Automatic Fusions of CUDA-GPU Kernels for Parallel Map

Motivation

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 fusible into complex kernels
- reusable
- easy to implement

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

Elementary Functions

Compiler Input

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 pressure 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 $O(|V|^2)$ in the best case and in $O(|V|^3)$ in the worst case.

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 $O(|V||E|)$ 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 the 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 additionally allocated shared memory and different number of elements processed per block and represented by table function:

$$\psi_{\text{routine}}(i, m, \Delta M, t_{\text{md}})$$

$$\tau(F) = \max_{r \in R} \left( \sum_{i \in I} \psi_{\text{routine}}(i, m, \Delta M, t_{\text{md}}) \right)$$

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 speed-up was 2.49x over the unfused version and 1.46x 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.

$$\sum_{x \in V} x_v = 1, \forall v \in V$$

$$\sum_{x \in \mathcal{S}} c(s) x_v \rightarrow \text{minimize}$$

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 offsets in this space.

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