Masarykova univerzita
Fakulta informatiky

Optimization of mapped functions sequences using fusions on GPU

Master Thesis

Bc. Jan Fousek

Brno, Autumn 2010

Declaration

Hereby I declare, that this paper is my original authorial work, which I have worked out by my own. All sources, references and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Advisor: Mgr. Jiří Filipovič

Acknowledgement

I would like to show my gratitude to my advisor Mgr. Jiří Filipovič for the valuable guidance. I would also like to thank my family and friends who supported me during the completion of the work.

Abstract

When implementing a function mapping on the contemporary GPU, several contradictory performance factors have to be balanced. Previously a decomposition-fusion scheme was devised to guide such an implementation and this work is here further elaborated. To ease this process, an automatic source-to-source compiler is presented, while the main subject of this thesis are the core algorithms for generation, pruning and search in the state-space of possible implementations of the mapped function. The performance of the generated implementation is evaluated together with the overall complexity of the optimization process.

Keywords

map, function mapping, GPU, CUDA, medium-granularity

Contents

1 **Introduction** . . . 3
2 **Preliminaries** . . . 5
  2.1 *Optimization Algorithms* . . . 5
    2.1.1 Branch and Bound . . . 5
    2.1.2 Integer Programming . . . 5
  2.2 *CUDA computing model* . . . 6
    2.2.1 GPU architecture . . . 6
    2.2.2 CUDA programming model . . . 7
    2.2.3 Execution on a GPU . . . 8
  2.3 *Function Mapping* . . . 9
    2.3.1 Mapped Function Granularity . . . 9
    2.3.2 Decomposition-Fusion Scheme . . . 10
3 **Related work** . . . 11
4 **Automatic Fusion** . . . 13
  4.1 *Compiler Design* . . . 13
    4.1.1 Library of Elementary Functions . . . 15
  4.2 *Configuration Space* . . . 16
    4.2.1 The Data Flow Graph . . . 16
    4.2.2 Degrees of Freedom . . . 17
    4.2.3 Performance Implications . . . 19
  4.3 *State Space Searching* . . . 20
    4.3.1 Generation of Valid Fusions . . . 21
    4.3.2 Implementation Combinations . . . 21
    4.3.3 Shared Memory Allocation . . . 22
    4.3.4 Linearizing the Fusions . . . 27
    4.3.5 Fusion Combinations . . . 29
  4.4 *Performance model* . . . 30
    4.4.1 Performance Factors . . . 31
    4.4.2 Performance Prediction . . . 32
    4.4.3 Acquiring the Input Values . . . 33
5 **Experimental Evaluation** . . . 35
  5.1 *Medium-Grained Operations* . . . 35

5.2 *Prediction Accuracy* 36
5.3 *Performance of the Generated Code* 37
5.4 *Compiler Efficiency* 40
6 **Conclusion** 42

Chapter 1

Introduction

The contemporary GPUs outperform CPUs in order of magnitude in both instruction and memory throughtput, and are programmable enough to be able to run general purpose computations. Their performance is enabled by simpler although more rigid architecture and massive parallelism.

Implementing a function mapping on a GPU can provide enough parallelism; however, implementing more complicated functions in a monolithic way on the GPU is not only challenging, but also often inefficient. For example, the diverse memory consumption in different stages of the computation leads to different optimal granularity, and fixing it on the whole can reduce the parallelism and performance of some parts. Therefore, we divide the computation to separate simple operations which are more easily implemented in a optimal way and which exchange the data via the global memory.

To improve the ratio between the global memory operations and arithmetical operations, suitable groups of separate operations are fused together again in order to exchange data via the faster on-chip memory. This part of the process is far less straightforward, and also the number of possible fusions and their implementations is large. This thesis introduces a source-to-source compiler which automates this scheme.

During the fusion step of the scheme, the compiler begins with a simplified description of the computation and a library of elementary operations with descriptive metadata, such as the expected size of the threadblock or an empirically evaluated performance behaviour under limited parallelism. In the first step, the compiler parses the computation description into the dataflow graph.

In the second step, it searches for gainful fusions and their implementations. Since the statespace of these implementations is very large even for reasonable inputs, a set of heuristics is used to prune it and to lead further optimizations. And as the performance estimation is based on approximated values, a small set of the most promising candidate implementations is benchmarked after the process to determine the most optimal implemen-

tation of the mapped function.

In the last step, the final code is generated using the code templates from the library of the elementary operations, and handed over to the user.

The decomposition fusion scheme, together with the compiler, enable the programmer to implement and evaluate generic *reusable* functions on one hand and, on the other hand, to write application specific code in a simple, high-level language. The actual implementation is then automatically generated by translating the high-level specification into calls to generic functions and fusing some of them in order to achieve an implementation as efficient as possible.

This thesis elaborates on an already published work done by Jiří Filipovič and co-authored by me to provide further details and steps towards a practical implementation. While the medium-grained mapping pattern and the decomposition-fusion scheme were published in [6], [8], and the concept of the optimizing compiler is submitted for publishing in [7], the design and implementation of the actual algorithms for generation, pruning and search for the most efficient mapped function implementation are results of this thesis.

The work is structured as follows. After the introductory chapter, a theoretical background and tools are presented in chapter two. Here, a generic description of the GPU architecture and a CUDA programming model is covered, accompanied by a description of the later utilised optimization algorithms and a formal description of function mapping. Chapter three sets the presented work into a broader context and also points to previously published results. The main contribution of this thesis is presented in chapter four. First, the overall design of the compiler is presented, followed by a description of the particular algorithms used during the state space generation and pruning. In chapter five, an experimental evaluation of the compiler is presented, followed by the last chapter, summing up the achieved results.

Chapter 2

Preliminaries

In this chapter, the basic principles used in the later text are presented. These include the generic sketches of the optimization algorithms in the first section, an overview of the architecture and programming model for the general purpose GPU, as it was established lately by NVIDIA, and the definition of function mapping on the GPU, which is the problem this work addresses.

2.1 Optimization Algorithms

2.1.1 Branch and Bound

The Branch and bound is an optimization technique for searching a state space of candidate solutions for the optimal one. It consist of two basic principles: first, candidate solutions are generated in an iterative manner – the so called branching. Second, an upper and a lower bound are determined for the value of the optimal solution and used to prune the state space by discarding whole sets of fruitless solutions during the branching phase.

This technique is especially useful when an exact solution of the optimization is required and when the upper and lower bound can be computed in an easy way with reasonable precision. A nice overview is provided in [12].

2.1.2 Integer Programming

Integer programming is a special case of a linear programing problem with integer variables. The linear programming is a technique for solving optimization problems defined by linear cost function and constraints. For a set of variables $X = x_0 ... x_n$ the term $c_0 x_0 + ... + c_n x_n$ represents the cost function, the constraints are denoted as $a_0 x_0 + ... + a_0 x_0 \leq k$ where k is some constant, and solution is defined as assignment of a value from the variable domain to a particular variable. In the case of the integer programming, the

variables are from the $\mathbb{N}$ domain and this variant is known to be NP-hard. Linear programming is an old albeit still popular technique dated back to the pre-computer era, and there are numerous solvers available. Therefore it was possible, with the help of an external fine tuned solver [3], to keep the solving time of our integer problem low. A detailed description of the linear programming and its application can be found in [14]

2.2 CUDA computing model

The CUDA (Computer Unified Device Architecture) is a computing architecture developed and used by NVIDIA GPU to enable generic computations on the GPU. The programming model itself is closely bound to the GPU hardware architecture and one has to keep this architecture in mind to understand the performance implications of particular design and implementation decisions. In following subsection I provide a short overview of both the GPU architecture and the CUDA programming model.

2.2.1 GPU architecture

The hardware layout of a GPU based on G80 is depicted on Fig. 2.2.1. The device consists of several multiprocessors, one device memory, a thread scheduler and an interconnection to the host. Each multiprocessor consists of a shared memory, the constant and texture caches, one instruction unit and several processors equipped with register memory.

All processors on one multiprocessor have to execute the same instruction at a time, as there is only one instruction unit on one multiprocessor. The thread scheduling is performed in the hardware by the thread scheduler.

The fastest and smallest memory – the registers – reside nearest to the particular processors. The size of the registers per multiprocessor ranges from 32 KB to 128 KB. All processors on a single multiprocessor have access to the shared memory which ranges in size from 16 KB to 48 KB, the access speed being similar to the access speed to the registers with some limitation on the access pattern. Beside the shared memory, the multiprocessors provide the read-only constant and texture caches. All multiprocessors can access the largest and slowest device memory with a latency of aligned access around hundreds of cycles and a bandwidth in an order of magnitude than in the case of registers and shared memory.

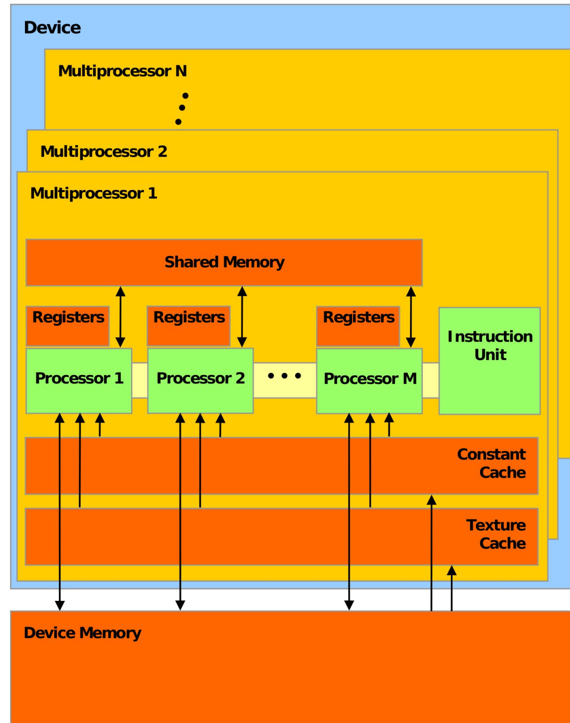

Figure 2.1: The hardware layout of the G80 based GPU (reprinted from [15]).

2.2.2 CUDA programming model

One of the possibilities how to write programs for the CUDA systems is the so-called *C for CUDA* which is an conservative extension of C language. These extensions allow the programmer to describe CUDA specific aspects of the program while keeping the actual code in the well known C. The extensions to C are following [15]: function type qualifiers denoting whether the function should be executed on the device or on the host, variable qualifiers specifying the type of device memory, a device kernel execution directive, and built-in variables describing the position of a thread within the CUDA thread hierarchy. Additionally, the access to the CUDA API is provided, enabling e. g. host-device synchronization, device management, etc.

In CUDA, the threads are organized hierarchically. Threads operating on spatially near data are grouped into three dimensional *thread blocks* which are organized into a *grid*. This thread hierarchy reflects the hardware layout

as can be seen on Fig. 2.2.2. A single thread is executed on a single processor and has access to its register memory. The thread block executes on a single multiprocessor and therefore the threads within it can exchange data via the shared memory. The device memory has to be used to share data between thread blocks.

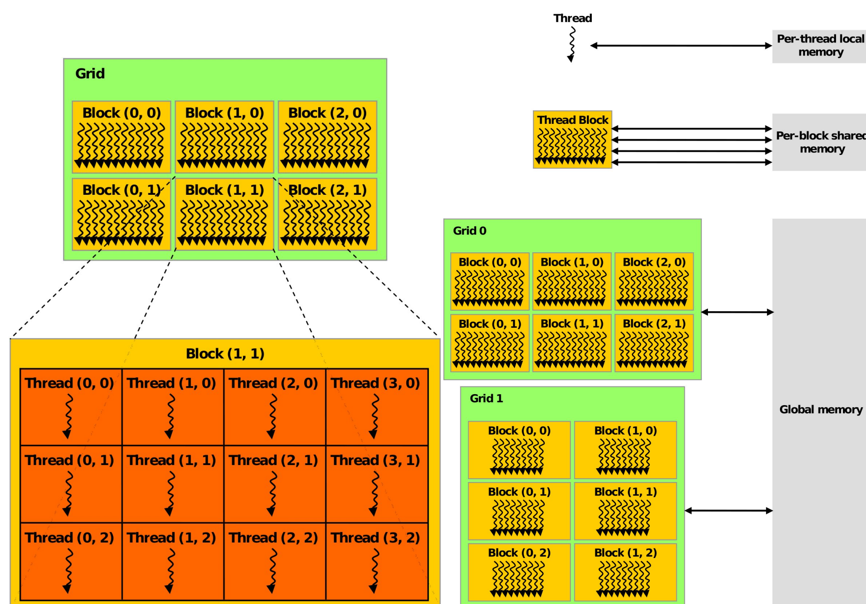

Figure 2.2: Comparison of the threading and memory hierarchy of the GPU (reprinted from [15]).

2.2.3 Execution on a GPU

A significant source of computational performance lies in the way how the latencies to the device memory are masked. On a multiprocessor, numerous[1] threads are executed in parallel and, with the help of hardware switching and planing, those who are not waiting for the return of data transfer are picked for execution. Thanks to this switching the computational and memory transfer times can be effectively overlapped and the overall performance can get closer to the computational or bandwidth theoretical peak. However, for this overlapping to be effective a large number of threads is

1. usually more than the number of processors on the given multiprocessor

required to be executed in parallel. If the parallelism is limited for any reason, so is the achieved performance.

The high numbers of threads being run at once are planned, switched and managed on the device directly in hardware, employing the so-called Single Instruction Multiple Data (SIMT) manner of execution. The threads are within the block scheduled and executed in consecutive groups of size 32 called *warps*. All threads within one warp start the execution at the same program counter, but are then allowed to execute independently. However if a divergence in the execution within a warp occurs, the warp is split and executed in serial manner, lowering the overall performance.

2.3 Function Mapping

We call n-ary function f to be *mapped* to the input lists $L_1, .., L_n$ when it is applied element-wise to these lists producing single list of results. Formally $map(f, L_1, .., L_n) = [f(l_{1,1}, .., l_{1,n}), f(l_{2,1}, .., l_{2,n}), .., f(l_{m,1}, .., l_{m,n})]$, where $l_{i,j}$ is j-th element of L_i.

Mapping a function to numerous inputs yields a sufficient degree of parallelism necessary for the GPU to operate efficienly. However, the optimal implementation becomes increasingly complicated task in case of more complex mapped functions. Two main tasks arise for the developer: the thread-to-data element granularity and the kernel-to-code granularity.

2.3.1 Mapped Function Granularity

The previously described CUDA thread and memory hierarchies suggest two basic approaches to the division of the workload among the threads. The first is the *fine-grained* implementation where a single thread processes a single data element, and the other is the *coarse-grained* approach where a single data element is processed by a block of threads. The coarse-grained implementation obviously requires the mapped function to be implemented in a parallel manner so that it can be performed by multiple threads.

Usually the decision on the granularity of function implementation is based on the size of the input and output data elements. In other words, small data elements which fit in the on-chip memories available to a single thread imply the fine-grained implementation, whereas the larger data elements have to be stored in the shared memory and the workload distributed over the block of threads.

However, there is a class of functions which don't fit neither to the fine- nor the coarse-grained pattern. These functions operate on data which re-

strict parallelism if processed by a single thread, since their consumption of the on-chip memory per function is high, but at the same time they don't provide enough workload for a block of threads, as the number of threads should be a multiple of warp size. In other words, the on-chip memory requirements are distributed over multiple threads but at the same time the size of the block remains sufficiently high.

2.3.2 Decomposition-Fusion Scheme

The core problem is that the granularity of the optimal implementation of a more complex mapped function often varies for distinct parts of the computation, and the amount of required on-chip resources is raised by the intermediate results stored there. Therefore, implementing the mapped function in a monolithic manner leads to limited performance. At the same time it is possible to decompose the mapped function into several successive simpler functions which store the intermediate results in the global memory. Such a decomposition can be seen as a directed acyclic graph (DAG) of data flow $G = (V, E)$, where the set of vertices V represents the particular functions and the edges E describe the data dependecies between them. For two vertices $v_1, v_2 \in V$ representing functions f_1 and f_2, there is an edge $e = (v_1, v_2) \in E$ representing a list L_{tmp} iff $L_{tmp} = map(f_1, L_0, ..L_n)$ and $map(f_2, L'_0, .., L_{tmp}, ..L'_m)$.

While this approach makes the variable granularity possible and the simpler functions are easier to develop, it introduces a non-trivial pressure on the global memory bandwidth by storing and loading the intermediate data. To overcome this disadvantage, we have introduced the *decomposition-fusion* scheme. First, the complex function is decomposed into fairly simple and reusable functions which are easy to implement in an optimal manner, and in the second step some of these functions are again fused together so that they can exchange the intermediate results via the faster on-chip memory. During the fusing phase there is a search for such sets of the elementary functions that provide maximal performance when fused. To achieve this goal the spared global memory traffic has to be balanced against the loss of performance implied by different parallelism of the fused functions or by increased on-chip resources by the intermediate results.

Chapter 3

Related work

The map is a popular primitive in parallel processing and it is frequently used in the form of a map-reduce pattern to solve numerous problems [5]. There already are implementations of map-reduce for the GPU architecture [4] [9], however they both limit the thread-to-data granularity to one thread per mapped data element. This approach is unsuitable for more complex mapped functions as the on-chip memory resources available to a single thread are strongly limited on the GPU. The mapped function and respective reduction are present in both [4] and [9], fused over the intermediate result stored in the on-chip memory. However, this fusion is performed every time since there is no difference in the usage of the GPU resources.

The initial inspiration came from an acceleration of the Finite Element Method equation assembly on the GPU. The size of the per-FEM-element data and parallelism attainable in our FEM simulation renders both coarse- and fine-grained implementation of the mapped function inefficient. Moreover, both the size of the intermediate results and the optimal degree of parallelism vary during the computation. Therefore in [8] we have introduced the medium-grained implementation of mapped functions and a fusion-decomposition scheme to guide the implementation of more complex mapped functions. An application of these principles was shown on an acceleration of a FEM matrix assembly for the StVenant material. A more detailed description of the medium-grained implementation is given in [6]. Further, the analysis of the medium-grained pattern along with a sketch of the automation of the fusion phase of the decomposition-fusion scheme were submitted for publication in [7].

Independently on our work, the medium-grained pattern was used by [11] in an acceleration of the discontinuous Galerkin method. This work focuses mainly at problems related to the discontinuous Galerkin, although some general characteristics of the medium-grained implementation are also mentioned. The motivation for the medium-grained implementation separated in multiple kernels was to find a better balance between the on-chip resources and the size of the problem solved by the mapped function.

However, both the separation of the computation and the implementation of the particular kernels were done by hand. [13]

An interesting analogous work was recently published by [2]. It deals with an optimization of the arbitrary source code, using BLAS on the CPU. While the BLAS routines are believed to be highly optimized, the subsequent call to these routines can result in suboptimal performance of the whole composition due to unoptimized memory access. This issue is addressed in [2] by fusing the loops of the selected subsequently called BLAS routines, giving rise to application specific fusions. Furthermore, a source-to-source compiler automating the whole process was presented and operated in analogous manner as the compiler presented in this thesis. It starts with an Matlab-like description of the computation, then enumerates the possible fusions and, with the help of a mixture of empirical and analytical performance prediction, determines the most efficient fusions.

Various work was done on the prediction of the CUDA code. Most effort was invested in the static analysis of the CUDA source code, resulting as shown in work of [1]. A workflow graph is extracted from the kernel source code and the performance prediction is computed from its structure. Since the analytical model presented in [1] has to capture all vital performance factors, it is fairly complicated and has no ability to adapt to changes in the GPU architecture introduced with newer generations of the hardware. Also, in our case there is no need to repeatedly analytically determine the performance of particular elementary functions as they don't change over time.

Chapter 4

Automatic Fusion

Several counteracting principles determine the resulting performance of particular fusion and also there are numerous variants of possible fusions for a given decomposed complex function. To avoid time costly and error-prone testing by hand, an automatic tool is presented in this section. This optimizing compiler is able to choose an optimal implementation of the mapped function and to generate the CUDA code from prepared templates of the particular fused functions. The choice of the optimal implementation of the mapped function is done by searching the state space of all possible implementations. This state space is very large and therefore the performance of particular implementations is not evaluated empirically but is instead predicted.

A short note to the terminology: The input is a description of a *mapped function* decomposed to several *elementary functions*. Each of these functions is taken from the *library of elementary functions*, where it is present in several variants refered to as *implementations of the elementary function*. A set of the elementary functions can build up a *fusion* and a *fusion linearization* can be subsequently created by ordering these fused functions. Setting several properties described below on the fusion linearization results in a final *fusion implementation* which can be translated to a CUDA source code.

4.1 Compiler Design

The compiler begins with a C-like description of the mapped function and a library of elementary functions with descriptive metadata such as input and output types, block sizes etc. In a first step the computation description is parsed into a data flow graph and performance of all required elementary functions is evaluated by benchmarking under various combinations of reduced parallelism and number of data elements processed per block.

In the second step the state space of possible implementations is searched and the performance prediction of the candidates is based on the previously

benchmarked performance of elementary functions. Detailed description of the performance prediction is given in section 4.4. As a result several best implementations are picked and the compiler generates the corresponding CUDA code. Finally the best implementation is chosen according to its real performance and handed over to the user. The compiler doesn't perform any low level optimizations and leaves these to the native CUDA compiler.

Example Listing 4.1 shows description of mapped function $f : X = ||A \cdot B \cdot v||_2 \cdot (C \cdot D + C)$ where A and B are 3×3 matrices, the v is a vector of size 3, the C and D are 5×5 matrices and $||x||_2$ is a vector l^2-norm. As you can see, the original function was decomposed to elementary functions with explicitly named and typed intermediate results. First four rows declare the types of the used variables. Note that the code describes computation of the mapped function, not the mapping itself which is also reflected in the types of the variables which describe single data elements instead of lists. Lines 6 and 15 denote the input and output arguments of the mapped function. At the present the number of output variables is limited to one for the sake of simplicity but this limitation should be lifted in future to enable mapping of more generic functions. Rest of the code describes the actual computation and is fairly self explanatory.

Listing 4.1: Example description of the mapped function

```
MATRIX3x3 A, B, M1;
MATRIX5x5 D, E, F, M2, M3;
VECTOR3 c, v1;
SCALAR s1;

input A, B, c, D, E;

M1 = mmul33(A, B);         // M_1 = A · B
v1 = mvmul33(M1, c);       // v_1 = M1 · c
s1 = venorm3(v1);          // s_1 = ||v_1||_2
M2 = mmul55(D, E);         // M_2 = D · E
M3 = madd55(M2, D);        // M_3 = M_2 + D
F = smmul55(M3, s1);       // F = M_3 · s_1

return F;
```

The language of the computation description is limited to basic function call without loops or branching possibilities. While the branching functionality is expected to be covered within particular kernels if not avoided altogether for performance reasons, finite loops can be in current implementa-

tion substituted by unrolling.

4.1.1 Library of Elementary Functions

The library of elementary functions can contain arbitrary simple functions, in our example case a range of basic linear algebra operations such as matrix-matrix multiplication, vector norm, etc. These are the basic building elements from which the mapped function is composed. All of them are written for data elements of fixed size (e. g.square matrices of size 5) and hand tuned for best possible performance. Programming of such functions is reasonably simple and it is possible to reach near optimal performance of these particular functions with reasonable effort.

Short note on the fixation[1] of the matrix size. We are dealing with the implementation of given complex function working with some data elements and we suppose that the size of these data elements is determined by the nature of the solved function. What changes for different calls to the mapped function – and is therefore problem dependent – is the number of input data element instances, not the size of the data elements.

The library functions have prescribed form to enable automated processing. Every function consists of load and store routines, which take care for data transfers of the inputs and output between the global and shared memory, and a compute routine performing the actual computation on the data stored in the shared memory. There is one load function for every input argument of the function. Thank to this separation of the memory transfers and the computational part it is possible to create fusions easily by gluing together only the compute routines of all fused vertices and using the appropriate load routines to fetch the input data and respectively the store routines to save the fusion results to the global memory.

The important properties of the elementary functions are stored in the metadata comments in the function sources. For the load function there is a notation of the data type which the function transfers and for the compute function the metadata stores the required sizes of grid and block.

There are two types of the on-chip memory resources available to the threads. It is the shared memory and the registers. While the overall design of the compiler was created with both in mind, at present it takes only the shared memory into account. This limitation was introduced for the sake of simplicity of the initial version of the compiler and will be removed in the

1. The size of the matrices is fixed during the benchmarking and compilation process. However there is no need to write every single elementary function by hand as they can be easily generated from parametrized templates by an external script.

future.

4.2 Configuration Space

4.2.1 The Data Flow Graph

As mentioned before the decomposition of given mapped function f can be represented by a *directed acyclic graph* (DAG) $G = (V, E)$ with vertices V labeled by the functions and edges E by the data types of the intermediate results. This representation is the core formalism used by the compiler. To capture the input and output parameters of the function f we add a vertex for each such a variable representing helper functions loading the input data from the host to the device memory and storing the results back to the host memory. Example of such a DAG is depicted on Figure 4.2.1. This DAG corresponds to the computation description shown on Listing 4.1.

Various information has to be stored in the vertex to make the generation of the CUDA code possible. The vertex represents a call to CUDA kernel and therefore it has to store the name of the implementation of the particular elementary function, the number of elements it processes per block in the case of the medium- or fine-grained implementation and also the other two dimensions of the thread block. Also the size of the grid has to be set for each kernel from the number, however it can be easily computed from the size of the input elements to be mapped on and the number of elements processed in one block. These variables can be set to arbitrary values during the initial creation of the data flow graph as they are subject of the optimization process described in the next section.

The fusions are represented as single vertices in the modified main graph G' associated with a DAG describing for a given fusion F the computation within as $G_F = (V_F, E_F)$ where $V_F \subseteq V$ and $E_F \subseteq E$. Similarly to the main graph G vertices representing the loading of the inputs and storing the outputs of the fusion are added to the G_F, however they represent loading and storing data to and from the on-chip memories as the intermediate results of the fused functions are stored there. Each such a load vertex is created for a group of input edges from E sharing one source to prevent redundant copying of identical data. Additionally each of such a I/O vertices has references to these incoming edges to the fusion which it represents so that the information about the data dependence is kept between particular functions both inside the fusion in G_F and outside of the fusion – in the DAG G'. Example of the fusion DAG within a mapped function DAG is depicted on Figure 4.2.1.

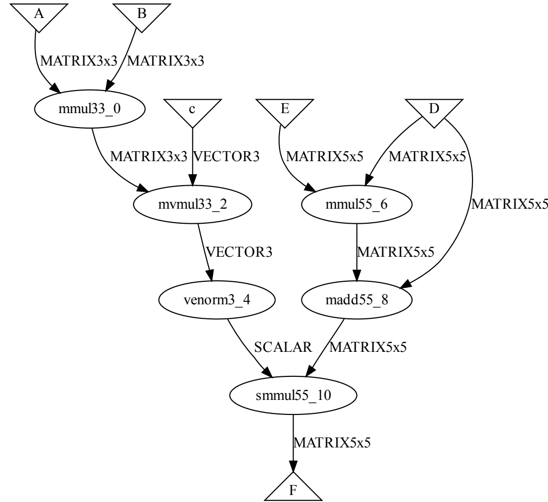

Figure 4.1: Example of a DAG representing the mapped function computation

Also the fusion has to bear the information about the thread block size and number of elements. The dimensions of the thread block required by the particular fused functions can differ and as the threads cannot be dynamically created and destroyed during the kernel execution the total number of threads has to remain constant during the lifetime of an CUDA kernel. Therefore the fusion vertex stores the maximal number of thread required by the particular fused functions as an one dimensional block and recalculates the position for each thread within this block before every execution of a fused function. This approach can render some of the threads idle if some function requires smaller block, however almost no performance is lost as they don't perform any computation and are quickly scheduled out. Considering the number of elements processed by one block this number has to be constant during the whole execution of the fusion kernel because it is not possible to exchange the intermediate results outside of the fusion.

4.2.2 Degrees of Freedom

There are numerous possibilities how to perform the second step of the decomposition-fusion scheme – i. e. the fusing – for given decomposition of

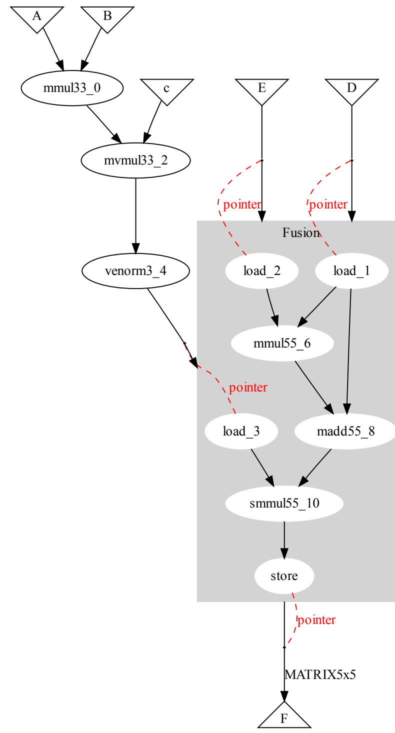

Figure 4.2: Example of a fusion subgraph within a DAG representing the mapped function computation

mapped function. Following possible choices has to be taken into consideration:

1. all fusible subgraphs of the data flow graph
2. all linearizations of particular fusions
3. all implementations of elementary functions within fusion
4. all valid combinations of fusions (non-overlapping)

The subgraph G_F of the data flow graph G can in general be fused if it is a connected component of the graph G and if there is no outgoing edge $e \in E_F$ from this component such that there is a path beginning with the

edge e and returning back to the component G_F. Although the fusible subgraphs are limited by this condition the number of such subgraphs for more complex functions can grow quickly. Further in the current implementation of the compiler a function can have only one output and therefore a fusion can be created only from a sink-like subgraph of the data flow graph if there is only one fused vertex with edges leading out of the fusion. However this limitation has been introduced only to simplify the implementation and will be removed in near future as functions with multiple outputs are also interesting in general.

To translate the a DAG representing the fusion subgraph to the CUDA code a linearization has to be made. A linearization of a DAG $G = (V, E)$ is such a ordering $\lambda : V \rightarrow< 1, |V| >$ of its vertices that $\forall (v_1, v_2) \in E : \lambda(v_1) < \lambda(v_2)$, i. e. that there is no edge leading to previous vertex. There can be obviously multiple possible linearizations for a given DAG.

Every elementary function can be present in the library in multiple distinct implementations. These implementations differ for example in required block sizes or performance behaviour. Therefore for every vertex in a fusion DAG some of these particular function implementations has to be chosen.

Finally multiple fusion can be part of the DAG representing the mapped function f. Therefore from all possible candidates most effective combination has to be chosen according to the resulting performance gain.

4.2.3 Performance Implications

The choices presented in previous section have direct implication on the performance of the subsequently generated code.

In mapped function decomposition the particular functions have to exchange the intermediate results. This is done outside the fusion over the global memory yielding every intermediate result to be stored to the global memory and at least once loaded from the global memory. This traffic is avoided inside the fusion as the fused functions store the intermediate results in the fast on-chip shared memory. Therefore the performance gained by easing the pressure on the global memory bandwidth is determined by the choice on the set of vertices building up the fusion.

The linearization predetermines the amount of shared memory required by the fusion. The shared memory of a CUDA kernel cannot be allocated dynamically and the size of the share memory required by a CUDA kernel puts an upper bound on the degree of achievable parallelism, because all blocks on one multiprocessor share one on-chip storage. Therefore it is vital to order the vertices of the DAG in such way that the intermediate results

are not stored longer than is necessary for the actual usage and at the same time the shared memory fragmentation can be avoided.

The implementations of the particular functions in a given fusion can operate on different granularity and can require different number of threads to process one element. While the loss of performance from the recomputation of the coordinates within a thread block is rather small (though not negligible as it may require modulo operation which is slow on contemporary GPUs), the different requirements of the on-chip resources by the implementations with different granularity can lead to loss of attainable parallelism in some parts of the computation and consequently limit the performance.

And finally only nonintersecting fusions can be used to implement given mapped function and so the performance contribution of every fusion has to be evaluated and compared to other overlapping fusions and respective standalone functions to determine the most efficient combination.

4.3 State Space Searching

Several decisions with numerous combinations has to be made during the search for the most efficient implementation of the mapped function. Every combination of the these decisions results in a different implementation of the mapped function and forms a single state in the state space of all possible implementations. The state space is generated and pruned in following order. First all possible fusions are generated based on a given mapped function DAG. The size of the fusions (number of fused vertices) is limited by a chosen constant to limit the number of such fusions and to keep the further computations feasible. Then for every fusion all possible combinations of the implementations of the fused functions are enumerated multiplying the number of generated fusions. After this step, every fusion is linearized forming a fusion implementation and the performance is estimated. At this point the unfruitful fusion implementations are discarded based on the performance estimation presented in Section 4.4. In the last step the candidates on the implementation of the mapped function are chosen form the sets of evaluated fusion implementations and the standalone vertices and handed over to the optional final empirical testing. Algorithms used in particular steps of the state space generation and pruning are presented in following sections.

4.3.1 Generation of Valid Fusions

Basically the number of possible fusions on a given DAG $G = (V, E)$ is in the worst case near the number of subsets of all vertices – $|2^V|$. However the performance gain of the fusions stagnates for large number of fused vertices so it is possible to consider only fusions of limited size. The stagnation is caused by the decreasing the ratio of the spared memory transfers compared to the amount of on-chip resources required by the fusion. Also with the growing size of the fusion the probability of reaching the computational bound grows. Therefore the number of fusions of size lower then k is bound by the number of subsets of V of size at most k is then:

$$\sum_{i=1}^{k} \binom{|V|}{i}$$

Note that this is the worst case of the number of possible fusions. The exact number depends on the actual structure of the DAG and is usually significantly lower. The algorithm enumerates all possible subsets of V of size up to the previously set k and tests for each such a subset if it can be turned into a fusion. A set of vertices $V_F \subseteq V$ can be *fused* if and only if for every edge $(v_1, v_2) \in E$ such that $v_1 \in V_F$ and $v_2 \notin V_F$ there is no path from v_2 to any of the fused vertices $v \in V$. Such a path would lead to a deadlock as it is not effectively possible to execute two different functions in parallel and synchronize between them. This property can be checked in very fast way by starting a breath-first search from every outgoing edge and coloring the visited vertices. Therefore every edge is tested only once on belonging to a forbidden path and the overall complexity is in $O(|E|)$.

The worst case number of possible fusions is reached on a DAG composed of one source and one sink vertex with an arbitrary big layer of vertices connected to both in between and is presented on the Figure 4.3 on the left. On the other hand the amount of possible fusion is quadratic when the structure of the DAG is linear as depicted on the Figure 4.3 on the right. Although the structure of a typical DAG describing a mapped function is not linear it is far from the worst case as the in-degree of the vertices is limited by the arity of the elementary functions.

4.3.2 Implementation Combinations

For every function inside a fusion there are several implementations to be chosen from in the library of elementary functions with different thread to

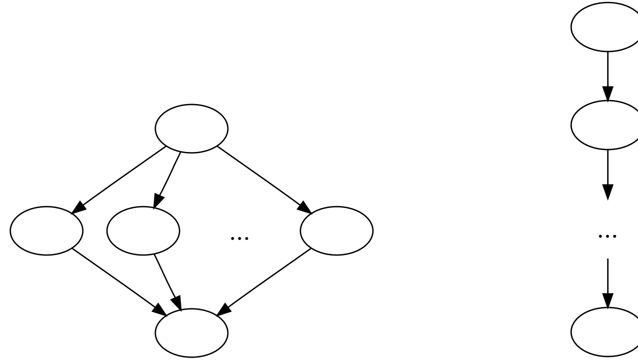

Figure 4.3: Left: the worst case structure of a DAG. Right: the easiest structure.

data granularity or – in the case of the fine- and medium-grained implementations – with different optimal number of elements processed by one thread block. The generation of all possible combinations of these implementations for one fusion is fairly simple and is done by enumeration. The number of the different implementations I_f of one function f is usually lower than ten and therefore for a fusion of functions $f_1, ..., f_k$ the number of the possible implementations combinations computed as $I_{f_1} \cdot ... \cdot I_{f_k}$ is limited by the choice of constant k limiting the size of generated fusions. The particular combination of the implementations determines the degree of limited parallelism (and subsequently performance) and it is therefore possible to prune the underperforming combinations by performance prediction described in section 4.4.

4.3.3 Shared Memory Allocation

The size of allocated shared memory has a big influence on the parallelism exposed by given fusion because the amount of the on-chip resources is limited and it is therefore crucial for the performance of the fusion to keep consumption low. The size of the shared memory of the particular linearization of the fusion is determined by the allocation of the variables in the shared memory. It is not necessary to allocate extra space for every variable because not every variable is used in every time point during the fusion computation. Therefore the space used by the variables holding the inter-

mediate results, which will be no more used, can be filled with the data of actually required variables. As the current GPU doesn't allow to allocate the shared memory dynamically a memory reusing scheme had to be devised and is presented in this section.

Instead of allocating the shared memory for every variable one big memory space is allocated for the whole fusion and the variables are then associated with a offset pointing to the beginning of the space designated to store that particular variable. It is possible to reuse the space belonging to the unneeded variables by simply associating another variable with offset pointing somewhere in the unneeded space. So far we consider all variables to be arrays of the same basic type, e. g. *float*. Also note that every variable can be associated with multiple edges: a set of outgoing edges of one function – describing situation where a result of one function is used as input of multiple functions – is associated with one variable.

Formally let the set B denote all shared memory variables of a given fusion with a size labeling function $\sigma : B \to \mathbb{N}$ where for some $b \in B$ the $\sigma(b)$ is the length of array representing the type of b in memory (for example the $\sigma(b_{matrix5\times5}) = 25$). Let the $\alpha : B \to \mathbb{N}$ be the association of the variables with the respective offset. The size of the shared memory space can be then computed as $\max_{b \in B}(\alpha(b) + \sigma(b))$.

The offset function α however has to assign the offset in such a way that the space occupied by two active variables does not overlap. This property can be formulated for given vertices V and the respective linearization $\lambda : V \to \mathbb{N}$ giving the position in the ordering as follows. For every variable $b \in B$ let the set of edges $E_b \subseteq E$ denote the edges associated with the variable. The variable b is called *active* on a position i of the linearization if there is an edge $(v_1, v_2) \in E_b$ such that $\lambda(v_1) \leq i \leq \lambda(v_2)$. Let $A_i \subseteq B$ denote the set of variables active at position i then on every position i for every two variables $b_1, b_2 \in A_i$ the intersection of corresponding memory intervals is empty: $[\alpha(b_1), \alpha(b_1) + \sigma(b_1)) \cup [\alpha(b_2), \alpha(b_2) + \sigma(b_2)) = \emptyset$.

This property also implies the lower bound on the total allocated shared memory space. Let us denote this lower bound as M_{lb} and define it as a the biggest sum of active variables on particular positions in the linearization:

$$M_{lb} = \max_{v \in V} \left(\sum_{b \in A_{\lambda(v)}} \sigma(b) \right)$$

The lower bound M_{lb} can be easily computed for given linearization and the usage is twofold. First it is used during the linearization generation

algorithm presented in next section as the approximation of the the size of the allocated memory of particular linearization allowing to prune search space of possible linearizations. Second it is used in the allocation computation algorithm 4.3 presented below, where it allows to recognize the optimal allocation and stop the search. However it is to be noted, that the value of optimal allocation can be in some cases higher than M_{lb} due to unavoidable fragmentation of the common shared memory space. Nevertheless this fact doesn't break the linearization algorithm as the value of M_{lb} is used only as a heuristic parameter and also the allocation algorithm remains sound and only looses the ability to recognize the optimal solution at first sight.

The branch and bound algorithm for the computation of optimal shared memory allocation is presented in listing 4.3. It performs two basic steps: first it uses simple hungry algorithm to generate initial solution and afterwards it searches for an optimization with the help of backtracking. Both the hungry and the optimal algorithm operate on set of the variables ordered by the first usage of the variable in the linearization. The solution is build gradually from the beginning of this set so that when the offset for the variable b on place i, denoted as $\alpha(b^i)$, is being computed than for every $j < i$ the offset $\alpha(b^j)$ is already fixed and for every $l > i$ the offset $\alpha(b^j)$ is undefined. Therefore the process of assigning the offset $\alpha(b^i)$ can be seen as a branching part of the branch and bound algorithm with the all possible values of the offset for variable b^i representing branches leading from actual partial solution to different complete solutions. Bounding part of the branch and bound scheme is performed in pruning the unfruitful branches of the partial solutions based on the lower and upper bound values presented later on.

Short remark to the notation. E_F is a set of edges within the particular fusion F, O_F is the ordering of the vertices of a fusion F, i. e. an array where the vertices are ordered by the linearization numbering λ. Sets $in(V)$ and $out(V)$ denote the incoming and outgoing edges of the vertex V.

Listing 4.2: Helper functions for the optimal allocation of the shared memory

```
1  function computeCollisionSets(O)
2    for every b ∈ B:
3      C_b = ∅
4    for every edge e = (v_1, v_2) such that v_1, v_2 ∈ O:
5      let i and j be the positions of v_1 and v_2 in O
6      b is such that e ∈ E_b
7      for k form i to j:
```

```
8        A_k = A_k ∪ b
9     for every A_i:
10      for every b ∈ A_i:
11        C_b = C_b ∪ A_i
12    return {C_b, b ∈ B}
13
14  function computeHungryAllocation(O)
15    computeCollisionSets(O)
16    for every b ∈ B
17      choose smallest m such that for every b_c in C_b:
18        [m, m + σ(b)) ∪ [α(b_c), α(b_c) + σ(b_c)) ≠ ∅
19      α(b) = m
20    return α
```

The algorithm uses in both steps the sets of collision variables C_b for every variable $b \in B$. Two variables $b_1, b_2 \in B$ are said to be in *collision* if they are both active at some position i of the linearization – $\exists i : b_1 \in A_i \wedge b_2 \in A_i$. These sets are for given variable b computed as an union of all edges starting or ending on any of the vertices in the range given by the indexes of the start and end vertices of the edges in E_b.

The hungry algorithm iterates over all variables and chooses for every variable b the smallest offset such that the variable doesn't overlap with previously processed ones. Let b^i be a variable for which the offset is to be chosen. Then for any $j < i$ the $\alpha(b^j)$ is defined, for every $k \geq i$ the $\alpha(b^k)$ is not defined. The only variables with which the b^i can overlap are those with which it is in conflict, therefore for the correct assignment it is enough to check against the variables in $b_c \in C_b$ such that $\alpha(b_c)$ was already assigned. The choice on line 17 is done by sorting the already assigned colliding variables by the offset and searching from zero for space big enough for the b^i to fit in. If there is no such a space, the offset is set to point right behind the space occupied by the last conflicting variable. Note that it is not important how the sort handles the variables b_c with undefined offset as they will be dealt with at later point of the solution generation.

Listing 4.3: Optimal allocation of the shared memory

```
1  function computeOptimalAllocation(O)
2    computeCollisionSets(O)
3    α_best = computeHungryAllocation(O)
4    M_best = M_{α_best}
5    M_lb = computeLowerBound(O)
6    if M_best = M_lb
```

```
7       return α_best
8     i = 0
9     while true:
10      choose smallest m such that for every b_c^j ∈ C_{b^i}, j < i:
11        [m, m + σ(b^i)) ∪ [α(b_c), α(b_c) + σ(b_c)) ≠ ∅
12      while (m + σ(b^i) ≥ M_best)
13        i = max({j : b^j ∈ C_{b^i} ∧ j < i})
14        m' = α(C_{b^i}) + 1
15        choose smallest m ≥ m' such that for every b_c^j ∈ C_{b^i}, j < i:
16          [m, m + σ(b^i)) ∪ [α(b_c), α(b_c) + σ(b_c)) ≠ ∅
17        if i = 0 ∧ m + σ(b^0) ≥ M_best
18          return α_best
19      α(b^i) = m
20      i++
21      if i = |B|
22        M_best = max_{b∈B}(α(b) + σ(b))
23        α_best = α
24        if M_best = M_lb
25          return α_best
26        let j be the first index such that α(b^j) + σ(b^j) = M_best
27        i = j
```

The memory allocation produced by the hungry algorithm can be fragmented and in that case it can be further optimized. However it gives a good upper bound for the value of optimal solution. Note that if the value of the hungry solution M_{best} is equal the lower bound M_lb described earlier it implies that it is in fact already optimal (line 6). However if the lower and upper bound differ it is not clear whether the lower bound can be reached and in such a case all possible assignments have to be evaluated. The algorithm searching for the optimal assignment of the offsets α is in principle same as the hungry algorithm but uses backtracking to search for alternative solutions.

For a given b^i an candidate offset is found in the same manner as in the hungry algorithm (line 10). Then the upper border of the memory taken by the data of the variable b^i is on line 12 checked against the upper bound M_{best} and if violated, backtracking loop starts. On line 13 the variable is chosen to which the algorithm will backtrack. It is the conflicting variable which was assigned an offset as last. The idea behind this choice is, that two successive variables b^i and b^{i+1} doesn't necessary have to be in collision and therefore when repairing the inefficient solution, there is no point

in manipulating variables, which cannot overlap with the variable we are backtracking from. Also by manipulating any other then the most recently assigned could result in skipping potential solutions. The offset for new actual variable is increased by one and corrected on line 15 if an conflict occurs. The backtracking continues further when the new offset causes the variable to violate the upper bound in the loop condition on the line 12. The sensitivity of the algorithm on the value of partial solution at this place is increased with time as the upper bound is toughened with every found solution as described further.

Note that the backtracking is initiated already during the construction of the offset assignment, every time the upper bound M_{best} is reached by the value of the partial solution. Therefore thank to this bounding on the partial solutions the unfruitful branches are cut away at the very moment when some variable is assigned offset leading to uninteresting solution. Also every generated complete solution is therefore better than any previous and leads to an update of M_{best} resulting in even stricter pruning in further iterations.

Also the restart of the offset assignment after the generation of a complete solution is directed at the variable which was first to define the value of the solution (determined on line 26). Therefore time is not wasted by optimizing the tail of the solution while the value of the solution is defined by the offset of some earlier variable.

The search for optimal assignment of α is terminated on one of two distinct conditions. If the value of some generated solution reaches the lower bound M_{lb} on line 24 or if the all possible assignments has been evaluated. Latter condition is detected on line 17 when the first variable b^0 reaches the actual upper bound M_{best} implying that there is no space left for further optimization as it is not possible to backtrack from the first variable.

4.3.4 Linearizing the Fusions

The linearization of the particular fusion F has contrary to the linearization of DAG representing the whole mapped function significant influence on the performation because the order of the particular kernels within a fusion determines the volume of shared memory required by the function. However to calculate exactly how much shared memory the linearized fusion will need the variable offsets has to be computed. As the computation of these offsets is very demanding task it is not possible to evaluate every candidate linearization of particular fusion. The number of all linearizations of a DAG can be large and therefore it was necessary to approximate

the value of shared memory during the search. For this approximation the lower bound M_{lb} from equation 4.3.3 was chosen as for most cases it is equal the value of optimal allocation for the given linearization and because the difference introduced by the fragmentation is acceptable for an approximation. This little degree of fragmentation is implied by the facts: that the size of the fusion is limited and that thank to the low arity of the library functions the degree of particular vertices is also low. Therefore the DAG representing the fusion tends to be rather less branched and doesn't provide enough complexity to introduce large fragmentation.

The linearization algorithm presented on the listing 4.4 is based on basic recursive algorithm presented in [10]. It enumerates all possible linearizations and keeps only the one with lowest lower bound on the shared memory λ_{best}. The set Z denotes vertices of the fusion graph with in-degree zero, function $I : V \rightarrow \mathbb{N}$ stores the in-degree so that it is not necessary to actually modify the graph when removing processed nodes and p is the actual position in the linearization λ. The algorithm is initialized with Z_0 being the set of load vertices, I_0 computed from the fusion DAG and $p_0 = 0$.

Listing 4.4: Linearization of a fusion

```
1  function nce(Z, I, p,λ)
2    if p = |F| ∧ M_lb^{λ_best} > M_lb^λ
3      λ_best = λ
4    else
5      for every v ∈ Z
6        I' = I
7        for every (v, v_1) ∈ out(v)
8          I'(v_1) = I'(v_1) − 1
9        Z_v = {w, w ∈ out(v) ∧ I'(w) = 0}
10       Z' = Z \ {v} ∪ Z_v
11       λ(v) = p
12       nce(Z',I',p+1,λ)
```

Every iteration of the loop on line 5 represents a possible decision for a vertex on a position p which can be chosen from the vertices with already computed (i. e.already placed in the linearization) predecessors. When a vertex is placed in the linearization, it doesn't further block any vertices and is virtually removed from the DAG by lowering the in-degree of its successors in the loop on line 7. Then the set of zero in-degree Z' is updated and next position in the linearization is processed.

The complexity of this algorithm is in $O(n!)$ as noted by [10], however recall that the size of the fusion is limited and also the typical degree of a

node is low. Therefore there was no reason to enploy far more sofisticated algorithm presented also in [10] which improves the complexity to $n2^{2^n}$ as with the size of the problem limited by some constant, the behaviour of both algorithms would be almost identical.

4.3.5 Fusion Combinations

As a last step a subset of all candidate fusions implementations is to be chosen. Fusions in this subset must not include one vertex of the original DAG more than once and the total performance of the whole mapped function implementation is to be maximized.

This task can be seen as an instance of set covering problem. For a DAG $G = (V, E)$ and a set of fusions $F \subseteq 2^V$ a set S of candidate subsets is constructed as $S = F \cup \{\{v\}, v \in V\}$. Then a performance of the fusions implementations and the standalone elementary functions is predicted (see section 4.4) and will be refferenced as a function of predicted time $\tau : S \rightarrow \mathbb{R}$. The cover is then a set $C \subseteq S$ such that $\bigcap C = \emptyset$ and $\bigcup C = V$. The total time of the cover is then $\sum_{c \in C} \tau(c)$.

In the linear notation each set $U \in S$ is associated with a *binary* variable x_U representing the participation of the set in the cover. The requirement of empty intersection is enforced by the equations 4.1 and the cost function is reformulated as equation 4.2.

$$\sum_{U : v \in U} x_U = 1, \forall v \in V \tag{4.1}$$

$$\sum_{s \in S} \tau(s) x_s \tag{4.2}$$

The optimization variant of the set cover problem is known to be NP-hard however the linear programming formulation has $|V|$ equations and $|V| + |F|$ variables and can be solved for the pruned state space with the help of generic linear programming solver. See section 5 for more detailed performance evaluation.

As the performance prediction τ is not exact it is desirable to generate several fusion covers with similar expected performance and empirically evaluate them. For this purpose we iteratively restart the linear programming solver on modified set S for given number of times so that we obtain alternative solutions. The algorithm works as follows: when for a candidate set S a cover C is computed consisting of fusions $U_1 ... U_n$ a new candidate

sets $S_1...S_n$ are created by subtracting the particular members of the cover C from the original candidate set S.

The algorithm computing the fusions covers is presented on Listing 4.5 To avoid repetitive modifications of the set of candidate fusions the algorithm keeps track of the already processed modification of the original set. These modifications are represented by sets of fusions $N \subseteq 2^F$ which were on line 16 subtracted from the original set. The algorithm begins with a set of fusions F and empty modification set in the queue. In each iteration an optimal cover is computed on line 3 and afterwards new candidate set is created for every fusion in the cover by subtracting it from the candidate set. On these new candidate sets the resulting optimal cover will differ from all previously computed covers and at the same time any cover can be reached as the difference between two successive candidate sets is only one fusion. The algorithm pseudocode is presented in listing 4.5.

Listing 4.5: Algorithm for fusions covers computation

```
1  function computeCover (F)
2    formulate linear problem from F
3    solve linear problem to get fusion cover C
4    return C
5
6  function generateBestCovers(F)
7    enqueue((F,∅))
8    B = ∅
9    N = ∅
10   while( queueNotEmpty() ∧ |B| < ϵ )
11     (F′,M) = dequeue()
12     C = computeCover(F′)
13     if( C = ∅ ) continue
14     B = B ∪ {C}
15     for each U ∈ F′ such that M ∪ U ∉ N
16       F″ = F′ \ U
17       enqueue((F″, M ∪ U))
18       N = N ∪ M ∪ U
19   return B
```

4.4 Performance model

The size of the state space of possible mapped function implementations and the time costs of code generation, compilation and performance mea-

surement does not allow the empirical evaluation of every possible fusion implementation in practice. Instead an empirical model is used to predict the performance implication of decisions made during the fusing phase and only handful of most promising candidates are evaluated in reality. In this section the performance factors are presented followed by the description of the model used for the prediction.

4.4.1 Performance Factors

The performance of particular CUDA code on given GPU is determined by numerous factors. While some of them depend on the implementation of the particular function, some depend on the choices made during the fusion phase as described in section 4.2.3.

The developers are responsible for the performance of the standalone elementary functions during the decomposition phase. That means taking care for coalesced memory access, avoiding bank conflicts, choosing optimal granularity and avoiding warp divergence. These performance factors are not influenced in the composition phase as they all relate to the implementation of the particular function which is not modified during the composition phase.

There are however factors influencing the performance of the particular functions within a fusion which relate to the implementation of the fusion and are therefore determined in the composition phase. It is the total amount of the allocated shared memory in the fusion, the overhead introduced by the computation reorganization between function implementations operating on different granularity inside the fusion and the volume of spared traffic to the global memory.

The amount of shared memory and the number of threads are both fixed for the whole fusion as it is not possible to resize the thread block or allocate the shared memory in a dynamical manner. Although the idling threads do not limit the performance directly, the computation between two functions with different requirements on the thread block size has to be reorganized. This reorganisation consists in recomputation of the thread coordinates within a block and limiting the number of threads taking part in the execution. Both parts introduce an overhead – the position is recomputed with the help of costly modulo operations and the limitation requires additional branching resulting in serialization of some warps. These performance limitations are introduced by the fusion and have to be outweighed by the spared global memory traffic for the fusion to be gainful in the means of performance. A performance model incorporating these

counter-working factors is presented in next section.

4.4.2 Performance Prediction

The elementary functions stored in the library all share same template-like design separating the load and store routines from the computational part. Therefore it is possible to empirically evaluate the performance of these building blocks separately and to compute the overall performance of a fusion from these values.

The most important feature of the GPU which has to be taken into account is the ability to overlap the warps executing the computational instructions with the warps performing the memory operations. Therefore the basic estimation of the performance of particular fusion has to evaluate time spent both in the global memory operations and in computation and determine if the performance of the fusion is memory- or computationally-bound. To make this prediction the empirically estimated times of the load, store and computational routines of particular fused functions are used. The equation 4.3 has to be further refined to incorporate previously described performance limitations. The estimation of these performance factors is presented in the rest of the section.

$$t_{bound} = max(\sum t_{load} + t_{store}, \sum t_{compute}) \tag{4.3}$$

The statically allocated memory of a fusion is usually larger than the memory required by the particular fused functions. It is either necessary for storing the intermediate results or is required by some larger function and is left unused because it is not possible to deallocate it before the end of the fusion execution. The additional memory lowers the number of blocks which can be executed on the multiprocessor at one time and therefore lowers the attained parallelism of the particular fused functions execution compared to the standalone execution. However it is not necessary to empirically evaluate the performance of every fused function in every fusion. Instead the performance of all elementary functions in the library is evaluated once with respect to a particular GPU. We refer to this process as to the *benchmarking* of the library of elementary functions and it is described in next section. This association of timing to the additional allocation and number of elements can be also seen as table function $\psi : R \times \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{R}$ where R is the set of all routines of all elementary functions.

First the amount of additionaly allocated memory is computed for each used routine r of the fused function f during the performance evaluation as $M_r = M_{fusion} - M^f_{function}$. Then the time estimations t_{load}, t_{store} and $t_{overall}$

for particular routines in the formula 4.3 are replaced with the corresponding values given by ψ resulting in following equation, where the sets R_c^F and R_l^F denote the compute and load routines used in the fusion F and where e is the number of elements processed by one instance of the fusion:

$$t_{fusion} = max(\sum_{r_l \in R_l^F} \psi(r_l, e, M_{r_l}) + \psi(r_s, e, M_{r_s}), \sum_{r_c \in R_c^F} \psi(r_c, e, M_{r_c})) \quad (4.4)$$

There is one more factor limiting the performance beside the size of the allocated shared memory – the number of unused threads. Similarly to the shared memory allocation the required number of threads used per data element can vary through the fusion while it is not possible to start and end threads (i. e.resize the block) during the lifetime of a CUDA kernel. One possibility how to model the performance loss could be introduce the number of additionally issued threads as a parameter to the function ψ and to evaluate all possible combinations of the limitation of the parallelism by shared memory and by the unused threads. That would however render the empirical evaluation either very time consuming or infeasible depending on the size of the elementary function library.

The performance behaviour of the fusions with very different thread-to-data granularity functions implementations has shown to be rather complex. Therefore it was decided that for implementation combinations differing only little in the thread number requirements the overhead will be approximated by adding a time spent by executing the modulo operation. This value is modeled as multiplication of empirically chosen constant by the number of threads required by the corresponding fused function. However this model underestimates the significant loss of the performance for larger differences in the block size requirements and therefore such fusions are a priori discarded.

4.4.3 Acquiring the Input Values

As mentioned above the timing for all elementary functions in the library is empirically evaluated once for given GPU. The benchmarking is performed by small single purpose programs generated by the compiler to measure the performance of particular routines of the elementary functions. Every routine of every function is benchmarked on all combinations of the additionally allocated shared memory and the number of elements, both from previously defined ranges.The range of additionally allocated shared memory starts and zero and ends at chosen limit. If the size of the allocated memory

of a kernel is too large it fails to execute giving natural upper bound for the additional allocation. The results of this benchmarking are aggregated to the performance files and made available to the compiler. It is also not necessary to evaluate the whole range as the difference in performance caused by allocation of one float variable is very low. Also the number of possible combinations of the number of elements and the shared memory is quite large and the compilation and run times of single benchmark are in order of seconds. Therefore the range is covered stepwise with reasonably small step and during the fusion performance evaluation the required values of additionally allocated memory and number of elements are rounded to corresponding multiplies of the step used during the benchmarking phase.

Chapter 5

Experimental Evaluation

In this chapter the experimental evaluation of previously presented algorithms is presented. It is divided in three basic topics targeting the accuracy of the performance prediction of the fusion implementations, the real performance of generated implementations of the mapped function and the timing of the compiler execution.

The experiments were performed on single workstation equipped with a quad core Intel® Core™ i7 950 at 3.07 GHz, 6 GB RAM and a NVIDIA GeForce GTX 480.

As a testing input the example code presented previously in Listing 4.1 was used as it provides a state space of interesting size and also diversity in the size of particular functions and reasonably complicated computation. It will be refered to as *Function I* in this chapter, while the second and simpler code presented in Listing 5.1 will be refered to as *Function II*.

5.1 Medium-Grained Operations

In this section the comparison of the performance behaviour of medium-grained operations on different GPU architecture generations is presented. As will be shown later on, the space for a performance gain is on newer Fermi architecture even wider than before, moreover different implementations became most efficient for different sizes of the matrices compared to the previous architecture. This fact implies that for a given fusion the most efficient combination of the elementary functions implementations can change with respect to the particular GPU architecture and supports the usefulness of the usage of an auto tuning compiler when transfering the code between different hardware architectures. This experiment is analogous to the results presented in [7], however it additionally provides the comparison to latest GPU architecture.

The operation under test is the square matrix multiplication. On the x-axis is plotted the size of the matrix and on the y-axis the attained band-

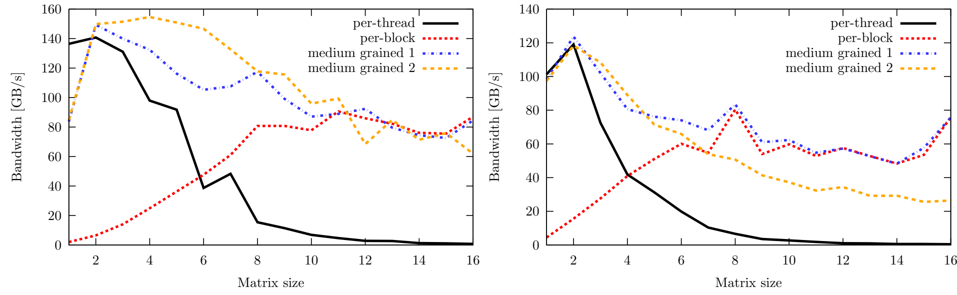

Figure 5.1: The performance of mapped matrix-matrix multiplications – left on GTX480, right on the GTX280.

width in GB/s. Four different implementations of the multiplication were measured: the fine-grained per-thread (one multiplication is computed by one thread), coarse-grained per-block (one thread computes one element of the resulting matrix) and two medium-grained. First medium-grained implementation processes the matrices analogously to the coarse-grained, however there are several multiplication computed in one block. Second medium-grained implementation uses one thread to compute one row of the resulting matrix and again computes multiple matrices in one block.

5.2 Prediction Accuracy

In this section the prediction of the performance of particular fusions is compared to real attained performance. For this purpose the compiler was modified to generate all possible fusions and several implementations of each fusion. For each implementation the value of predicted performance was stored to be compared with later measured real performance.

The real performance was measured by running the computation on randomly generated input lists of 31744 data elements repeatedly for 1000 times. A mean time needed for one data element was taken as a resulting value and the performance in processed data elements per second was derived from this result.

The results are presented in the Figure 5.2. The particular fusions are on the x-axis ordered descending by the real attained performance which is plotted on the y-axis together with the corresponding predicted value. As mentioned previously, the performance is in millions of processed data elements per second. As can be seen from the figure, the predicted performance is in most cases lower than the real attained performance, however

it copies the general trend of the real performance. There are however some fusions which are slightly underestimated at the beginning of the spectrum. This performance estimation error can be either product of suboptimal generation of the code, e. g. some redundant synchronization or thread block recalculation, or the fact, that for example necessary synchronization between the particular fused function calls is not taken into account in the performance prediction.

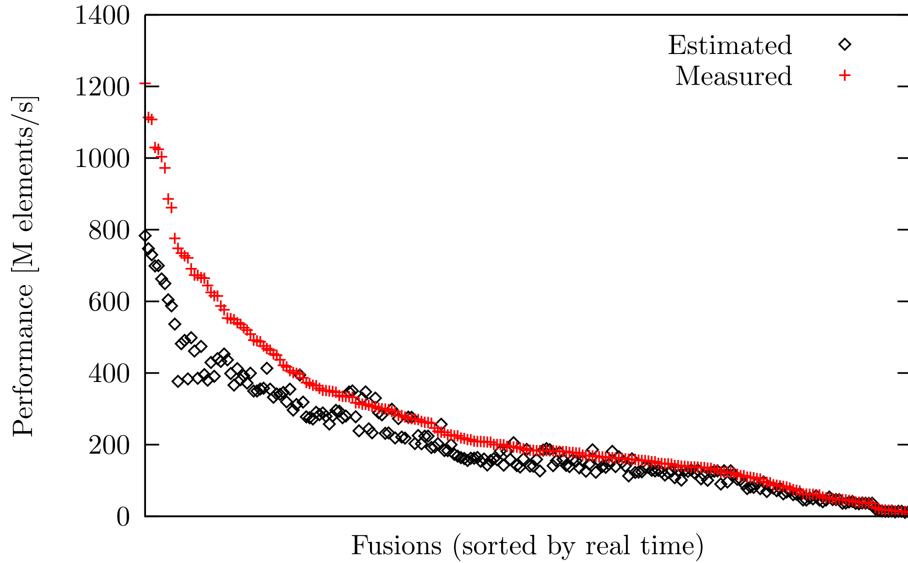

Figure 5.2: Prediction accurancy for fusions on GTX 480.

5.3 Performance of the Generated Code

The final result of the whole optimization process is the CUDA implementation of the mapped function. In this section the predicted performance of the generated implementations is confronted with the real attained performance. The number of best solution generated by the compiler can be arbitrary set. In the common use it should be around 50 to keep the empirical evaluation time reasonably low. However for the sake of this experiment the number was risen to 300 mapped function implementations to show the relation between the predicted performance and the real one also for the implementations marked by the compiler as less perspective.

As in to previous section, the measured value was the time of the com-

putation for one of 31744 the data elements for 1000 repetitions and the mean performance in the milions of data elements per second was derived.

The Figures 5.3 and 5.3 show the real performance of the first 300 selected mapped function implementations in the order of selection on two example mapped functions. The first function was already presented in Listing 4.1 representing the computation of the term $||A \cdot B \cdot v||_2 \cdot (C \cdot D + C)$ where A and B are 3×3 matrices, the v is a vector of size 3 and the C and D are 5×5 matrices. Second function is described in listing 5.1 and represents computation of the term $A \cdot B + A \cdot C + A \cdot D$ on the square matrices of size 5. Note that both functions are synthetic examples created fore the sole purpose of the testing and although these particular computations can be simplified on the algebraical level, it doesn't relate to the work presented here. In both figures there is a fitted curve plotted to better illustrate the overall trends. It is the polynomial of fifth degree fitted to the data by the least squares method. The least squares method is a standard approach to solving overdetermined system e. g. determination of the coefficients of the fitted polynomial.

Listing 5.1: Description of the mapped function

```
MATRIX5x5 A, B, C, D, M1, M2, M3, M4, F;

input A, B, C, D;
M1 = mmul55(A, B);
M2 = mmul55(A, C);
M3 = mmul55(A, D);
M4 = madd55(M1,M2);
F = madd55(M4, M3);
return F;
```

It can be observed from both figures, that the best implementations are selected among first and are even more or less sorted by the performance. In the case of the first function the fluctuation in the attained performance rises around the 30th selected implementation as the compiler begins to generate less effective implementations. This behaviour can be caused by underestimation of the performance degrading factors by the performance prediction. Also note, that this order is generated by the queue and therefore wouldn't be in principle strict even if the performance prediction was exact. Such a difference between the predicted and real performance can be caused by accumulated error of the performance prediction for particular fusions of which the mapped function implementation consists. It is also possible, that the NVIDIA nvcc compiler was able to perform the low-level

optimization better than in the case of neighbouring implementations and boosted the performance against the predicted value.

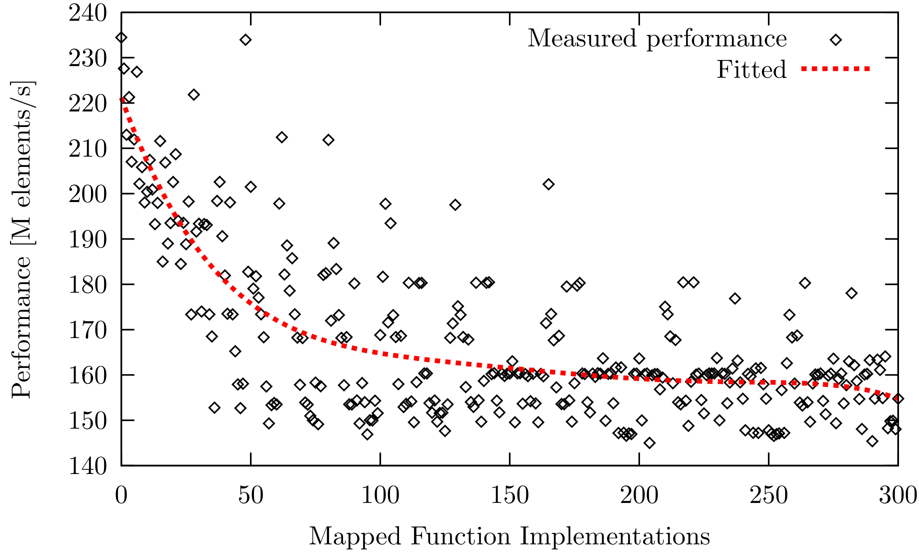

Figure 5.3: Real performance of the implementations in the order of selection by the compiler (i. e. by the predicted performance) of the computation from Listing 4.1.

In the case of the second function – with the results presented on Figure 5.3 – the improved accuracy is connected to the higher homogeneity of the tested function. Recall that all elementary functions used in this example operate on the matrices of the same size and therefore have the same consumption of the on-chip resources. This is also the reason for the stable level of performance in the middle of the range under observation.

The overview of the position of the five best solution in the previously shown generated order is presented in the Table 5.3. It can be seen that the best solutions can be found in very few iterations.

	position				
function I	0	48	1	6	28
function II	2	0	3	1	4

Table 5.1: Positions of the best five solutions in the generated order.

In the case of the function I the performance of the global memory

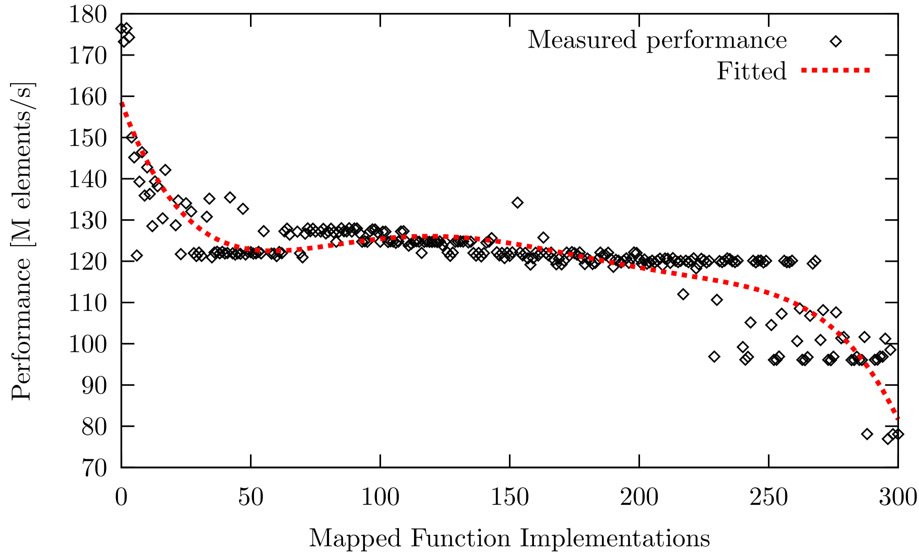

Figure 5.4: Real performance of the implementations in the order of selection by the compiler (i. e. by the predicted performance) of the computation from Listing 5.1.

implementation was **94.2** milions of elements per second while the best generated implementation achieved **234.5** Melems/s which is 2.5× better. Similarly in the case of the function II the global memory implementation yielded the performance of **68.5** Melems/s and the best generated implementation processed **176.5** Melems/s, again 2.6× better. This significant speedup justifies the effort invested in the second phase of the decomposition-fusion scheme.

5.4 Compiler Efficiency

In this section the running times of the substantial parts of the compiler functionality are presented. All timings were obtained during the processing of the function I. The time taken by the generation of the state-space (i. e. all fusion implementation candidates) is presented on the first row of the Table 5.4. Then the determination and CUDA code generation of the first 50 mapped function implementation is shown on row two. On the last row is presented the time for compilation of the CUDA code of a single function implementation by the NVIDIA nvcc compiler.

The running times of the state-space generation and cover determina-

state-space generation	0.21 s
cover determination and code generation	7.34 s
compilation of a candidate	2.08 s

Table 5.2: Positions of the best five solutions in the generated order.

tion leave enough space for processing of more complex mapped functions. If the running time would rise too quickly on large mapped functions it is possible to explicitly divide the computation of these functions and optimize the parts separately. Also note that the amount of time spent by compilation of the mapped function candidates can be easily adjusted by the number of these candidates generated as an output of the repetitive optimization process.

The compilation time of one benchmark instance is 1.77 s and the running time depends on the function implementation and routine under test, but oscillates around two seconds. Recall, that the number of benchmark instances[1] is relatively large, depending on the density of the parameter coverage and the number of implementations of the elementary functions. However the benchmarking is done only once per elementary function implementation on given GPU and the long benchmarking time is therefore acceptable as the elementary functions are supposed to be reused frequently.

1. combination of fusion implementation routine, number of elements and additionally allocated shared memory

Chapter 6

Conclusion

The main subject of this thesis was to define the state space of all possible implementations of a given mapped function and to devise and implement algorithms that would search this space for the most efficient implementation.

The state space was defined in detail, as was the complexity of the particular steps of its generation. Additionally, several heuristics were proposed to reduce its size and to make the search for an efficient implementation feasible.

As another result of this thesis, algorithms for searching the state space were devised and implemented. It was shown how the performance of the resulting implementation related to the choices made during the implementations process, and this knowledge was used to effectively predict the performance and to guide the automated generation of the resulting code. It is now possible to predict with reasonable precision the performance of a given implementation of given fusion and subsequently of the whole mapped function implementation.

Finally, these algorithms were implemented to the core of a source-to-source compiler, and the functionality of this implementation was proven and evaluated on a synthetic example. It was shown that the state space search was feasible and capable of interesting optimizations.

This implementation demonstrates and extends the usefulness of the medium-grained approach to the mapped function implementation, and moreover, it elaborates on and automates the fusion-decomposition scheme devised previously to guide the implementation of such mapped functions.

In future work, when the language accepted by the compiler is enriched by looping, further optimizations will be possible in the initial phase where fusible subsets are generated. Also, the accuracy of the prediction can be improved by incorporation of the synchronization, computation reconfiguration within a fusion, performance loss introduced by idle threads and more detailed modeling of the overlaping of the computation and memory operations. The implementation has been done with further development

in mind and is prepared for changes in the structure of the considered functions, and for modifications of the performance prediction.

Bibliography

[1] S.S. Baghsorkhi, M. Delahaye, S.J. Patel, W.D. Gropp, and W.W. Hwu. An adaptive performance modeling tool for GPU architectures. In *Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel computing*, pages 105–114. ACM, 2010.

[2] G. Belter, ER Jessup, I. Karlin, and J.G. Siek. Automating the generation of composed linear algebra kernels. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–12. ACM, 2009.

[3] M. Berkelaar, K. Eikland, and P. Notebaert. Lpsolve, Version 5.1.0.0, 2004. *URL http://lpsolve.sourceforge.net/5.5*.

[4] B. Catanzaro, N. Sundaram, and K. Keutzer. A map reduce framework for programming graphics processors. In *Workshop on Software Tools for MultiCore Systems*. Citeseer, 2008.

[5] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[6] J. Filipovič and J. Fousek. Medium-grained functions mapping using modern GPUs. In *Symposium on Application Accelerators in High-Performance Computing*, 2010.

[7] J. Filipovič and J. Fousek. Towards efficient implementation of mapped functions on GPUs. *Submited for publication to Parallel Computing*, 2011.

[8] J. Filipovič, I. Peterlík, and J. Fousek. GPU Acceleration of Equations Assembly in Finite Elements Method-Preliminary Results. In *Symposium on Application Accelerators in High-Performance Computing*. Citeseer, 2009.

[9] B. He, W. Fang, Q. Luo, N.K. Govindaraju, and T. Wang. Mars: a MapReduce framework on graphics processors. In *Proceedings of the*

17th international conference on Parallel architectures and compilation techniques, pages 260–269. ACM, 2008.

[10] C.W. Kessler. Scheduling expression DAGs for minimal register need. *Computer Languages*, 24(1):33–53, 1998.

[11] A. Klöckner, T. Warburton, J. Bridge, and J.S. Hesthaven. Nodal discontinuous Galerkin methods on graphics processors. *Journal of Computational Physics*, 228(21):7863–7882, 2009.

[12] E. L. Lawler and D. E. Wood. Branch-and-bound methods: A survey. *Operations Research*, 14(4):pp. 699–719, 1966.

[13] G.R. Markall, D.A. Ham, and P.H.J. Kelly. Towards generating optimised finite element solvers for GPUs from high-level specifications. *Procedia Computer Science*, 1(1):1809–1817, 2010.

[14] J. Matoušek and B. Gärtner. *Understanding and using linear programming*. Springer Verlag, 2007.

[15] NVIDIA. Compute Unified Device Architecture Programming Guide version 2.3. 2009.