Automatic Fusions of CUDA-GPU Kernels for Parallel Map

Jan Fousek
Faculty of Informatics
Masaryk University
izaak@mail.muni.cz

Jiří Filipovič*
Faculty of Informatics
Masaryk University
fila@ics.muni.cz

Matúš Madzin
Faculty of Informatics
Masaryk University
gotti@mail.muni.cz

ABSTRACT

When implementing a function mapping on the contemporary GPU, several contradictory performance factors affecting distribution of computation into GPU kernels have to be balanced. A decomposition-fusion scheme suggests to decompose computational problem to be solved by several simple functions implemented as standalone kernels and to fuse some of these functions later into more complex kernels to improve memory locality. In this paper, a prototype of source-to-source compiler automating the fusion phase is presented and the impact of fusions generated by the compiler as well as compiler efficiency is experimentally evaluated.

1. INTRODUCTION

In this paper, we address the optimization of the implementation of function mapping by optimizing the distribution of the computation into kernels. We call the function f is mapped to lists of input elements $L_1, .., L_n$ when f is applied element-wise to $L_1, .., L_n$ producing list of output elements. Formally $map(f, L_1, .., L_n) = [f(l_{1,1}, .., l_{n,1}), .., f(l_{1,m}, .., l_{n,m})]$, where $l_{i,j}$ is j-th element of L_i. The function f is mapped to particular elements independently, thus it can be massively parallelized. However, it can be difficult to implement the mapped function efficiently as it is necessary to find efficient distribution of the solved task into kernels.

CUDA GPUs use several levels of explicitly managed memories, which differ in their size, latency, bandwidth and lifetime. We distinguish between two major types of memory – off-chip and on-chip. The off-chip memory has usually greater size, longer latencies, lower bandwidth and a lifetime of the application. The on-chip memory is smaller and faster, thus it is used to store data which is accessed frequently during the computation. It has also restricted lifetime, thus the on-chip memory can be used only for intermediate data used during GPU kernel execution and data passed from one kernel to another have to be stored in off-chip memory.

The instruction throughput of the GPUs is significantly higher than the off-chip memory throughput. Thus, it is necessary to perform enough operations per data passed from or to the off-chip memory within single kernel to exploit the computational power of the GPU. However, it is not always beneficial to put more computation into a single kernel. During the computation, the size of processed data as well as number of threads which can efficiently process data in parallel vary significantly. If it happens within single kernel, there is risk that the achievable parallelism is reduced in part of the kernel computation which can decrease the overall performance. Moreover, the larger amount of computation within single kernel can consume more on-chip memory by intermediate data, which can reduce parallelism too as the size of on-chip memory is limited.

Because putting more computation into single kernel can bring performance benefit due to reducing off-chip bandwidth pressure as well as performance loss due to reduced parallelism, finding the optimal distribution of computation to kernels can be difficult.

To handle the problem of distributing computation into kernels, we have suggested a decomposition-fusion scheme [3]. The whole function is decomposed into several elementary functions and implemented as standalone kernels, which are often memory-bounded, as they have usually low computations to memory transfers ratio. After that, sets of these functions, which can perform better in one kernel, are fused.

This scheme brings several advantages. The elementary functions are easier to implement and often reusable, as they can be more general compared to the more complex ones. To develop more complex kernels, the simple functions are fused (which basically means that their computation parts using data in on-chip memory are serialized), which is easier and less error-prone that manually developing complex functions from scratch.

The kernels fusions are not provided by general compilers, as it is very hard to maintain code correctness of fused code (e. g.there is implicit global barrier between kernels invocations and the thread-to-data arrangement can be different in different kernels). On the other hand, with scheme proposed in [3] and extended in this paper it is possible to develop a source-to-source compiler, which is able to automatically optimize sequences of simple functions by fusing them.

The contributions of this paper are as follows:

- we define the space of all possible fusions and algorithm used to traverse and prune it
- we introduce performance prediction method allowing us to search for implementation available in pruned fusions space which should have the highest performance

*The financial support provided by Ministry of Education, Youth and Sport of the Czech Republic under the research program MSM0021622419 is acknowledged.

- based on these observations, we introduce prototype of source-to-source compiler automatizing generation of code with fusions
- we evaluate the performance of the generated code and the compiler efficiency

2. RELATED WORK

Our original motivation comes from finite element analysis, where the same computations are mapped to many elements of mesh during assembly of equations. In the literature, several GPU acceleration of per-element computations can be found [6] [7] [8] with manually optimized code to kernels distribution. The manual optimization is possible, because concrete computational problem is solved [6] [7], or several quite complex kernels are developed to solve specific class of problems [8]. We have introduced the fusion of elementary functions in [3], however, without detail study on searching for most the effective fusions.

The fusions of GPU kernels are studied in several papers from different fields. The GPU implementation of MapReduce fuses map kernels [1] exchanging data in on-chip memory. However, all functions are always fused here as there cannot be significant differences in on-chip memory usage and single data element is always processed by one thread. The more heterogeneous fusions are allowed in Thrust library [4], which allows programmer to explicitly fuse memory-bounded functions into single kernel. The additional value of research presented in our paper is automating the decision which kernels should be fused to maximize performance.

3. COMPILER DESIGN

The complex mapped function is decomposed to several *elementary* functions and is solved by sequence of their calls. The compiler works with *library* of such elementary functions and simple user *high-level code* which describes sequence of their calls. It is possible to execute elementary functions as standalone kernels, or fuse some subsets of them into more complex kernels. The compiler searches the space of possible fusions and fuse the functions when it is expected to improve the performance. The compiler's results can be improved by automatic empirical evaluation of several implementations of the complex function with the highest expected performance.

The current implementation of the compiler introduces several restrictions for the elementary functions design and their fusion.

All elementary functions are expected to have one or more inputs and single output element. They have to be implemented as call of loads (separate for each input element), compute and store routines. This eases the fusion generation, as when some functions are fused, the stores and loads for elements that resists in on-chip memory are omitted and the rest of calls are glued into single kernel (see Figure 1).

Another simplification restricts on-chip memory allocation such that all data elements are expected to be stored in shared memory. Thus, the memory consumption of the fusion can be determined easily and functions with different parallel granularity (number of threads solving single instance) can be fused without difficult data rearrangement.

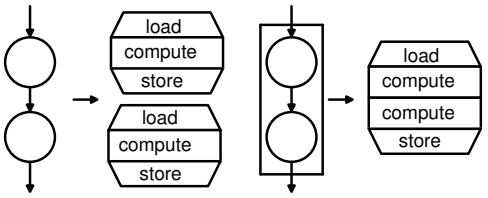

Figure 1: Example of simple fusion.

The code generator has to maintain correct on-chip memory allocation, computation of thread coordinates (when some fused functions runs in different parallel granularity) and synchronizations.

However, these simplifications bring some limitations too. First, as the fusion is performed by omitting loads and stores and does not affect compute routines (e. g. performing loop fusions). Second, the data elements used by functions have to be small enough to be processed in single load(s)-compute-store sequence (the only way to process larger data elements which require iterative loading and computing is to divide them to smaller elements). Third, as only shared memory can be used to store processed data elements, the performance can be reduced, because registers cannot reduce shared memory usage and because some computations can be faster using registers [9].

4. AUTOMATIC FUSIONS GENERATION

There is a large number of possibilities how to generate fusions of elementary functions. The subset of elementary functions that is to be fused is called *fusion*. The concrete implementation of a fusion (i. e. the form that the CUDA code can be unambiguously generated from) is called *fusion implementation* and the selection of fusion implementations and standalone functions that together solve the complex function is called *fusions combination*.

In this section, we describe the space of possible fusions, their implementations and combinations of fusion implementations. As the space is large, we introduce several methods to prune it and predict the performance of the remaining fusions implementations.

4.1 Example of Mapped Function

Example Listing 1 shows high-level description of mapped function $f : F = ||A \cdot B \cdot v||_2 \cdot (C \cdot D + C)$ 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. Recall that this code describes the computation of a single instance of the function (i. e. all input and output variables are single elements of input and output lists of the *map* function) and multiple instances are applied to multiple elements in parallel by *map*.

We note that this function is artificial and not optimized from algebraic point of view – it is example of function where the size of data elements and exploitable parallelism varies significantly, thus the searching for efficient fusions is difficult here. As you can see, the

original function has been decomposed to elementary functions with explicitly named and typed intermediate results. First four rows declare the types of the used variables. Lines 6 and 15 denote the input and output arguments of the mapped function.

Listing 1: Example description of the mapped function

```
1 MATRIX3x3 A, B, M1;
2 MATRIX5x5 D, E, F, M2, M3;
3 VECTOR3 c, v1;
4 SCALAR s1;
5
6 input A, B, c, D, E;
7
8 M1 = mmul33(A, B);        // M1 = A · B
9 v1 = mvmul33(M1, c);      // v1 = M1 · c
10 s1 = venorm3(v1);        // s1 = ||v1||2
11 M2 = mmul55(D, E);       // M2 = D · E
12 M3 = madd55(M2, D);      // M3 = M2 + D
13 F = smmul55(M3, s1);     // F = M3 · s1
14
15 return F;
```

Most of used functions are implemented with multiple parallel granularities: single instance of *mmul33* can be solved by 1, 3 or 9 threads, *mvmul33* by 1 or 3, *mmul55*, *madd55* and *smmul55* in 1, 5 or 25 threads. We note that shared memory access in these functions must be implemented carefully to minimize bank conflicts between their individual instances (see [2] for more details).

We will use this example throughout the rest of the paper to illustrate the particular steps of the optimization process and code generation.

4.2 Degrees of Freedom

There are several degrees of freedom in fusing given decomposition of mapped function. The computation is defined by direct acyclic graph (DAG) $G = (V, E)$, where vertices V represent simple functions calls and edges E represent data dependency between functions.

The space of all possible fusions consists of:

- all fusible subgraphs of the DAG
- all linearizations of particular fusions
- all implementations of elementary functions within fusion
- all valid (non-overlapping) combinations of the fusions implementations

These degrees of freedom are elaborated in more detail in following paragraphs.

4.2.1 Fusible Subgraphs

The subgraph $G_F \subseteq G$ can in general be fused, if there is no outgoing edge from G_F such that there is a path beginning with this edge and returning back to the component in G_F. Such a path needs to compute some data outside the fusion but during fusion computation, which cannot be efficiently performed with current GPU synchronization mechanisms. In our example in Section 4.1, this situation occurs for example if we try to fuse $A \cdot B$ and $||\ ||_2$, as to compute $||\ ||_2$, we need to compute whole $A \cdot B \cdot v$ and the last multiplication is not included in the fusion.

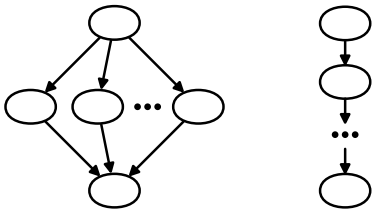

Figure 2: Left: the worst case structure of a DAG. Right: the easiest structure.

Although the fusible subgraphs are limited by this condition, the number of such subgraphs for more complex functions can grow quickly. In general the number of possible fusions on a given $G = (V, E)$ is in the worst case near the number of subsets of all vertices, i. e. $\mathcal{O}(|2^V|)$ (see Figure 2 left). On the other hand the amount of possible fusions is $\mathcal{O}(|V|^2)$ when the structure of the DAG is linear as depicted on the Figure 2 on the right.

4.2.2 Linearization

To translate a DAG representing the fusion subgraph to the CUDA code a linearization has to be made. A linearization of a DAG $G_F = (V, E)$ is an ordering on the vertices such that for every vertex there is no edge leading to previous vertex. There can be obviously multiple possible linearizations for a given DAG corresponding to different order of execution of data-independent kernels. The number of possible linearizations of a fusion is highly dependent on the corresponding DAG complexity and can reach up to $|V|!$ different linearizations (e. g. the worst case structure on Figure 2). The execution order can affect the amount of used on-chip memory and hence the parallelism reduction. Thus, the space of possible linearizations has to be searched for the one with lowest memory consumption. An example of linearization and corresponding allocation is depicted on Figure 3.

4.2.3 Implementation of Elementary Functions

Every elementary function can be present in the library in multiple distinct implementations, which differ for example in parallel granularity (using only functions with the best performance in unfused form is generally insufficient as some of them can perform worse in fusion, e.g. because of higher sensitivity to parallelism reduction). Therefore for every vertex in a G_F some function implementation has to be chosen. Let the fusion contain n calls of functions, where $\#f_i$ denotes number of implementations of i-th called function. The number of possible implementations is $\prod_{i=1}^{n} \#f_i$.

4.2.4 Fusions Combinations

Finally multiple fusions 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.

For example the previously defined function f can be implemented as $\{||A \cdot B \cdot v||_2\} \cdot \{(C \cdot D + C)\}$, or $||\{A \cdot B\} \cdot v||_2 \cdot \{(C \cdot D + C)\}$ etc, where $\{f_1, .., f_n\}$ expresses that $f_1, .., f_n$ are fused.

4.3 Search Space Pruning

4.3.1 Generation of Valid Fusions

The space of all valid fusions is large, however, it can be pruned. We can discard fusion if it does not form a connected component of the graph G. This condition prunes the fusions that cannot reduce memory bandwidth – in example presented in Section 4.1, if we fuse $A \cdot B$ and $C \cdot D$, there is no significant performance benefit as these two multiplication do not reuse any data.

The more important observation is that the size of fusions (number of fused vertices) can be limited by a chosen constant. It is important to realize, that the performance gain of the fusion stagnates for large number of fused vertices so it is possible to consider only fusions of limited size. The stagnation is caused by increasing the ratio of the computations to off-chip memory transfers, thus the large fusions become computationally bounded and adding more functions into the fusion does not improve performance. Moreover, in many cases of the large fusions the amount of required on-chip memory and varying parallelism reduces the performance more significantly than the spared memory transfers can increase. 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 in worst case: $\sum_{i=1}^{k} \binom{|V|}{i}$.

The reduction of maximal fusion size decreases significantly the complexity of the following space exploration. First, it reduces the overall number of the fusions and second, the following steps are performed on fusions with size limited by constant, thus even exponential algorithms are feasible.

4.3.2 Linearizing the Fusions and Memory Allocation

The linearization is chosen to minimize expected on-chip memory requirements. The space of all linearizations is explored and the linearization with lowest bound of memory consumption is selected.

The algorithm generating linearizations is based on basic recursive algorithm presented in [5]. It enumerates all possible linearizations in a recursive manner and keeps only the one with lowest bound of consumption of the on-chip memory. The complexity of this algorithm is in $\mathcal{O}(|V|!)$, however recall that the size of the fusion is limited, thus there is no significant benefit in using more sophisticated algorithm with complexity $\mathcal{O}(|V|2^{2^{|V|}})$ presented also in [5].

While an elementary function f is executed within a fusion, all elements processed by f have to be stored in the on-chip memory as well as all elements that are produced by any function before f and consumed after f. The lower bound of memory consumption of the whole fusion is a maximum of memory requirements during all executions of functions within the fusion. Recall that only shared memory is currently used by the compiler and it is used in continuous blocks for all elements, thus the fragmentation of the memory can occur and the fusion with particular linearization can need more memory, that is expected by using the lower bound. In current implementation, we select linearization only comparing their lower bound of memory consumption. It is possible to compute the memory requirements of each linearization exactly, however, it is more computationally expensive and in fusion of bounded size it seems the fragmentation does not occur frequently.

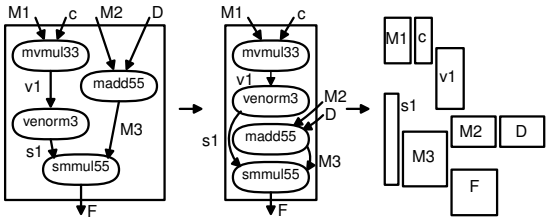

Figure 3: DAG - linearization - allocation

To determine exact allocation of elements in the shared memory, a branch and bound algorithm is used. The branching part of the algorithm moves elements offset in the memory, the bounding part discards allocations which exceeds memory requirements of the best already explored allocation. The algorithm ends when lower bound is found or when all possible allocations are explored. The time complexity is $\mathcal{O}(m^n)$, where n is number of elements and m is sum of memory requirements of all elements, as there is no reason to place any element in offset larger than m. The n is bounded by maximal fusion size, the m is bounded by need of usage of small elements due to small on-chip memory.

4.3.3 Fusions Combinations

As a last step a subset of all candidate fusions implementations is to be chosen. Fusions in this subset must include all vertices of the original DAG and no vertex of the original DAG can be included more than once. The total predicted 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 $G = (V, E)$ and a set of fusions implementations F we define function $\nu : F \to 2^V$ representing the nodes covered by given fusion implementation. Additionally we denote I_v the most efficient implementation of the elementary function v: $\nu(I_v) = v$. A set S of candidate subsets is constructed as $S = F \cup \{I_v, v \in V\}$. Then a performance of the fusions implementations and the standalone elementary functions is predicted and is referenced as a function of predicted time per function instance $\tau : S \to \mathbb{R}$. The cover is then a set $C \subseteq S$ such that $\bigcap\{\nu(c), c \in C\} = \emptyset$ and $\bigcup\{\nu(c), c \in C\} = V$. The total time of the cover is then $\sum_{c \in C} \tau(c)$.

In the linear programming 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 1 and the cost function is reformulated as equation 2.

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

$$\sum_{s \in S} \tau(s) x_s \tag{2}$$

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 is solved for the pruned fusions space, thus is feasible.

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 restarting works as follows: when for a candidate set S a cover C is computed 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. These new candidate sets are used if have not been generated previously. Using this algorithm any cover can be reached as the difference between two successive candidate sets is only one member of C.

4.4 Performance Prediction

The main idea of the performance prediction of the fusion implementation is to sum running times of particular functions used in the fusion. As the elementary functions share same template-like design separating the load, store and computation routines, it is possible to benchmark this routines separately. However, the parallelism in the fusion can be reduced (as in the fusion larger amount of shared memory per function instance can be allocated) as well as different block sizes can be used (i. e. different number of function instances can be processed within single block). Thus, the routines are benchmarked for certain ranges of reduced parallelism and block sizes. The performance is predicted separately for memory transfers and computation.

The GPUs are able to overlap the computation and memory transfers. In current implementation, we expect there are no limitation in overlapping and the function performance is determined by the performance of slower one of memory transfers and computations.

The association of timing to the additional memory allocation and number of function instances processed within one thread block can be also seen as table function $\psi : R \times \mathbb{N} \times \mathbb{N} \to \mathbb{R}$ where R is the set of all routines of all elementary functions.

First the amount of additionally allocated memory is computed for each used routine r of the fused function f in a fusion F during the performance evaluation as $\Delta M_r = M_F - M_f$. As some fused functions can require different thread granularity per instance, the thread coordinates have to be recomputed in such fusions, which can require time expensive integer divide and modulo operations. Thus, the t_{md} denotes the time spent on thread recomputation needed to process single function instance. Then the running time of single instance is estimated by Equation 3, where sets R_c^F denotes compute routines and R_m^F denotes load/store routines used in the fusion F and i denotes number of function instances processed by single thread block. As the amount of shared memory used per instance is already known at this point, the only i which does not cause total size of shared memory exceeding can be assumed.

$$\tau(F) = max \left(\begin{array}{c} \sum_{r_m \in R_m^F} \psi(r_m, i, \Delta M_{r_m}), \\ \sum_{r_c \in R_c^F} \psi(r_c, i, \Delta M_{r_c}) + t_{md} \end{array} \right) \quad (3)$$

5. EXPERIMENTAL EVALUATION

In this chapter the experimental evaluation of our compiler is presented.

The experiments were performed on single workstation equipped with Intel® Core™ i7 950, 6 GB RAM and NVIDIA GeForce GTX 480 with CUDA Toolkit 3.1 and Linux Driver 256.40.

As a testing input the example code presented in Listing 1 has been used as it provides a state space of interesting size and also diversity in the size of processed data elements and reasonably complicated computation.

5.1 Prediction Accuracy

In this section, the predicted and real performance of fusions implementation is compared. For this purpose the compiler has been modified to generate all possible fusions and several implementations of each fusion. The real performance was measured by running 31744 instances of mapped function repeatedly for 1000 times. The resulting performance is expressed as a mean number of computed output elements per second.

The results are presented in the Figure 4. 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 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. The overall underestimation can be explained by a systematical error in the benchmarked performance of the library functions, e. g. an influence of the kernel invocation is not negligible compared to the execution time. Further the performance estimation error fluctuation observable through the whole spectrum can be either product of varying ability of nvcc to optimize sequences of compute routines or reduced ability of overlapping memory transfers and computation.

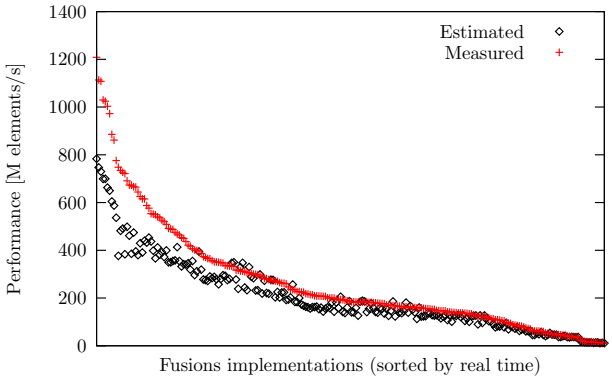

Figure 4: Prediction accurancy for fusions implementations.

5.2 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 expected best solutions generated by the compiler can be arbitrary set. For the sake of this experiment the number was set to 300.

The Figure 5 shows the real performance of the first 300 selected mapped function implementations in the order of selection. In the figure 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.

It can be observed, that the best implementations are selected among first and are even more or less sorted by the performance. The overall best five implementation are generated on positions **0**, **48**, **1**, **6** and **28**, so the best solutions are found in very few iterations. We note that this example is quite difficult for performance prediction as the size of processed data as well as parallel granularity varies significantly during computation. For more regular example, the performance prediction is far more exact.

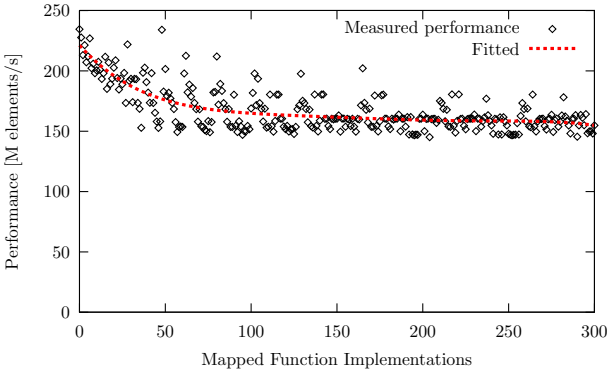

Figure 5: Real performance of the generated implementations in the order of predicted performance.

The best generated implementation uses fusion of $||A \cdot B \cdot v||_2$ which is performed by 3 threads per instance (for the norm computation, the parallelism is reduced to 1 thread per instance) and $\cdot(C \cdot D + C)$ using 5 threads per instance. It achieve **234.5** millions of resulting elements per second which is **2.49**× better that the best implementation without fusions (**94.2** Melems/s) and **1.46**× better than the best implementation when all functions are fused into single kernel (**160.2** Melems/s).

5.3 Compiler Efficiency

The time taken by the generation of the state-space (i. e. all fusion implementation candidates) is **0.21 s**. The cover determination and CUDA code generation of the mapped function implementation is **0.15 s** and it take **2.08 s** to compile single implementation.

The running times of the state-space generation and cover determination leave 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.

6. CONCLUSION AND FUTURE WORK

In this paper, we have introduced an optimizing source-to-source compiler, which is able to automatically fuse kernels of mapped functions and to generate implementations with good performance. The experimental evaluations shows feasible running times of the compiler as well as its good ability to quickly find effective implementations.

We plan to develop more detailed performance prediction model which enables to estimate the computation and memory transfers overlap.

The code generation can be improved by allowing elementary functions to exchange data via registers when it is possible, which brings also quite challenging changes in the performance prediction.

We also plan to employ the compiler on more complex applications such as Finite Element Method equations assembly, which was our initial motivation.

7. REFERENCES

[1] 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.

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

[3] 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*, 2009.

[4] J. Hoberock and N. Bell. Thrust: C++ template library for cuda. http://code.google.com/p/thrust/, 2010.

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

[6] 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.

[7] D. Komatitsch, D. Michea, and G. Erlebacher. Porting a high-order finite-element earthquake modeling application to nvidia graphics cards using cuda. *Journal of Parallel and Distributed Computing*, 69(5), 2009.

[8] 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.

[9] V. Volkov and J. W. Demmel. Benchmarking gpus to tune dense linear algebra. In *Conference on Supercomputing (SC08)*. ACM/IEEE, 2008.