of a GPU kernel and a set of operations ($Operations$)

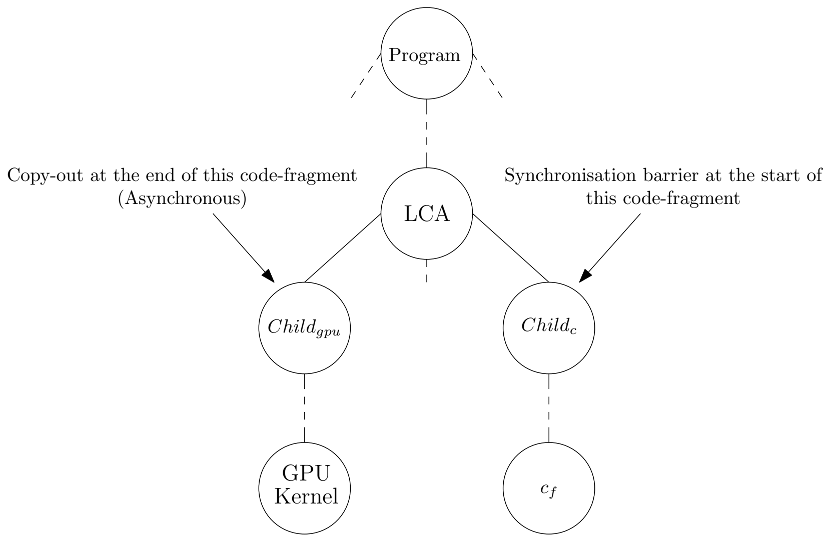
Schedule the action (line 13) In the code fragment that represents the LCA, the algorithm defines a period p in the code where a copy action takes place. At the start of p, the copy action is started asynchronously to the host and the accelerator. At the end of p, the copy action must be finished. A synchronisation barrier is inserted to synchronise the copy action with the host. This period is maximised in length to get maximal overlap with computations on the host and the GPU.

The procedure of scheduling copy-in actions is defined by InsertCopyIn. Figure 4.7(a) gives a schematic overview of the period where a copy-in action takes place. A copy-in action a is started right after the code fragment, that is represented by the *LCA's child node* $Child_p$, is ended. A LCA's child node is a node in the extended call graph that is a direct child of the LCA. The LCA's child node is invoked directly from the LCA. $Child_p$ contains the last producing operation p_l that is related to a. After p_l was invoked for the last time (i.e. after $Child_p$), all related data-elements are ready to be copied from the host to the GPU. At this moment in the code, the related copy-in action is started.

Since the copy-in action is launched asynchronously, the tool inserts a synchronisation barrier just before the LCA's child node $Child_{gpu}$ to synchronise the copy-in action with the host. After this moment, all data-elements that are related to a reside on the GPU's memory.


Program
Copy-in at the end of this code-fragment (Asynchronous)
LCA
Synchronisation barrier at the start of this code-fragment
$Child_p$
$Child_{gpu}$
p_l
GPU Kernel

(a) Copy-in action


Program
Copy-out at the end of this code-fragment (Asynchronous)
LCA
Synchronisation barrier at the start of this code-fragment
$Child_{gpu}$
$Child_c$
GPU Kernel
c_f

(b) Copy-out action

Figure 4.7: Scheduling of copy actions in the extended call graph

Copy-out actions a (that are related to a specific kernel k) are scheduled similarly. Figure 4.7(b) gives a schematic overview of the period where a copy-out action takes place. The copy-out action is started right after the end of the LCA's child node that contains the GPU kernel $Child_{gpu}$. Since GPU kernels are launched asynchronously to the host and CPU-GPU copy actions can only be started from the host, a synchronisation barrier is scheduled before the copy-out action to make sure that the final invocation of kernel k has been finished. After this moment, all related data-elements are ready to be transferred. The copy-out action must be finished just before the start of the LCA's child node $Child_c$. $Child_c$ is the node in the extended call graph that contains the first consuming operation c_f of the data-elements that are involved in a. Therefore, a synchronisation barrier is scheduled before $Child_c$ to ensure that a is finished before consumption on the host. The scheduling process of a copy-out action and its corresponding synchronisation barriers is performed with InsertCopyOut.

Scheduling allocation (InsertAllocation) and free (InsertFree) actions for data-structures that are used on the GPU take a different approach. These actions cannot be performed asynchronously from the host code. Allocation and free actions on the GPU take little time. We observed that allocation and free actions generally take less then 5 ms, independent of the memory size that is being allocated. The tool identifies a window in the ECG where allocation and free actions can take place. This window is depicted in Figure 4.8.

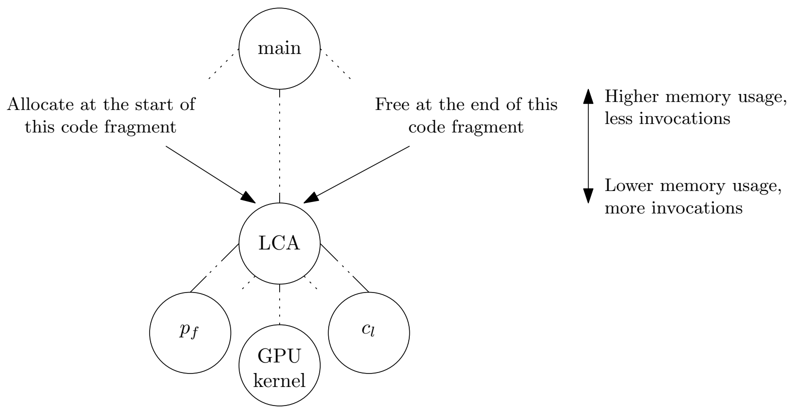

main
Allocate at the start of this code fragment
Free at the end of this code fragment
Higher memory usage, less invocations
Lower memory usage, more invocations
LCA
p_f
GPU kernel
c_l

Figure 4.8: Scheduling of allocation & free actions

The effect of scheduling the allocation and free actions at the extremes of the window is explained next. Allocation and free actions can be scheduled at the highest level in the order tree. This implies that the allocation actions are done at the start of the program and free actions are done at the end of the program. This causes the smallest number of invocations. However, memory is allocated over a long period of time. Therefore, the GPU's limited memory space is used inefficiently. Memory is allocated while not being used by the GPU.

By scheduling allocation and free actions lower in the tree, i.e. deeper in the program, the GPU's memory space is utilised more efficiently. The time where data-elements are allocated but not used on the GPU decreases. The lowest level in the extended call graph, where allocation and free actions can be scheduled, is indicated by the least-common-ancestor of the first producing operation p_l and the last consuming operation c_l. p_f and c_l indicate the maximal life-span of the data-structure that has to be allocated or freed. Scheduling allocation and free actions at this level causes the largest number of invocations of the action (surrounding loops cause redundant invocations of the allocation and free actions). However, memory space is used most efficiently. Scheduling allocation and free actions at deeper level causes incorrect behaviour, since copy actions will then be started before the allocation and will end after the free action.

In the remainder of this thesis, we schedule allocation and free actions at the deepest possible level in the extended call graph; at the position as indicated in Figure 4.8.

By executing this algorithm, the actions that are stated in Table A.2 are translated to the transfer and allocation strategy as stated in Listing A.2.

Size of copy & allocation actions In both the CUDA and OpenACC programming models, all copy and allocation actions have to be annotated by the range of data-elements that need to be copied. As discussed in Section 2.3.4, the tools of Vector Fabrics return information about the set of data-elements $data_{x,d_a}$ that is accessed within a data-structure x for a dependency d_a. The set of data elements that has to be copied or allocated for an action a is defined by (4.6).

$$DATA_a = \bigcup_{d_a \in a} data_{d_a} \tag{4.6}$$

Out of $DATA_a$, the tool annotates each copy and allocation action by the smallest continuous memory-segment that contains all data-elements in $DATA_a$.

Chapter 5

GPU kernel optimisations

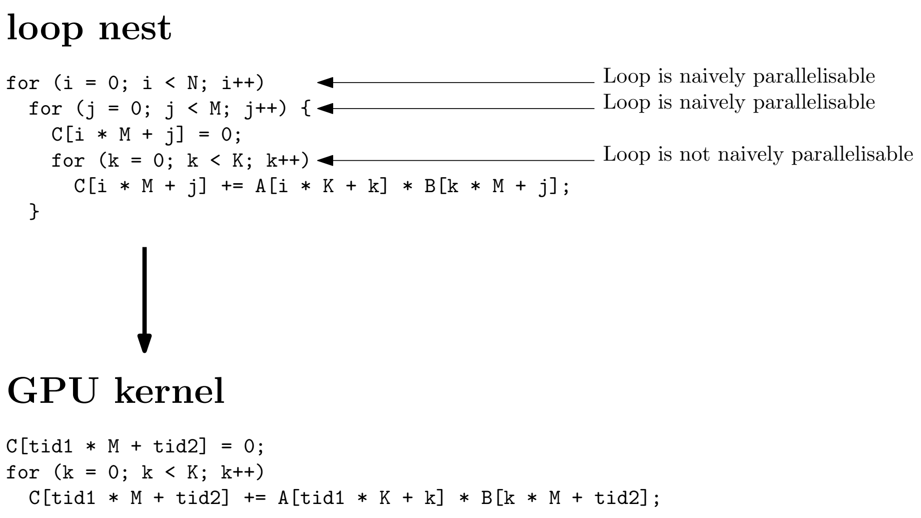
In this chapter, an approach is discussed to optimise the data-access strategies in GPU kernels. As stated by Van den Braak [15], Yang [13] and Ryoo [16], the majority of GPU kernels is bounded in performance by its memory access pattern. By employing inefficient memory access patterns in the GPU kernel, the available memory access bandwidth to the GPU's advanced memory hierarchy is not fully utilised. Therefore, the overall performance is significantly suppressed.

We have designed tools that focus on optimising the following two aspects.

- **Coalescing** The first tool searches for a *coalescing optimal thread configuration* of a GPU kernel. In a coalescing optimal thread configuration, the GPU can apply a maximal level of coalescing to the kernel. However, not all memory accesses are coalesced by applying such thread configuration. Therefore, we apply a code transformation to improve the coalescing capabilities of the GPU.
- **Data-sharing** The second tool detects reuse of data among threads in a thread block. By using this information, the on-chip scratchpad memory can be exploited to cache data that is shared among multiple threads in a thread block.

With these tools, a programmer or automatic transformation framework can improve the performance of GPU kernels by utilising the GPU's memory hierarchy more efficiently. The tools focus on optimising memory access patterns that are described by linear functions in terms of their surrounding loops and the GPU's thread configuration. The memory access functions are observed from the program's execution trace (Section 5.1). As stated by Baskaran et al. [6], the majority of memory access patterns that occur in real life applications is described by a linear function.

The GPU kernels that are considered in this chapter are naively parallelised GPU kernels. The GPU kernels are generated from *embarrassingly parallel loop nests*. Loops $l_0, \ldots, l_n$ in an embarrassingly parallel loop nest L can be transformed into a GPU kernel without additional code transformations other then transforming each iteration of the loop nest into a separate thread. Every consecutive iteration of a parallelised loop is executed by a consecutive group of threads. An example of the parallelisation process of a matrix multiplication function (which is an embarrassingly parallel loop nest) can be seen in Figure 5.1. Figure 5.2 shows the generated thread configuration for this kernel. The mapping $M = \{l_0 \mapsto dim_0, \ldots, l_n \mapsto dim_n\}$ for all loops $l_0, \ldots, l_n \in L$ is called a *loop mapping*. A loop mapping states for every loop, that is parallelised over separate threads, the dimension of the GPU's thread configuration over which the loop is parallelised. Whenever we mention *CPU's kernel equivalent* in this chapter, we indicate the sequentially executed CPU's equivalent of the GPU kernel.


loop nest
Loop is naively parallelisable
Loop is naively parallelisable
Loop is not naively parallelisable
GPU kernel

```
for (i = 0; i < N; i++)
  for (j = 0; j < M; j++) {
    C[i * M + j] = 0;
    for (k = 0; k < K; k++)
      C[i * M + j] += A[i * K + k] * B[k * M + j];
  }
```

```
C[tid1 * M + tid2] = 0;
for (k = 0; k < K; k++)
  C[tid1 * M + tid2] += A[tid1 * K + k] * B[k * M + tid2];
```

Figure 5.1: Naive parallelisation of matrix multiplication. `tid1` and `tid2` identify the thread in the GPU's thread configuration.

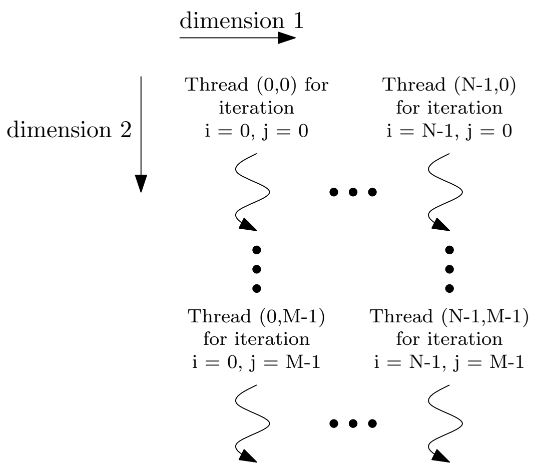

dimension 1
dimension 2
Thread (0,0) for iteration i = 0, j = 0
Thread (N-1,0) for iteration i = N-1, j = 0
Thread (0,M-1) for iteration i = 0, j = M-1
Thread (N-1,M-1) for iteration i = N-1, j = M-1

Figure 5.2: Generated thread configuration for the matrix multiplication kernel. Instantiation of the two dimensions is defined in the loop mapping.

The remainder of this chapter is organised as follows. Section 5.1 discusses how the linear access functions of memory read operations are generated. In Section 5.2 we focus on optimising the GPU kernel for coalescing. Finally, Section 5.3 discusses the tool to detect sharing among threads in a thread block.

5.1 Extraction of linear access patterns

This section describes an analysis tool to determine the access function of a memory operation. This access function is required for the optimisations as proposed in the next sections of this chapter.

The tool extracts the access function of a memory operation x in terms of the CPU's kernel equivalent. Given the iteration number of each surrounding loop in the GPU kernel and the thread number, this function determines the data-element that is accessed for that invocation of x.

In what follows, we only focus on recognising access functions that are linear in terms of the CPU's kernel equivalent. The loops are indicated by $l_1, \ldots, l_u$. Equation (5.1) describes the general format of the access function, where $iter_{(x,i)}$ describes the iteration number of loop i and $C_{(x,i)}$ describes the coefficient of loop i.

$$f_x(iter_{(x,1)}, \ldots, iter_{(x,u)}) = C_{(x,1)} \times iter_{(x,1)} + \ldots + C_{(x,u)} \times iter_{(x,u)} \tag{5.1}$$

Since we use execution traces for analysis, we can only prove that linearity holds for the observed program traces. As stated in Section 2.3.2, it is a programmers best practice to have a large set of test cases that can be used as an input for analysis. However, these test-cases may not cover all situations in which the memory operation is executed. The memory operation can expose different (non-linear) behaviour in situations that are not observed during analysis. For these unobserved situations, the optimisation steps as described in Sections 5.2.1, 5.2.2 and 5.3 may not improve the performance of the application to the fullest extend.

The tool observes a stream of observations S_x (5.2) for a given memory operation x. Each of the observations $s_{(x,i)}$ is a tuple as shown in (5.3). An observation consists of an *iteration vector* $IV_{(x,i)}$ and the data-element that is accessed $elt_{(x,i)}$. An iteration vector (5.4) indicates for all u surrounding loops the specific iteration number in which the observation occurred.

$$S_x = s_{(x,1)}, \ldots, s_{(x,n)} \tag{5.2}$$

$$s_{(x,i)} = IV_{(x,i)} \times elt_{(x,i)} \tag{5.3}$$

$$IV_{(x,i)} = iter_{(x,1,i)}), \ldots, iter_{(x,u,i)} \tag{5.4}$$

The tool searches for all coefficients $C_{(x,j)}$ $(0 \leq j \leq u)$ in $f_x(IV_{(x,i)})$ (5.1), such that for every observation $s_{(x,i)} = IV_{(x,i)} \times elt_{(d,x,i)}$ it holds that $f_x(IV_{(x,i)}) = elt_{(d,x,i)}$; the access function holds for all observations.

The problem of extracting all $C_{(x,j)}$'s in (5.1) boils down to the problem of solving a set of linear equations to an integer solution. Most approaches on solving this problem are offline approaches where a set of equations should be stored first in order to solve the system at once. These approaches are not suitable for dynamic analysis, due to the large number of samples that are observed for a memory operation. The memory requirement is too large.

The tool uses an online approach that is based on the well known subtraction method to solve a set of linear equations. During processing the set of observations, the tool keeps track of the set of coefficients that is already extracted by observing previous observations ($Cs_{(x,generated)}$) and an equation eq_x that describes the linear combination of all coefficients that are not yet found after observing the previous observations. By processing new observations, the tool learns about unsolved $C_{(x,j)}$'s that are described in eq_x. Furthermore, the coefficients in $Cs_{(x,generated)}$ are checked for validity in the new observation. By using the subtraction method, memory usage remains minimal.

After analysing all observations, the tool outputs either one of the following three cases for the analysis of memory operation x.

- The tool did not observe enough samples to generate all coefficients $C_{(x,j)}$ of the linear memory access pattern. For at least one surrounding loop $loop_{(x,j)}$ its corresponding $C_{(x,j)}$ is unknown. Therefore, there is not enough information to check if the memory access pattern is linear or non-linear.
- The tool observed that the access pattern is non-linear (i.e. it holds that $f_x(IV_{(x,i)}) \neq elt_{d,x,i}$ for at least one observation $s_{(x,i)} = IV_{(x,i)} \times elt_{(d,x,i)}$ where $C_{(x,0)}, \ldots, C_{(x,u)} \in Cs_{(x,generated)}$).
- The tool observed that the access pattern is linear for the given set of observations (i.e., for all observations $s_{(x,i)} = IV_{(x,i)} \times elt_{(d,x,i)}$ it holds that $f_x(IV_{(x,i)}) = elt_{d,x,i}$). The tool outputs all coefficients $C_{(x,j)} \in Cs_{(x,generated)}$.

5.2 Coalesced memory access

This section focusses on improving a GPU kernel for coalescing. As discussed in Section 2.1.1, a GPU uses coalescing to group memory accesses to the off-chip memory of a warp (a group of neighbouring threads in the X-dimension of the GPU's thread configuration) into larger memory transactions. In this way, bandwidth utilisation increases. In order to exploit these coalescing capabilities to the fullest extent (i.e. the number issued memory transactions is minimal), data should be accessed in a specific way. Consider for example a GPU kernel, where each thread performs computations on a two dimensional array. This situation is depicted in Figure 5.3. Three access patterns for a warp can be identified.

1. When threads in a warp access data elements in a row-wise fashion, coalescing capabilities are fully exploited. All data-elements that are retrieved in a memory transaction are used for computation. This case is depicted as (a) in Figure 5.3.
2. All threads in a warp access the same data-element. Only one memory transaction is issued, however not all data-elements that are retrieved are used in the warp. This case is depicted as (b) in Figure 5.3.
3. When threads in a warp access data elements in a column-wise fashion, a separate memory transaction is performed for every thread in the warp. This access pattern is the most inefficient in terms of coalescing. This case is depicted as (c) in Figure 5.3.

Other access patterns are less common, but their coalescing performance will be between the first and the third case.

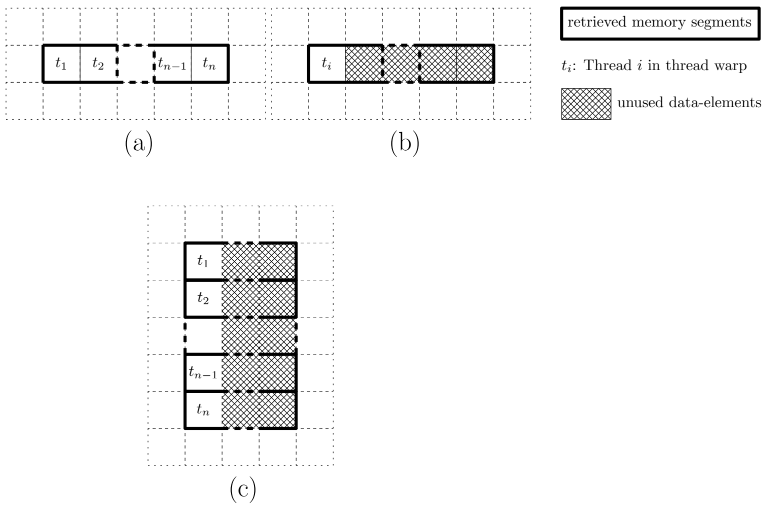


Figure 5.3: Access patterns determine level of achieved coalescing.

In this section an approach is described that optimises a GPU kernel for coalescing. This approach consists of two steps. In the first step a loop mapping is generated such that the level of coalescing is maximised. In the second step a code transformation is applied to optimise the memory read operations that are not coalesced under a given thread configuration.

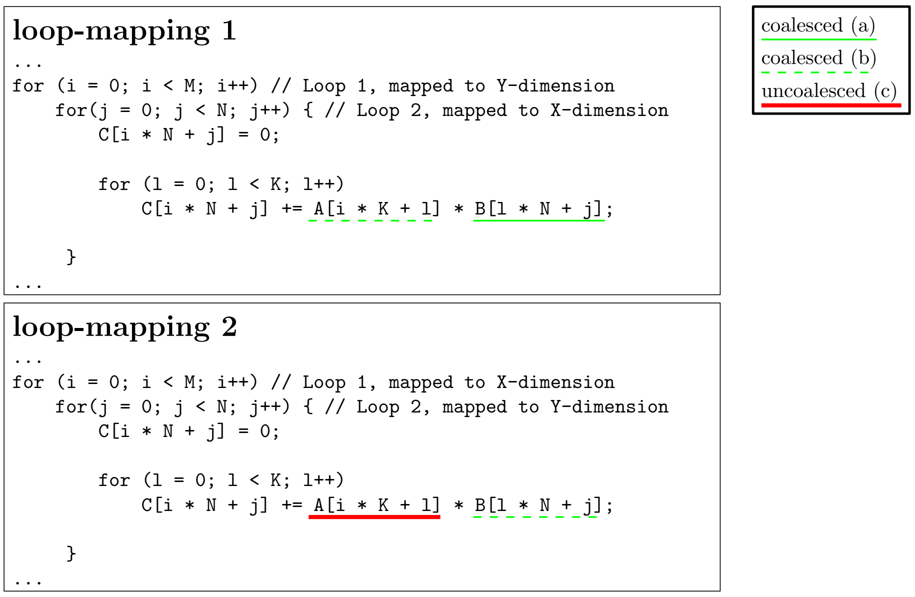
In order to reason about coalescing for a certain memory operation, the analysis uses the access functions as observed in the analysis of Section 5.1. Therefore, the analysis can only reason about linear access patterns.

5.2.1 Coalescing optimal thread-configuration

Coalescing performance varies over every loop mapping that is possible in generating a GPU kernel from a loop nest. As can be seen from the results of the benchmarks (Section 6.2) mapping loops over the wrong dimensions of the GPU's thread configuration degrades overall performance significantly. The main cause of this performance degradation is the lack of coalescing possibilities. The tool that is described in this section generates a loop mapping so that the coalescing capabilities of the GPU are maximised in the naively generated kernel.

Figure 5.4 gives an example of how a loop mapping effects coalescing in a matrix multiplication function. The GPU can take maximal advantage of the coalescing capabilities for memory read operation `B[l * N + j]` when loop 2 is mapped to the X-dimension of the GPU's thread configuration in loop-mapping 1. In this loop mapping, memory read operation `A[i * K + l]` is coalesced in accordance to case (b) of Figure 5.3. For loop mapping 2, memory read operation `A[i * K + l]` is uncoalesced, while memory read operation `B[l * N + j]` is coalesced in accordance to case (b) of Figure 5.3.

The tool takes a CPU's kernel equivalent loop nest for input. The set of loops that has to be parallelised in separate threads is indicated by L. The tool retrieves all memory read operations


coalesced (a)
coalesced (b)
uncoalesced (c)

loop-mapping 1

```
...
for (i = 0; i < M; i++) // Loop 1, mapped to Y-dimension
    for(j = 0; j < N; j++) { // Loop 2, mapped to X-dimension
        C[i * N + j] = 0;

        for (l = 0; l < K; l++)
            C[i * N + j] += A[i * K + l] * B[l * N + j];

     }
...
```

loop-mapping 2

```
...
for (i = 0; i < M; i++) // Loop 1, mapped to X-dimension
    for(j = 0; j < N; j++) { // Loop 2, mapped to Y-dimension
        C[i * N + j] = 0;

        for (l = 0; l < K; l++)
            C[i * N + j] += A[i * K + l] * B[l * N + j];

     }
...
```

Figure 5.4: Loop mapping effects coalescing capabilities. Cases (a), (b) and (c) relate to the cases of Figure 5.3.

$R_L = \{r_1, \ldots, r_n\}$ that are located inside the GPU kernel. Next, an exhaustive search is performed over all possible loop mappings $\mathfrak{M}$ that can be generated from L. During this search, the analysis searches for the loop mapping $M_{max} \in \mathfrak{M}$ that results in the highest overall *coalescing score* C_M (5.5).

An overall coalescing score is an indicator of the level of coalescing that can be applied with a given loop mapping. It is a summation of the coalescing factors $coalfactor_{(r,M)}$ of all memory read operations $r \in R_L$ multiplied by the number of iterations $iters_{l_x}$ of the loop l_x that is mapped to the X-dimension of the thread configuration. Warps are groups of threads in the X-dimension of the GPU kernel. The number of iterations in l_x relates directly to the number of warps that are produced.

The coalescing factor gives an indication of the number of memory transactions that has to be issued to serve all threads in a warp. Since we only consider linear access functions, we only need to calculate the coalescing factor for two general cases. In this description, $retr\_bytes$ represents the number of bytes that is retrieved for every invocation of r. $C_{(r,l_x)}$ represents the coefficient of l_x in the linear access function of r.

- In the first case it holds that $C_{(r,l_x)} \neq 0$. The coalescing factor is described by Equation (5.6). When $|C_{(r,l_x)}| - retr\_bytes$ reduces, the number of unused bytes in a memory transaction reduces and less memory transactions are issued to serve all threads in a warp. Therefore, the level of coalescing increases.
- In the second case it holds that $C_{(r,l_x)} = 0$. If this holds, only one memory segment will be retrieved from the off-chip memory. The coalescing factor is maximal.

The tool returns the loop mapping that has the highest overall coalescing score C_M.

$$C_M = iters_{l_x} \times \sum_{r \in R_L} coalfactor_{(r,M)} \tag{5.5}$$

$$coalfactor_{(r,M)} = \frac{1}{|C_{(r,l_x)}| - retr\_bytes} \qquad \text{if } C_{(r,l_x)} \neq 0 \tag{5.6}$$

$$coalfactor_{(r,M)} = \infty \qquad \text{if } C_{(r,l_x)} = 0 \tag{5.7}$$

5.2.2 Further coalescing optimisations

The loop mapping M_{max}, as proposed by the tool of Section 5.2.1, does not always imply maximal coalescing performance. Data access patterns may vary over the individual memory operations such that a GPU can not apply coalescing techniques to every memory read operations by a given loop mapping. In the coalescing optimal loop mapping of the matrix multiplication application (loop mapping 1 of Figure 5.4), the GPU can apply coalescing techniques to memory read operation `B[l * N + j]`. The other memory read operation `A[i * K + l]` is also coalesced, but the memory segment that is retrieved for a warp is not fully used by the warp. Neighbouring data elements are retrieved across multiple invocations of the warp, but are only used for computations in one invocation. Data elements are retrieved redundantly across multiple invocations of memory read `A[i * K + l]`, while this can be avoided. The tool that is presented in this section applies a code transformation to increase the coalescing capabilities of the GPU by making use of the on-chip scratchpad memory. That is, access patterns as shown in cases (b) and (c) of Figure 5.3 are transformed to get a access pattern as shown in case (a) of Figure 5.3. The code transformation is schematically sketched in Figure 5.5. Next, we discuss when the code transformation can be applied. After that, we discuss the code transformation in more detail.

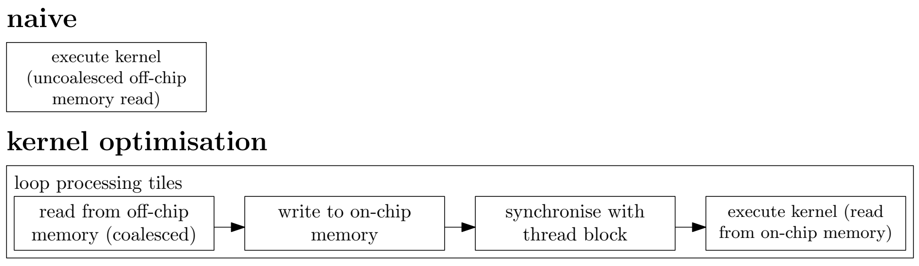

naive
execute kernel (uncoalesced off-chip memory read)
kernel optimisation
loop processing tiles
read from off-chip memory (coalesced)
write to on-chip memory
synchronise with thread block
execute kernel (read from on-chip memory)

Figure 5.5: Program flow after applying the coalescing optimisation

By applying this optimisation, the level of coalescing is increased and the memory segments as retrieved for a warp are utilised more efficiently (i.e. more data-elements are used during computation). However, this comes at the costs of two extra memory accesses (read and write) to the shared memory and a synchronisation buffer, per processed tile of data. Therefore, the optimisation is only applied when the level of coalescing is maximal after the application of the code transformation. That is when warps access a continuous memory segment. Only for the memory operations that satisfy this condition it is experimentally checked that the benefits of coalescing outweigh the additional costs. A memory read operation for which a warp retrieves a continuous memory segment is called a fully coalesced memory read operation.

Only those memory read operations are considered for this code transformation that are classified to have a access pattern as shown in cases (a) and (b) of Figure 5.3. This set of memory read operations is indicated by R_{nc}. For each memory read operation $r_{nc} \in R_{nc}$, it is checked if this operation can be converted into a fully coalesced operation. It should hold that $|C_{(r_{nc},l_d)}| = retr\_bytes$, where $C_{(r_{nc},l_d)}$ represents the coefficient of the deepest surrounding loop l_d in the memory access function of r_{nc}. $retr\_bytes$ represents the number of bytes that is retrieved for every invocation of r_{nc}.

To all memory read operations r that satisfy the previous condition the following optimisation is applied. The step size of the deepest surrounding loop l_d of r_{nc} is multiplied by the thread block size in the X-dimension. Within this loop, two phases are defined. In the first phase, data elements are loaded, in a coalesced optimal way, from global memory to a buffer that is located on shared memory. After the data has been loaded, a synchronisation step is done over all threads in the thread block to make sure that all data is available in shared memory whenever the actual computation starts. After the data is loaded, an extra loop is executed that contains the original kernel. In the kernel, r_{nc} is changed to load from the shared memory buffer. The transformation of a matrix multiplication kernel can be observed in Listings 5.1 and 5.2. $tidx$ and $tidy$ are the coordinates of the thread in the thread configuration. $tbidx$ represents the coordinate of the thread in the thread block in the X-dimension of the thread configuration.

Listing 5.1: Naive matrix multiplication kernel

```
C[tidy * M + tidx] = 0;
for (k = 0; k < K; k++)
  C[tidy * M + tidx] += A[tidy * K + k] * B[k * M + tidx];
```

Listing 5.2: Coalescing optimised matrix multiplication kernel

```
C[tidy * M + tidx] = 0;
for (k = 0; k < K; k += TB_size_x) {
  __shared__ float shared0[TB_size_x];
  shared0[tbidx] = A[tidy * K + k + tbidx];
  __synchthreads();
  for (l = 0; l < TB_size_x; l++)
    C[tidy * M + tidx] = shared0[l] * B[(k + l) * M + tidx];
  __syncthreads();
}
```

5.3 Reuse through shared memory

Another set of optimisations that can be applied to a GPU kernel is related to caching frequently used data-elements in the on-chip shared memory. A GPU is equipped with a software managed scratchpad memory, the so-called shared memory. The shared memory is located on-chip, therefore access latencies and available communication bandwidths are much better in comparison to the off-chip memory. By caching data-elements that are used multiple times across different threads in a thread block, the number of memory transactions to the off-chip memory can be reduced.

An application that benefits from caching is an image filtering kernel where neighbouring threads use the same data elements. The situation is depicted in Figure 5.6.

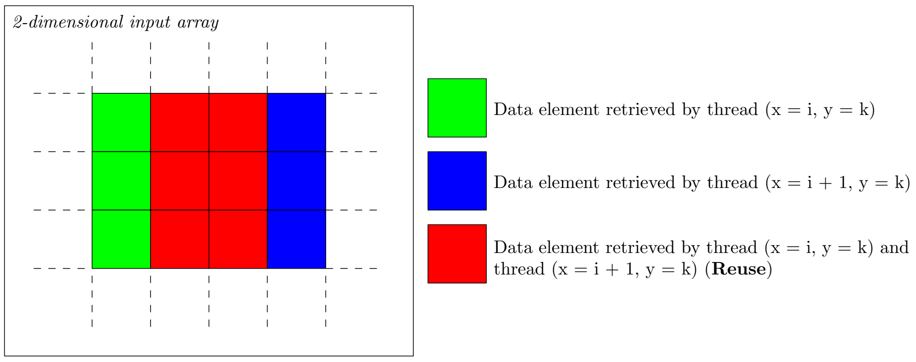

2-dimensional input array
Data element retrieved by thread (x = i, y = k)
Data element retrieved by thread (x = i + 1, y = k)
Data element retrieved by thread (x = i, y = k) and thread (x = i + 1, y = k) (Reuse)

Figure 5.6: Overlapping data regions in an image filtering kernel (3×3 window)

As was discussed in Section 2.1.1, newer architectures of NVidia GPUs are equipped with hardware managed caches that reduce the need for this caching optimisations. However, due to the freedom of the warp schedulers on the GPU's SMs, this hardware managed cache may not perform as good as expected. By using scratchpad memories, software can manage which data elements are loaded in the cache. In this way, cache misses, as occurred with a hardware managed cache, are avoided.

In this section, we propose a tool to detect caching possibilities. The tool uses the access function of a memory operation as was generated in Section 5.1.

The tool generates a reuse mapping, stating for every invocation (described by the thread identification (t_x, t_y, t_z) of the thread within a thread block and an iteration vector $iter_{(r,1)}, \ldots, iter_{(r,n)}$ of all loops in a thread) of a memory read operation r, the first invocation of r in the thread block that retrieves the same data element. The mapping is defined by equation (5.8).

$$(t_x, t_y, t_z) \times \{iter_{(r,1)}, \ldots, iter_{(r,n)}\} \mapsto (t'_x, t'_y, t'_z) \times \{iter'_{(r,1)}, \ldots, iter'_{(r,n)}\} \tag{5.8}$$

Invocations of a memory operation r within a thread are ordered according to a lexicographical ordering over its surrounding loops $loop_1, \ldots, loop_n$. Invocations of r that are done in a previous iteration have a lower lexicographical ordering. This ordering is called the intra-thread ordering. We also define a lexicographical ordering across threads in a thread block. This is called the inter-thread ordering. That is, the x-dimension of the GPU thread configuration has higher priority in ordering then the y- and z-dimension. The y-dimension of the thread configuration has higher priority in ordering then the z-dimension. We define a lexicographical ordering among all invocations of r in a thread block where the inter-thread ordering has a higher priority in ordering then the intra-thread ordering. This is called the thread block ordering.

The tool uses the access function, as extracted with the tool from Section 5.1. There is reuse for a memory read operation r in a sequentially executed loop nest $loop_1, \ldots, loop_n$ if there are at least two different, valid (within loop bounds) iteration vectors iv_1 and iv_2, that differ from each other for two loops i and j, that access the same data element. This implies that there are at least two loops (loops i and j) in the surrounding loop nest of r that have different coefficients C_i and C_j in the access function of r. Equation (5.9) holds if there is reuse. Based on this information, the tool finds the reuse mapping.

$$\begin{aligned}
f(iv_1) = f(iv_2)\wedge \\
iv_1 = iter_{(r,1)}, \ldots, iter_{(r,i)}, \ldots, iter_{(r,j)}, \ldots, iter_{(r,n)}\wedge \\
iv_2 = iter_{(r,1)}, \ldots, iter'_{(r,i)}, \ldots, iter'_{(r,j)}, \ldots, iter_{(r,n)}\wedge \\
iter_{(r,i)} \neq iter'_{(r,i)}\wedge \\
iter_{(r,j)} \neq iter'_{(r,j)}\wedge \\
i \neq j
\end{aligned} \tag{5.9}$$

The tool determines the reuse mapping in two steps. In the first step, the *intra-thread* reuse mapping is produced. This information is used in the second step, to produce the reuse mapping over the thread block. The intra-thread reuse, is the reuse of data within a certain thread. That is, multiple invocations in one thread access the same data element due to the structure of surrounding loops. Both steps are explained next.

In the first step, the intra-thread reuse mapping is generated. For every invocation of r (described by the iteration vector $iter_{(r,1)}, \ldots, iter_{(r,n)}$) inside a thread, the tool recursively checks all possible pairs of loops i and j if the related data element is loaded in a previous invocation of r in accordance to the conditions of (5.9). The recursive process for each invocation of r continues until the first intra-thread invocation has been found that retrieves the same data element.

With the information that was produced in the first step, the reuse mapping is generated over all invocations of r across all threads in the thread block. Like with the intra-thread reuse mapping, pairs of loops are recursively considered to find the first invocation in the thread block that retrieves the same data element. Two types of pairs are considered now.

- The first element represents the coefficient of a loop in the CPU's kernel equivalent that is mapped to one of the dimensions of the GPU's thread configuration. The second element represents the coefficient of a loop that is executed within a thread.
- Both elements represent coefficients of loops in the CPU's kernel equivalent that are mapped to dimensions of the GPU's thread configuration.

In a similar way as with the generation of the intra-thread reuse mapping, each pair of coefficients is evaluated recursively to generate the final reuse mapping. After each recursive step, the intra-thread reuse mapping is used, to find the first invocation within the thread that is currently being evaluated.

With the generated information, the code transformation as schematically shown in Figure 5.7 can be applied to cache reused data elements in shared memory for a memory read operation r. This code transformation has to be applied manually. The automatic code transformation is subject to future work.

In the first step, all data-elements that are required for the execution of a thread block are retrieved in a non-redundant way. That is, every data-element that is required in a thread block is retrieved from the off-chip memory by one thread in the thread block. The data is stored in the on-chip scratchpad memory. In order to coordinate the work over all threads in the thread block, the

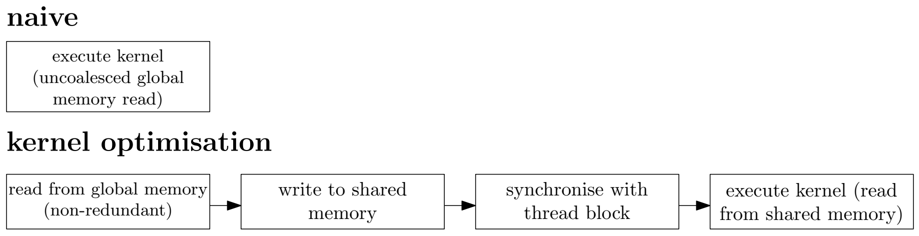

naive
execute kernel (uncoalesced global memory read)
kernel optimisation
read from global memory (non-redundant)
write to shared memory
synchronise with thread block
execute kernel (read from shared memory)

Figure 5.7: Program flow after applying the reuse through shared memory optimisation

mapping as produced by the tool is used. By using a condition, only those invocations that are in the co-domain of the generated mapping are allowed to read from the off-chip memory and store in the on-chip memory. After a synchronisation barrier, the actual kernel is executed. Instead of reading from global memory, the shared memory buffer is addressed.

Figure 5.7 shows that this optimisation comes at the costs of an additional read and write operation to the shared memory and idling time of the threads due to the synchronisation barrier. Since data is buffered on the GPU's on-chip memory, the occupancy factor for the kernel on the GPU decreases, which increases kernel execution time. Therefore, this optimisation is not always beneficial. A metric is required to detect the *reuse level* for the memory operation r; the number of times a data-element is used in a thread block. In the generated mapping, a data element is represented by its corresponding invocation. The reuse level of a data-element is calculated by counting the number of occurrences of its corresponding invocation in the codomain of the generated mapping. An average reuse level among all data elements that are loaded in a thread block is calculated by dividing the number of elements in the domain of the inter thread mapping by the number of unique elements in the codomain of this mapping.

Chapter 6

Performance evaluation

In this chapter, three real-life applications (series of matrix multiplications, n-body simulation and image embossing) are elaborated to show the performance impact of applying the optimisation steps as described in Chapters 4 and 5.

The workflow in optimising the GPUGPU accelerated program is as follows:

1 Naive GPU kernels are generated from indicated code fragments, according to the parallelisation process as described in the introduction of Chapter 5.

2 The naive GPU kernel is optimised with the tools that are described in Chapter 5.

   2.1 Determine a coalescing optimal thread configuration (Section 5.2.1)

   2.2 ∗ Improve coalescing with code transformation (Section 5.2.2)

   ∗ Use shared memory for caching data (Section 5.3)

3 An efficient copy and allocation strategy is generated for the GPGPU accelerated application (Chapter 4).

The optimisations of step 2.2 are applied separately. The combination of both optimisations are subject to future work.

For each step, the execution time is measured. That is, for the optimisations of step 2, the kernel execution time is measured to observe the impact of the kernel optimisations. For the optimised copy and allocation strategy (step 3), we determine the speedup of the complete application, compared to a naive allocation and transfer strategy. In this naive strategy, data-structures that are read in the kernel are transferred to the kernel before the start of the kernel and all produced data-structures are copied to the host after each kernel execution. In step 3, we use the fastest GPU kernel, as was generated by step 2. The applications are compiled by CUDA version 5.0 under default settings of the compiler.

The benchmarks are executed on two systems.

- **system 1** desktop computer with an Intel Core i7-3770 processor and a NVidia GT-640 GPU (Kepler architecture) with 1 gigabyte of memory. The GPU and CPU are connected through a PCI-express 3.0 bus.

- **system 2** desktop computer with an Intel Core 2 Duo E7300 processor and a NVidia GTS-250 GPU (GT200 architecture) with 1 gigabyte of memory. The GPU and CPU are connected through a PCI-express 2.0 bus.

The L1 data cache of the NVidia GT-640 GPU is configured in 16 kB hardware managed data cache and 48 kB shared memory. The NVidia GTS-250 GPU cannot be configured with a hardware managed L1 cache.

The remainder of this chapter is organised as follows. Section 6.1 introduces the analysed applications, their corresponding extended call graph (representing the program's execution flow) and the optimisations that are applied to them. Section 6.2 discusses the resulting performance.

6.1 Applications

6.1.1 Series of matrix multiplication

The first application that is optimised is a series of matrix multiplication kernels. This application was discussed as a running example through Chapter 4.

Figure 1.1 has shown that the application benefits from the reduction in the number of copy and allocation actions. By scheduling the copy actions in parallel to host and CPU code, the performance can be increased even more. In order to emphasise the effect of the generated copy and allocation strategy over a naive strategy, the application as explained in Chapter 4 is surrounded by a loop that executes the series of matrix multiplications multiple times. For each iteration of the loop, matrices `A`, `B`, `C` and `D` are produced on the host. In the next step, three matrix multiplication kernels are executed on the GPU. The produced arrays (`E`, `F` and `G`) are consumed on the host by a function `totalSum`. Figure 6.1 shows the extended call graph (ECG) for this program. By applying the tool of Chapter 4, the data transfer and allocation strategy is generated as shown in Figure 6.2.

The matrix multiplication kernel itself is considered to be bound in performance by the utilised memory bandwidth [16]. Therefore, the kernel benefits from the code optimisations as proposed in Chapter 5.

The matrix multiplication kernel is a GPU kernel that is executed by a two-dimensional thread configuration. Each thread calculates the value of one element in the two-dimensional output matrix. Therefore, there are only two choices in mapping the loops over the thread configuration of the GPU. As was discussed in Section 5.2.2, one thread configuration should result in lower execution times since this configuration gives better coalescing possibilities. As was shown in Section 5.2.2, the coalescing performance of the matrix multiplication kernel can be improved by applying a code transformation to increase the coalescing capabilities for the other memory read operation.

As a next step, data-reuse is determined within thread blocks. The tool of Section 5.3 observes that neighbouring threads (in one of the GPU's thread dimensions) access the same rows and columns. Given this information, the tool suggests to cache data in shared memory. However, the shared memory is too small to store all data-elements that can be reused in a thread-block at once. Therefore a kernel is generated that uses a tiling mechanism. Tiles of data are loaded

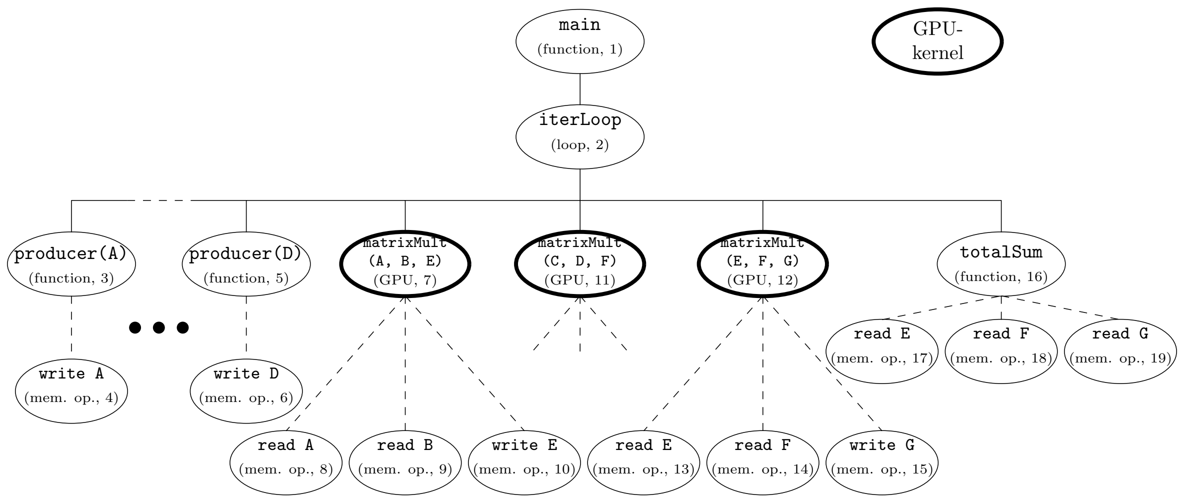


Figure 6.1: Relevant parts of the extended call graph for the matrix multiplication application. Each node represents a code fragment and is indicated by the type of the code fragment and the key in the order tree. `matrixMult(C, D, F)` has the same structure as `matrixMult(A, B, E)` and `matrixMult(E, F, G)`.

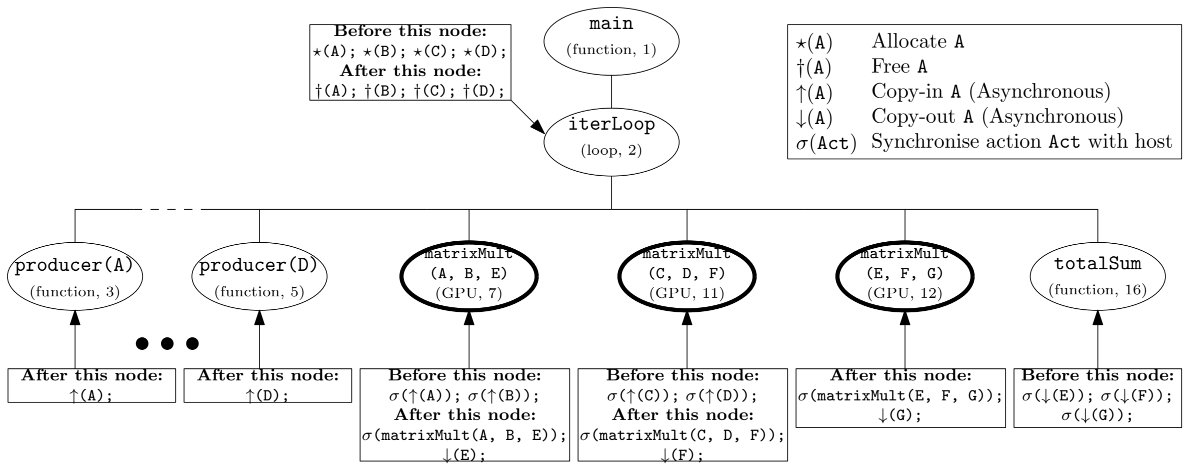


Figure 6.2: The generated copy and allocation strategy for the series of matrix multiplication application of Figure 6.1.

in the scratchpad memory. Once loaded, the matrix multiplication algorithm continues with the data that is buffered.

6.1.2 N-body simulation

The second application that is optimised is n-body simulation. For a set of particles (*bodies*), the algorithm calculates a force (`force`), a velocity (`vel`) and position (`pos`) over time.

The structure of the application is as follows. A procedure can be distinguished that sets initial data (setInitData). After the data is set the calculate procedure is called where all calculations are done. calculate contains a loop (timeLoop) that dictates the time steps for the integration of the velocity and position over time. Within this loop, the actual calculations are done. Two kernels exist in this loop. In the first kernel (gravKernel), the gravitation is calculated over the set of bodies. In the second kernel (intrKernel), the calculated forces are integrated over time and the new velocity and position of the particles is calculated. Figure 6.3 shows the extended call graph of the application.

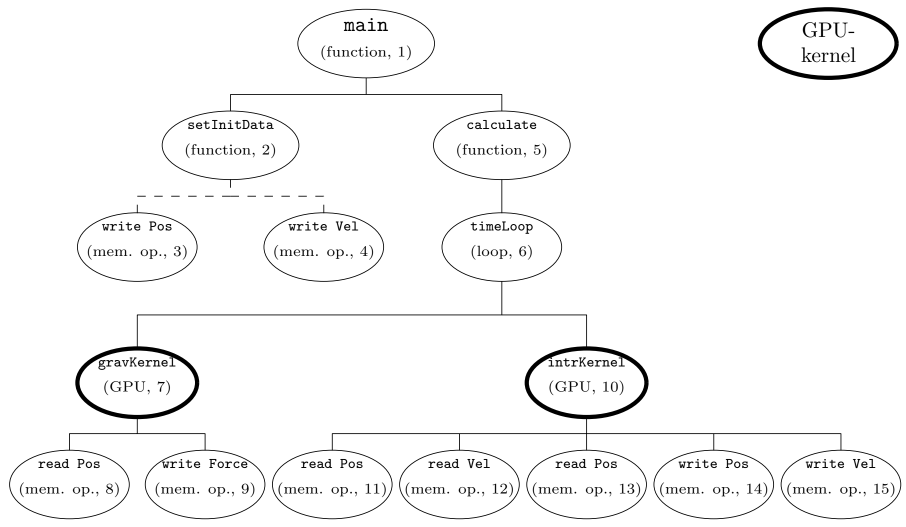


Figure 6.3: Relevant parts of the extended call graph for the N-body simulation. Each node represents a code fragment and is indicated by the type of the code fragment and the key in the order tree.

The tool schedules the allocation and free actions for pos and vel at the start respectively end of main, since main is the least common ancestor of the dependencies that relate to these data-structures. Similarly force is allocated and freed around timeLoop. The copy-in actions for the data structures (pos and gel) as set by setInitData are started after setInitData has ended. A synchronisation barrier is scheduled before timeLoop, to make sure that all data is available when the GPU kernels are started. In this way, no redundant copy and synchronisation actions are executed. The resulting copy and allocation strategy is shown in Figure 6.4.

The GPU kernel optimisations as discussed in Chapter 5 can not be applied to the kernels in the n-body simulation.

6.1.3 Image embossing

The final application that is optimised is image embossing. Image embossing is an image filtering operation where each pixel of an image is replaced either by a highlight or a shadow, depending on the light/dark boundaries on the original image. Low contrast areas are replaced by a grey background. The application processes a series of input images (processLoop). For each iteration of this loop the following steps are performed.

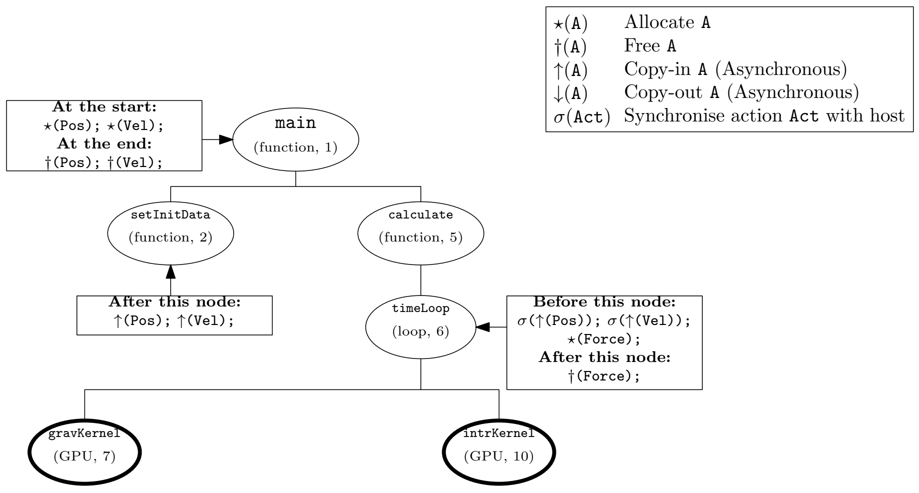

⋆(A) Allocate A
†(A) Free A
↑(A) Copy-in A (Asynchronous)
↓(A) Copy-out A (Asynchronous)
σ(Act) Synchronise action Act with host
At the start: ⋆(Pos); ⋆(Vel); At the end: †(Pos); †(Vel);
main (function, 1)
setInitData (function, 2)
calculate (function, 5)
After this node: ↑(Pos); ↑(Vel);
timeLoop (loop, 6)
Before this node: σ(↑(Pos)); σ(↑(Vel)); ⋆(Force); After this node: †(Force);
gravKernel (GPU, 7)
intrKernel (GPU, 10)

Figure 6.4: The generated copy and allocation strategy for the N-body simulation of Figure 6.3.

1. the input image is read (`readInImg`)
2. the output image is create (`createOutImg`)
3. the filter is applied over the complete image by `processImg`
4. the resulting image is being written (`writeOutImg`).

The application of the filter to the image (*processImg*) is performed by a GPU kernel.

Figure 6.5 shows the extended call graph for this application.

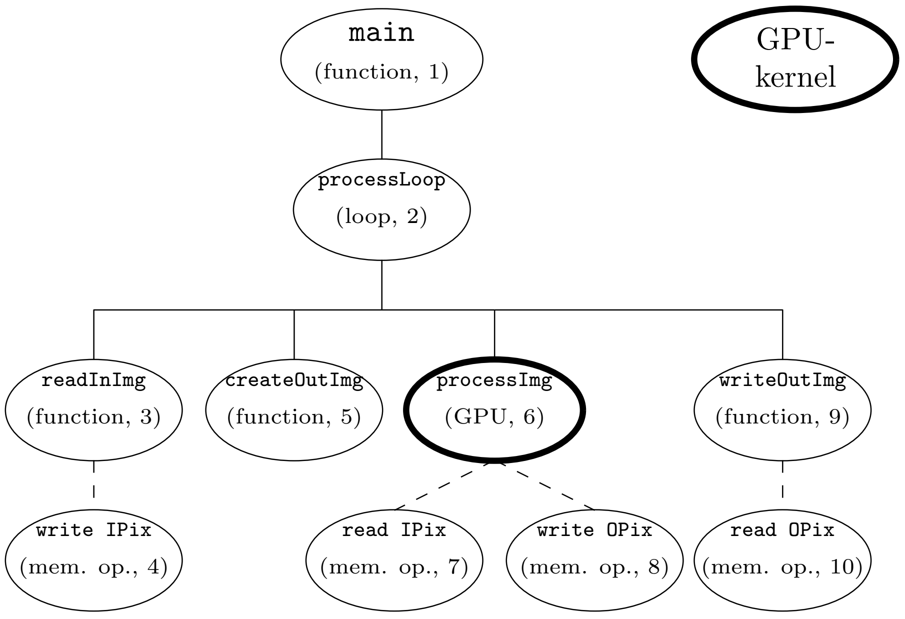
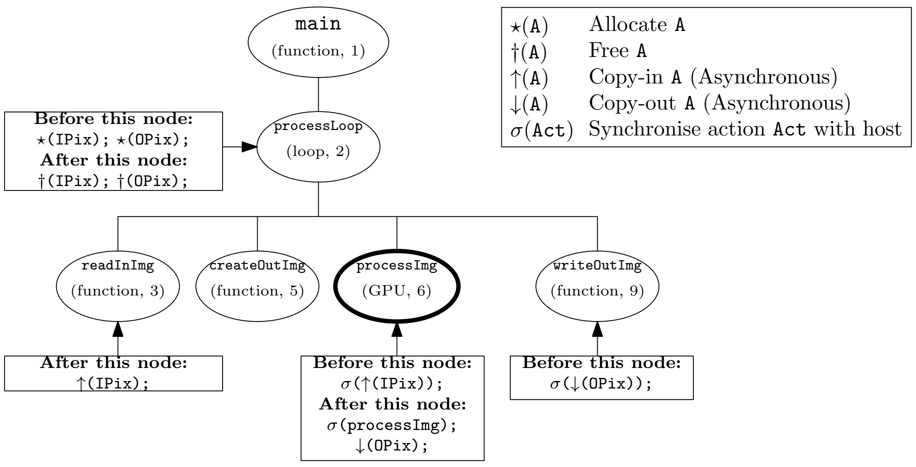


Figure 6.5: Relevant parts of the extended call graph for the image embossing application. Each node represents a code fragment and is indicated by the type of the code fragment and the key in the order tree.

The memory space that is required for the GPU kernels (`IPix` and `OPix`) is allocated before the start of `processLoop`. It is freed at the end of `processLoop`. The copy-in action of the input image is started at the end of `readInImg`. The copy-in action should be ended at the start of the GPU kernel. The output pixels are transferred from the GPU to the host at the end of the GPU kernel. A synchronisation barrier is scheduled at the start of `writeOutImg` to ensure that all pixels are transferred from the GPU to the host. This copy and allocation strategy is summarised in Figure 6.6.



main
(function, 1)
processLoop
(loop, 2)
readInImg
(function, 3)
createOutImg
(function, 5)
processImg
(GPU, 6)
writeOutImg
(function, 9)
Before this node: ⋆(IPix); ⋆(OPix); After this node: †(IPix); †(OPix);
After this node: ↑(IPix);
Before this node: σ(↑(IPix)); After this node: σ(processImg); ↓(OPix);
Before this node: σ(↓(OPix));
⋆(A) Allocate A
†(A) Free A
↑(A) Copy-in A (Asynchronous)
↓(A) Copy-out A (Asynchronous)
σ(Act) Synchronise action Act with host

Figure 6.6: The generated copy and allocation strategy for the image embossing application of Figure 6.5.

Since the reuse of data-elements (i.e., pixels are reused between neighbouring threads) is high, data caching on the on-chip memory is considered. Unlike in the matrix multiplication kernel, all data-elements as required by a thread block fit on the on-chip memory. No tiling techniques need to be applied.

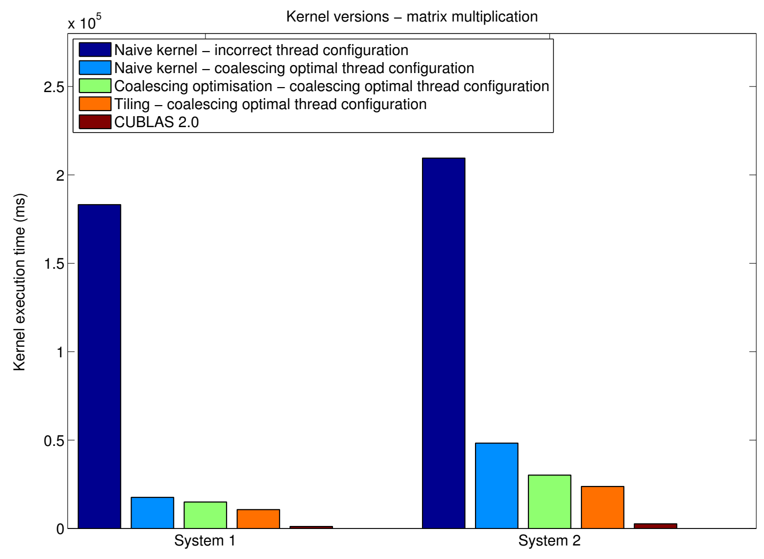
6.2 Performance

6.2.1 Kernel optimisations

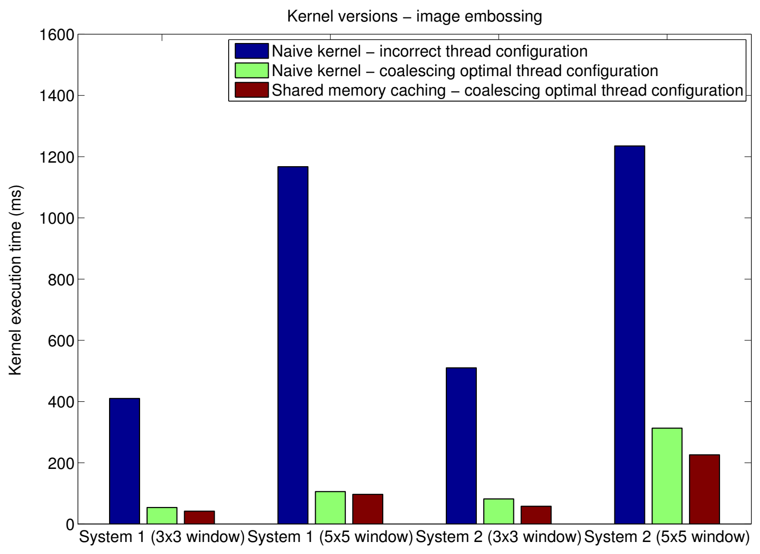
Figure 6.7 shows the effect of the proposed kernel optimisations of Chapter 5 on the set of benchmarks. For the matrix multiplication, we compared the performance of our kernels to the performance of the matrix multiplication kernel as implemented in the CUBLAS 2.0 library [17].

Figure 6.7 shows that coalescing has a large impact on the kernel execution times. By choosing the right thread configuration, execution times can be reduced significantly, without applying any code transformations.

The effect of the coalescing optimising code transformation is different between both systems. The effect of the optimisation is larger for system 2 (NVidia GTS-250) then for system 1 (NVidia GT-640). This is because of the hardware managed cache that is implemented on the NVidia GT-640. Therefore, the number of memory accesses to the off-chip memory reduces.



(a)



(b)

Figure 6.7: Kernel execution times after applying the kernel optimisations

Considering the caching optimisation where data is explicitly cached on the GPU's on-chip shared memory, we observed that the optimisation has a larger effect on system 2 then on system 1. That is because of the on-chip L1 data cache that is implemented in the GT-640. This cache already reduces the number of accesses to the off-chip memory, without explicitly caching on the on-chip memory. Therefore, the average latencies in accessing off-chip memory reduces. It is surprising that using the shared memory in system 1 for the image embossing application, reduces the kernel execution time compared to a naive kernel. Due to the spatial locality of the algorithm, we expected that the on-chip L1 cache gives high hit rate. Using the shared memory for caching implies an extra synchronisation barrier in the thread block. Therefore, execution times should increase. As is shown in our results, this is however not the case. Using the shared memory implies a reduction in the number of accesses to the off-chip memory. We suspect that in the naive kernel cache misses occur in the L1 data cache due to freedom of the warp schedulers to schedule warps in arbitrarily over time. By using the shared memory, we enforce an optimal caching behaviour. Observing the exact behaviour of the cache is a subject for future work.

CUBLAS 2.0 matrix multiplication We compared the matrix multiplication kernel with NVidia's matrix multiplication kernel as is implemented in the CUBLAS 2.0 library. The resulting performance of the CUBLAS implementation is around 10x better compared to our implementation. This can be explained as follows.

The CUBLAS implementation uses a tiling approach as is similar to our kernel. However, a set of additional optimisations is performed to increase the performance. The first two optimisations are considered as algorithmic optimisations. The multiply-add operation (of element `A[i]` and `B[i]`) is translated into two separate operations in the assembly code. The current NVidia architecture only allows one source operand from shared memory. Therefore, the compiler splits the multiply-add operation into a separate load to `A[i]` and a multiply-add instruction that uses the result of `A[i]`. By storing the tile of array `B` in shared memory, one can calculate the outer product. In this way, the multiply-add instruction can be translated into one operation in the assembly code. The second optimisation is loop unrolling which decreases execution time. We consider such optimisations as algorithmic, therefore our tools do not optimise for this.

Furthermore, the CUBLAS implementation is optimised to prevent bank conflicts (Section 2.1.1). As was seen in section 6.1.1, data-elements are reused among threads. Since these data-elements are stored on the on-chip shared memory and warps access the same data-elements simultaneously, bank conflicts occur. This optimisation is subject to future work.

Finally, the CUBLAS implementation is optimised to get the highest occupancy factor (Section 2.1.3). This is a subject for future work.

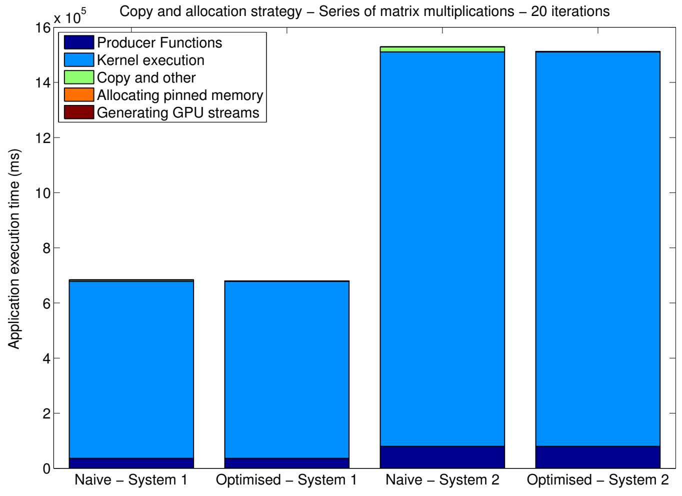
6.2.2 Copy and allocation strategies

Figure 6.8 shows the resulting performance of an optimised copy and allocation strategy compared to a naive strategy. In these applications, we used the fastest GPU kernels as generated by our tools of Chapter 5. As can be seen from these results, the effect of applying an optimised copy and allocation strategy is minimal for matrix-multiplication and N-body simulation. The effect of an optimised copy and allocation strategy is less then a percent. The computational bandwidth of the GPU kernels is too small compared to the bandwidth of the system bus. However, most of the copy actions between CPU and GPU can be hidden with computations on the host and GPU.

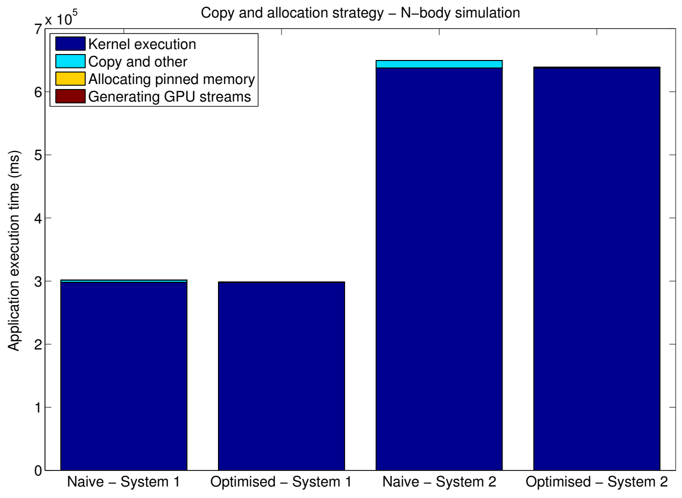
For the image embossing application, the overall execution time increases. The effect of doing the data-transfers asynchronously to the host and GPU has almost no effect since the data transfers do not overlap with computations. Doing the data transfer asynchronously has a negative effect on the performance since allocating streams takes significant time. The time spent in copying data is reduced since the optimised data transfer strategy uses pinned memory on the host, which implied higher transfer bandwidths between the host and the GPU (Figure 4.7). However, the improved performance vanishes since allocating pinned memory takes significantly more time compared to allocating pageable memory. The allocating time is constant in the application since this is done outside the code fragment that is represented by `processLoop` in Figure 6.6. When the number of processed images increases, using pinned memory becomes beneficial.

The effect of optimising the copy and allocation strategy increases for the matrix multiplication application, when we use the highly optimised CUBLAS implementation. This result is shown in Figure 6.9. The CUBLAS kernel is around ten times faster compared to the kernel as generated by our tools, therefore the effect of an optimised copy and allocation strategy increases. By optimising the copy and allocation strategy, we can reduce the overall execution time for the CUBLAS implementation with ten percent. The effect is larger for system 2 then for system 1. The performance gap between the system bus and the GPU computational power is larger compared to system 1.

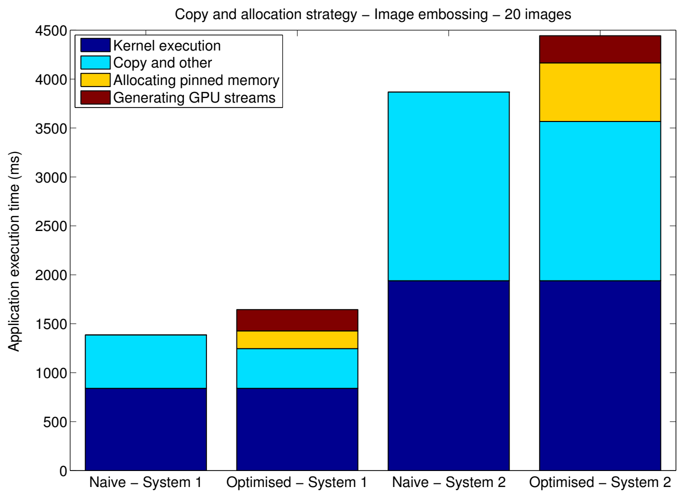
Figure 6.10 focusses on the time spent in copying and allocating data for the matrix multiplication application using CUBLAS. As can be seen from this Figure, substantial time is spent in generating the streams, so copy actions can take place asynchronously. However, the total costs of generating these streams is constant over different number of iterations of the `iterLoop`. These streams are generated once at the start of the program.



(a)



(b)



(c)

Figure 6.8: Application execution times after optimising the copy and allocation strategies.

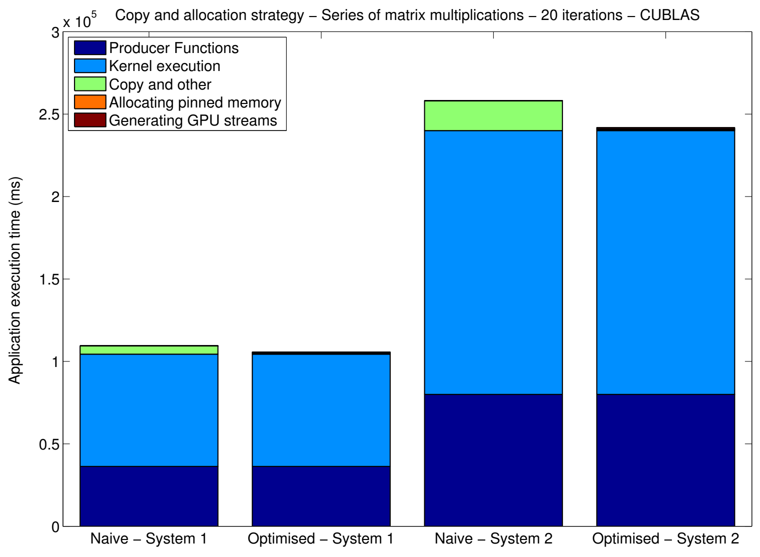


Figure 6.9: Copy and allocation strategy - Matrix multiplication application - CUBLAS

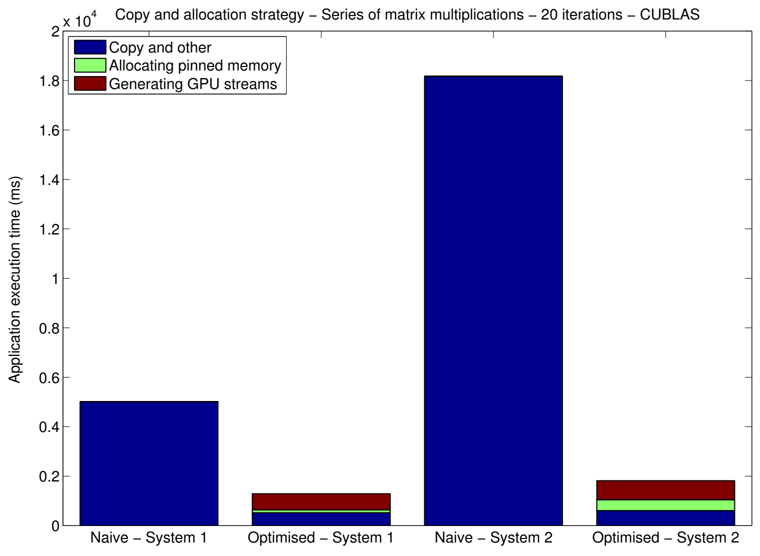


Figure 6.10: Copy and allocation strategy - Matrix multiplication application - CUBLAS - Focus on copy and allocation

Chapter 7

Future work

The tools that are proposed in this thesis can aid programmers in porting a sequential program to a GPGPU accelerated program. Furthermore, they can be used as a basis for an automatic transformation framework to transform a sequential CPU program into a GPU accelerated program. However, the tools have a set of limitations that should be addressed in future work. This chapter focusses on these limitations.

CPU-GPU copy and GPU allocation strategy Regarding the generation of CPU-GPU copy and GPU allocation strategies, the tool groups copy actions for different elements in a data-structure to transfer it all at once. As discussed in Chapter 4, copy actions are scheduled at the highest possible level in the extended call graph. Each copy action transfers continuous memory segment of all related data-elements (Section 4.3). By scheduling copy actions in this way, the largest communication bandwidth between CPU and GPU is achieved (Figure 4.7). However, this approach may not always be beneficial. Too many data-elements that are copied, are not used by consuming operations. Time is wasted on transferring useless data-elements. A better approach would be to split up the copy actions in this case. An example situation is depicted in Figure 7.1. In this extreme case, only the first and last element of X are used by the consuming operations. Other data-elements are copied, but remain unused. It may be better to split the copy actions such that only the consumed data-elements are transferred. Less bytes are transferred but since each copy action has a smaller size, communication bandwidths decrease. This is subject to future work.

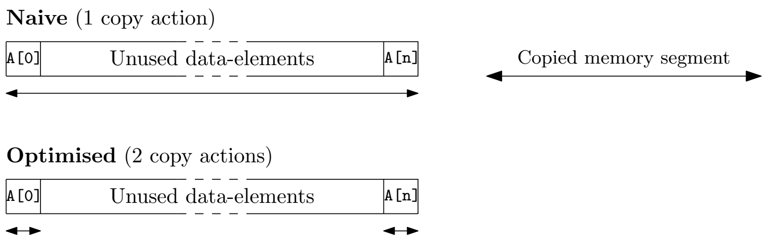


Figure 7.1: Comparing performance of multiple smaller copy actions and 1 grouped copy action

Furthermore, tiling techniques [18] can be considered. With tiling (Figure 7.2), one invocation of a kernel X is split over multiple invocations. Each of the kernel invocations will work on a subset of data-elements from the original kernel invocation X. In parallel to each kernel invocation, data is being transferred between the host and the GPU. Memory transfers overlap with computations on both the host and the GPU. Tiling may be considered when the copy actions (as generated by the tool in this thesis) are not overlapped with either host or accelerator code. Applying tiling comes at the cost of extra synchronisation barriers and lower bandwidth utilisation (since each copy action will transfer less bytes).

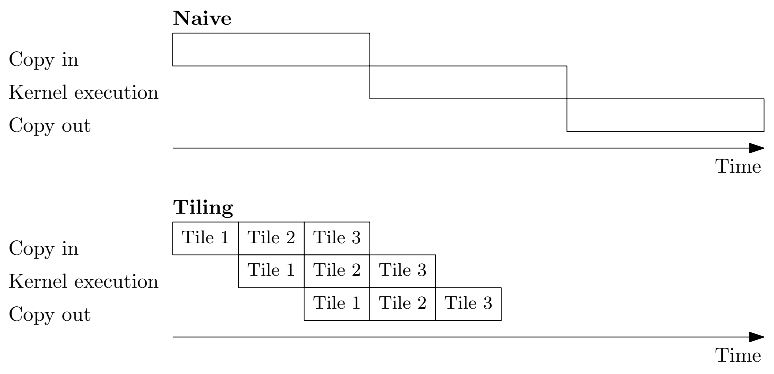


Figure 7.2: Tiling techniques to overlap memory transfers with computation

GPU kernel optimisations In relation to caching of data on on-chip memory buffers, the tools only consider memory read operations individually. This implies that only kernels can be processed where memory operations access data-structures, that are stored on the off-chip memory, only once per thread; that is, there is no kernel internal read-after-write dependency related to that data-structure. The majority of GPU kernels exposes this behaviour. By using the dependency analysis of Vector Fabrics, it is a trivial extension to support GPU kernels with kernel internal dependencies.

The tools that are proposed in this thesis do not take the limited size of the shared memory into account. A situation where this caused a problem was observed for the reuse analysis of the series of matrix multiplication application (Section 6.1.1). All data that was needed to serve a thread block did not fit on the GPU's shared memory, therefore we had to implement a tiling mechanism inside the GPU kernel. A topic of future work that can be considered is the automatic application of these tiling techniques inside GPU kernels. A thread block processes tiles of data elements in a loop. These tiles are loaded into the on-chip scratchpad memory to reduce access latencies. This is an extension of the tiling techniques that are considered to optimise the CPU-GPU communication pattern.

Another issue that is still open, is the reduced occupancy (Section 2.1.3) of the GPU when the kernel optimisations are implemented. Since a thread block will use more resources (shared memory), less thread blocks can be fitted on the GPU for concurrent execution. Therefore, less data-transfers are overlapped with computations, and the application execution time will increase.

Chapter 8

Conclusions

GPUs provide large computational power, due to the availability of a large set of parallel processing units. Data-parallel functions can be executed significantly faster on a GPU then on a CPU. Therefore, there is a lot of interest in porting sequentially executed functions to multithreaded GPU kernels. However, the porting process is time consuming and non-trivial, especially for programmers who are no expert in the field of GPU programming. Therefore the need arises for frameworks that can aid programmers in the porting process or are even capable to generate GPU accelerated programs automatically.

This thesis presented a set of analysis tools that can aid programmers or automatic transformation frameworks to transform a CPU program to a GPGPU accelerated program. Two issues are identified that can significantly suppress overall performance of a GPGPU accelerated program: inefficient data-transfer strategies and inefficient usage of the GPU's advanced memory hierarchy. The tools that are presented in this thesis focus on optimising both aspects. The four main contributions of this thesis are methods to:

- generate efficient copy and allocation strategies in GPGPU-programming (Chapter 4);
- observe linear memory access patterns from program execution traces (Section 5.1);
- map loops over the GPU's thread configuration, such that the level of memory coalescing is maximal (Section 5.2.1). Whenever memory operations are still uncoalesced, a code transformation is applied to optimise these operations individually for coalescing (Section 5.2.2);
- discover reuse of data within thread blocks. This enables a code transformation to cache data on the GPU's on-chip scratchpad memory (Section 5.3)

Although this thesis focusses on GPGPU programming, the optimisations that are proposed in this thesis are applicable to a wider range of accelerators. The generation of efficient copy and allocation strategies is relevant for every accelerator that is physically separate from the host device. The coalescing analysis can be applied to every computational unit that uses coalescing techniques to improve memory bandwidth utilisation. The reuse analysis is relevant to every accelerator that uses an advanced hierarchy of caches and explicitly managed scratchpad memories.

The tools use a dynamic analysis framework (as provided by Vector Fabrics) to obtain information about the analysed application. However, most of the tools as discussed in this thesis, can also be based on static analysis. For the generation of optimal copy and allocation strategies, any framework that reveals dependencies in a program suffices. The kernel optimisations use the observed memory access functions as input for analysis. We use dynamic analysis to reconstruct this function from observed samples. However, the memory access function can also be extracted with static analysis techniques like symbol analysis.

The tools are evaluated for a set of applications. As the reader can observe from Chapter 6, the effect of optimising the copy and allocation strategy is maximal when multiple GPU kernels are executed. For a matrix multiplication application where a series of matrix multiplication kernels is executed, we show that the overhead, as caused by memory transfers, is almost completely removed. The impact on the overall speedup of the application is related to the throughput of the GPU kernel. We show for a matrix multiplication application that the overall execution time can be reduced with around 5% when the highly optimised CUBLAS implementation of matrix multiplication is used. By using a more naive implementation of the matrix multiplication kernel, the performance benefit is less then 1%.

Coalescing has a significant impact on the generated kernels. Searching for a thread configuration that provides the largest amount of coalescing is always beneficial and does not imply any additional costs during execution. By picking the right coalescing optimal thread configuration, the execution time of a kernel can be reduced more then 90%. By applying the proposed code transformation, the coalescing capabilities of a kernel can be improved even further.

Caching frequently used data in the GPU's on-chip scratchpad memory can reduce kernel execution times since less accesses to the GPU's off-chip memory need to be performed. In the case that the GPU is equipped with a hardware managed L1 data cache the effect of this optimisation reduces but still has a positive effect. The hardware managed cache cannot perform as a cache where its behaviour is specified explicitly in software.

The tools that are discussed in this thesis help programmers and automatic transformation frameworks to generate high performance GPGPU accelerated programs. The issues that are addressed in this thesis have a large impact on the execution time of the program but are non-trivial and hard to observe by programmers and state-of-the-art automatic transformation frameworks.

Bibliography

[1] L.A. Barroso. The price of performance. *Queue*, 3(7):48–53, 2005.

[2] Nvidia. CUDA C programming guide, 2011.

[3] NVIDIA CAPS Enterprise, Cray Inc. and the Portland Group. The OpenACC Application Programming Interface, Nov 2011.

[4] A. Munshi. The opencl specification, 2009.

[5] F. Nielson, H.R. Nielson, and C. Hankin. *Principles of program analysis*. Springer, 2004.

[6] M. Baskaran, J. Ramanujam, and P. Sadayappan. Automatic C-to-CUDA code generation for affine programs. In *Compiler Construction*, pages 244–263. Springer, 2010.

[7] H.G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, pages 358–366, 1953.

[8] B.G. Ryder. Constructing the call graph of a program. *Software Engineering, IEEE Transactions on*, (3):216–226, 1979.

[9] P.J.J.M. Custers. Algorithmic species: Classifying program code for parallel computing. Master's thesis, Eindhoven University of Technology, 2012.

[10] M. Amini, F. Coelho, F. Irigoin, and R. Keryell. Static compilation analysis for host-accelerator communication optimization. *Int. Work. Lang. and Compilers for Par. Computing (LCPC)*, 2011.

[11] T.B. Jablin, J.A. Jablin, P. Prabhu, F. Liu, and D.I. August. Dynamically managed data for cpu-gpu architectures. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, pages 165–174. ACM, 2012.

[12] T.B. Jablin, P. Prabhu, J.A. Jablin, N.P. Johnson, S.R. Beard, and D.I. August. Automatic cpu-gpu communication management and optimization. In *ACM SIGPLAN Notices*, volume 46, pages 142–151. ACM, 2011.

[13] Y. Yang, P. Xiang, J. Kong, and H. Zhou. A gpgpu compiler for memory optimization and parallelism management. In *ACM Sigplan Notices*, volume 45, pages 86–97. ACM, 2010.

[14] C. Juravle. Automatic program analysis for data parallel kernels. Master's thesis, Utrecht University, 2011.

[15] G.J. van den Braak, B. Mesman, and H. Corporaal. Compile-time gpu memory access

optimizations. In *Embedded Computer Systems (SAMOS), 2010 International Conference on, vol., no*, pages 200–207, 2010.

[16] S. Ryoo, C.I. Rodrigues, S.S. Baghsorkhi, S.S. Stone, D.B. Kirk, and W.W. Hwu. 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*, pages 73–82. ACM, 2008.

[17] V. Volkov and J.W. Demmel. Benchmarking gpus to tune dense linear algebra. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, page 31. IEEE Press, 2008.

[18] P. Visschers. Reducing the overhead of data transfer in data-parallel programs. Master's thesis, Utrecht University, 2011.

Appendix A

Data-transfer strategy in matrix multiplication application

Listing A.1: Naive data-transfer and allocation strategy generated for Listing 4.1

```
void main(void)
{
  ...
  for (i = 0; i < N; i++) {
    randomize_array(A);
    randomize_array(B);
    randomize_array(C);
    randomize_array(D);

    // matrix_mult(A, B, E)
    allocate_GPU(A, N * size)
    allocate_GPU(B, N * size)
    allocate_GPU(E, N * size)
    copy_in_GPU(A, N * size)
    copy_in_GPU(B, N * size)
    matrix_mult_GPU(A, B, E)
    GPU_synchronise(matrix_mult_GPU(A, B, E))
    copy_out_GPU(E, N * size)
    free_GPU(A)
    free_GPU(B)
    free_GPU(E)

    // matrix_mult(C, D, F)
    allocate_GPU(C, N * size)
    allocate_GPU(D, N * size)
    allocate_GPU(F, N * size)
    copy_in_GPU(C, N * size)
    copy_in_GPU(D, N * size)
    matrix_mult_GPU(C, D, F)
    GPU_synchronise(matrix_mult_GPU(C, D, F))
    copy_out_GPU(F, N * size)
    free_GPU(C)
    free_GPU(D)
    free_GPU(F)

    // matrix_mult(E, F, G)
    allocate_GPU(E, N * size)
    allocate_GPU(F, N * size)
    allocate_GPU(G, N * size)
    copy_in_GPU(E, N * size)
    copy_in_GPU(F, N * size)
    matrix_mult_GPU(E, F, G)
    GPU_synchronise(matrix_mult_GPU(E, F, G))
    copy_out_GPU(G, N * size)
    free_GPU(E)
    free_GPU(F)
    free_GPU(G)

    total_sum(E, F, G);
  }
  ...
}
```

Listing A.2: Optimised data-transfer and allocation strategy generated for Listing 4.1

```
void main(void)
{
  allocate_GPU(A, N * size)
  allocate_GPU(B, N * size)
  allocate_GPU(C, N * size)
  allocate_GPU(D, N * size)
  allocate_GPU(E, N * size)
  allocate_GPU(F, N * size)
  allocate_GPU(G, N * size)
  ...
  for (i = 0; i < N; i++) {
    randomize_array(A);
    copy_in_GPU_async(A, N * size)
    randomize_array(B);
    copy_in_GPU_async(B, N * size)
    randomize_array(C);
    copy_in_GPU_async(C, N * size)
    randomize_array(D);
    copy_in_GPU_async(D, N * size)

    // matrix_mult(A, B, E)
    GPU_synchronise(ALL EVENTS)
    matrix_mult_GPU(A, B, E)
    GPU_synchronise(matrix_mult_GPU(A, B, E))
    copy_out_GPU_async(E, N * size)

    // matrix_mult(C, D, F)
    GPU_synchronise(ALL EVENTS)
    matrix_mult_GPU(C, D, F)
    GPU_synchronise(matrix_mult_GPU(C, D, F))
    copy_out_GPU_async(F, N * size)

    // matrix_mult(E, F, G)
    GPU_synchronise(ALL EVENTS)
    matrix_mult_GPU(E, F, G)
    GPU_synchronise(matrix_mult_GPU(E, F, G)
    copy_out_GPU_async(G, N * size)

    GPU_synchronise(copy_out_GPU_async(E, N * size))
    GPU_synchronise(copy_out_GPU_async(F, N * size))
    GPU_synchronise(copy_out_GPU_async(G, N * size))
    total_sum(E, F, G);
  }
  ...
  free_GPU(A)
  free_GPU(B)
  free_GPU(C)
  free_GPU(D)
  free_GPU(E)
  free_GPU(F)
  free_GPU(G)
}
```

Table A.1: Kernel actions after step 1 for the input program of Listing 4.1. Dependencies are denoted by a 3-tuple where its first field represents the data-structure that is involved in this action. The second field represents the functions where the producing instructions of the dependency reside. The third field represents the functions where the consuming instructions of the dependency reside.

	`matrix_mult(A, B, E)`
Copy-in	{ A × `randomize_array(A)` × `matrix_mult(A, B, E)` }, { B × `randomize_array(B)` × `matrix_mult(A, B, E)` }
Copy-out	{ E × `matrix_mult(A, B, E)` × `print_array(E)`, E × `matrix_mult(A, B, E)` × `matrix_mult(E, F, G)`}
Allocation & Free	{ A × `randomize_array(A)` × `matrix_mult(A, B, E)` }, { B × `randomize_array(B)` × `matrix_mult(A, B, E)` }, { E × `matrix_mult(A, B, E)` × `print_array(E)`, E × `matrix_mult(A, B, E)` × `matrix_mult(E, F, G)` }
	`matrix_mult(C, D, F)`
Copy-in	{ C × `randomize_array(C)` × `matrix_mult(C, D, F)` }, { D × `randomize_array(D)` × `matrix_mult(C, D, F` }
Copy-out	{ F × `matrix_mult(C, D, F)` × `print_array(F)`, F × `matrix_mult(C, D, F)` × `matrix_mult(E, F, G)` }
Allocation & Free	{ C × `randomize_array(C)` × `matrix_mult(C, D, F)` }, { D × `randomize_array(D)` × `matrix_mult(C, D, F)` }, { F × `matrix_mult(C, D, F)` × `print_array(F)`, F × `matrix_mult(C, D, F)` × `matrix_mult(E, F, G)` }
	`matrix_mult(E, F, G)`
Copy-in	{ E × `matrix_mult(A, B, E)` × `matrix_mult(E, F, G)` }, { F × `matrix_mult(C, D, F)` × `matrix_mult(E, F, G` }
Copy-out	{ G × `matrix_mult(E, F, G)` × `print_array(G)` }
Allocation & Free	{ E × `matrix_mult(A, B, E)` × `matrix_mult(E, F, G)` }, { F × `matrix_mult(C, D, F)` × `matrix_mult(E, F, G` }, { G × `matrix_mult(E, F, G)` × `print_array(G)` }

Table A.2: Kernel actions after step 2 for the input program of Listing 4.1. Dependencies are denoted by a 3-tuple where its first field represents the data-structure that is involved in this action. The second field represents the functions where the producing instructions of the dependency reside. The third field represents the functions where the consuming instructions of the dependency reside.

`matrix_mult(A, B, E)`	
Copy-in	{ `A` × `randomize_array(A)` × `matrix_mult(A, B, E)` }, { `B` × `randomize_array(B)` × `matrix_mult(A, B, E)` }
Copy-out	{ `E` × `matrix_mult(A, B, E)` × `print_array(E)`}
Allocation	{ `A` × `randomize_array(A)` × `matrix_mult(A, B, E)` }, { `B` × `randomize_array(B)` × `matrix_mult(A, B, E)` }, { `E` × `matrix_mult(A, B, E)` × `print_array(E)`, `E` × `matrix_mult(A, B, E)` × `matrix_mult(E, F, G)` }
Free	{ `A` × `randomize_array(A)` × `matrix_mult(A, B, E)` }, { `B` × `randomize_array(B)` × `matrix_mult(A, B, E)` }

`matrix_mult(C, D, F)`	
Copy-in	{ `C` × `randomize_array(C)` × `matrix_mult(C, D, F)` }, { `D` × `randomize_array(D)` × `matrix_mult(C, D, F` }
Copy-out	{ `F` × `matrix_mult(C, D, F)` × `print_array(F)` }
Allocation	{ `C` × `randomize_array(C)` × `matrix_mult(C, D, F)` }, { `D` × `randomize_array(D)` × `matrix_mult(C, D, F)` }, { `E` × `matrix_mult(A, B, E)` × `print_array(E)`, `F` × `matrix_mult(C, D, F)` × `matrix_mult(E, F, G)` }
Free	{ `C` × `randomize_array(C)` × `matrix_mult(C, D, F)` }, { `D` × `randomize_array(D)` × `matrix_mult(C, D, F)` }

`matrix_mult(E, F, G)`	
Copy-in	-
Copy-out	{ `G` × `matrix_mult(E, F, G)` × `print_array(G)` }
Allocation	{ `G` × `matrix_mult(E, F, G)` × `print_array(G)` }
Free	{ `E` × `matrix_mult(A, B, E)` × `matrix_mult(E, F, G)` }, { `F` × `matrix_mult(C, D, F)` × `matrix_mult(E, F, G` }, { `G` × `matrix_mult(E, F, G)` × `print_array(G)` }