Automatic Fusions of CUDA-GPU Kernels for Parallel Map

Jan Fousek izaak@mail.muni.cz

Jiří Filipovič fila@mail.muni.cz

Matúš Madzin gotti@mail.muni.cz

Masaryk University, Czech Republic

Motivation

Studied problem The GPU implementation of the function which is applied element-wise to the list of elements is studied. Despite easy parallelization, it is difficult to find efficient code-to-kernels distribution.

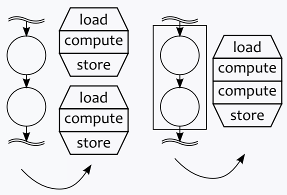
The goal To determine the efficient distribution of a computation of a mapped function into GPU kernels to balance memory locality and parallelism reduction.

Approach A proposed decomposition-fusion scheme suggests to decompose computational problem to be solved by several simple functions and some of these functions later automatically fuse into more complex kernels to improve memory locality.

Contribution We present a source-to-source compiler automating the fusion phase and the search for the most efficient implementation.

Elementary Functions

The basic building bricks are the elementary functions – simple hand tuned kernels. They conform to the load/compute/store template to enable automated fusion of several compute routines exchanging the intermediate results via shared memory.



Multiple implementations of every elementary function are made with different performance characteristics.

Pros:
- easily fusable into complex kernels
- reusable
- easy to implement

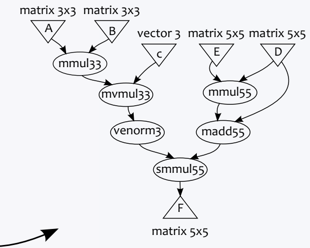
Cons:
- often memory bounded, fusions needed for better efficiency
- many possible fusions

Compiler Input

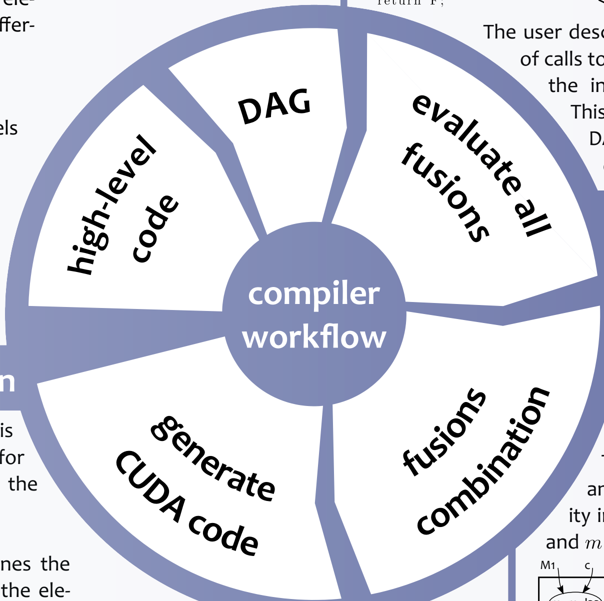
```
// code for f : F = ||A · B · c||_2 · (D · E + D)
MATRIX3x3 A, B, M1;
MATRIX5x5 D, E, F, M2, M3;
VECTOR3 c, v1;
SCALAR s1;
input A, B, c, D, E;

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;
```



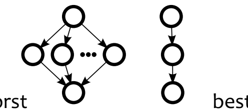
The user describes the mapped function as a sequence of calls to the elementary functions which exchange the intermediate results via off-chip memory. This sequence is then parsed into a data flow DAG with function calls as vertices and data dependencies as edges.



Fusions

To decrease the preassure on the off-chip bandwidth, several elementary functions can be fused into one kernel and exchange the intermediate results via on-chip.

A subgraph G_F of the graph $G = (V, E)$ can be fused only if there is no outgoing edge from G_F such that there is a path beginning with this edge and returning back to the G_F. The number of such subgraphs is in $\mathcal{O}(|V|^2)$ in the best case and in $\mathcal{O}(|2^V|)$ in the worst case.

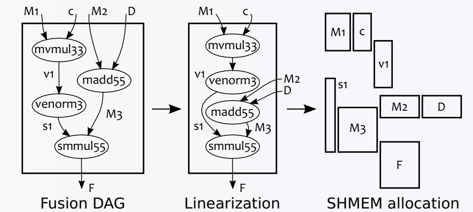


To overcome the high complexity of the algorithms performed on every fusion, the maximal size of a fusion is bound by a constant k, thus limiting the nuber of subgraphs to: $\sum_{i=1}^{k} \binom{|V|}{i}$.

Memory Allocation

As the GPU doesn't allow dynamical shared memory allocation, we have devised a memory reusing scheme. One large block is allocated and the variables are associated with offests in this space.

To determine the optimal offsets a branch and bound algorithm is used with complexity in $\mathcal{O}(m^n)$ where n is number of elements and m is sum of their size in memory.



Fusion Implementations

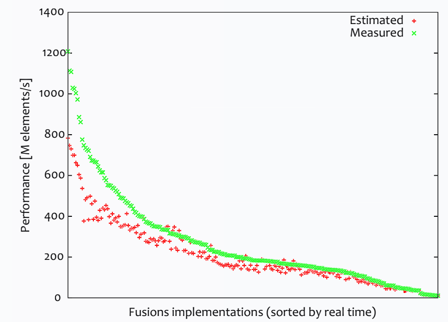
For every fusion there are several variants of translation to the CUDA code differing in:

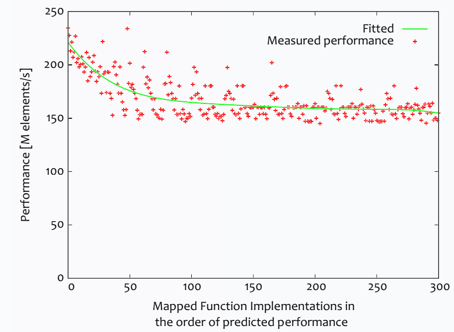
- the linearization of the fusion DAG
- the choice of the elementary functions implementations
- the number of elements processed by one block

Linearization The linearization enumerating algorithm runs in $\mathcal{O}(|V_F|!)$ and only the implementations of the fusion linearization with best lower bound on the on-chip memory consumption are further considered.

Granularity Every elementary function can process the data elements in different granularity. However the number of data elements processed by one fusion kernel has to be constant and therefore the number of active threads can vary throughout the execution.

Experimental evaluation





Performance Prediction

The running time for every fusion is predicted to enable the search for most efficient implementation of the whole mapped function.

The performance prediction combines the empirically evaluated behaviour of the elementary functions and the characteristics of a given fusion.

The load, compute and store routines of all elementary functions are benchmarked for certain ranges of parallelism reduction by additionaly allocated shared memory and different number of elements processed per block and represented by table function:

$$\psi(routine, elems, \Delta SHMEM)$$

To model the GPU capability of overlapping the computation and memory transfers all memory transfer and compute times of all elementary functions are summed separately:

$$\tau(F) = max \begin{pmatrix} \sum_{r_m \in R_m^F} \psi(r_m, i, \Delta M_{r_m}), \\ \sum_{r_c \in R_c^F} \psi(r_c, i, \Delta M_{r_c}) + t_{md} \end{pmatrix}$$

Compiler efficiency

Generation of the state-space:	0.21 s
CUDA code generation:	0.15 s
Compile time of the generated code:	2.08 s

The best generated implementation speedup was **2.49×** over the unfused version and **1.46×** over the implementation with all kernels fused.

Combinations of Fusions

As a last step the most efficient subset of all candidate fusions implementations and standalone kernels U is to be chosen. This task can be formulated as a set covering problem and solved by the linear programming.

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

$$\sum_{s \in S} \tau(s) x_s \quad \text{minimize}$$