Medium-grained functions mapping using modern GPUs

Jiří Filipovič
Masaryk University
fila@ics.muni.cz

Jan Fousek
Masaryk University
izaak@mail.muni.cz

***Abstract*—The *map* is a higher-order function that applies a given function to the list or lists of elements producing the list of results. The mapped function is applied to each element of the list independently, thus can be performed for all elements in parallel, making the GPU an interesting platform to be implemented on.**

Although the map introduce a high level of parallelism when it is applied to sufficiently large number of elements, its implementation can be difficult with respect to utilizing GPU parallel model granularity by mapped functions.

In this paper, we show the performance gap between fine-grained (per-thread) and coarse-grained (per-block) implementation of mapped function and introduce the medium-grained implementation that can fill this gap. We also discuss some memory access implications arising from this method and show example how to use them to estimate the performance of different implementations.

I. Introduction

The modern GPUs need to run many threads concurrently to be used efficiently. To met this requirement, accelerated tasks must expose enough parallelism, which can be achieved by running a function working on large data elements, or by mapping the function to many smaller elements in parallel.

In this paper, we focus on mapping the function to the list of relatively small data elements. We call that n-ary function f is *mapped* to lists $L_1, .., L_n$, when the f is applied element-wise to $L_1, .., L_n$ producing a 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.

We focus on NVIDIA CUDA platform in this paper as it is a first and broadly used general purpose architecture and programming model of GPUs. It is out of the scope of this paper to describe basic CUDA concepts, which can be found in NVIDIA CUDA Programming Guide [1]. The straightforward use of CUDA threading model suggests to run one instance of mapped function (i.e. producing single output) in two possible granularities:

- *fine-grained*, when function is performed by one thread
- *coarse grained*, when function is performed by all threads within one thread block

To use a GPU efficiently, it is important to utilize its fast on-chip memory (mainly registers and shared memory) for the intermediate data used during computation. However, the on-chip memory is shared by all threads running on multiprocessor and has restricted size, thus it can reduce the parallelism and consequently overall GPU performance when it is used extensively. The functions with small on-chip memory requirements can be implemented to be solved by one GPU thread, whereas the function with higher memory requirements has to be parallelized to distribute its memory requirements between several threads. If we follow the CUDA threading model straightforwardly, we should implement a parallel function to be performed by a thread block. However, for some functions, the thread block can be too large structure. Its size should be a multiple of 32 threads (the size of one warp), and some functions do not expose as large amount of parallelism but use too much of memory resources to be efficiently performed by thread.

This work was supported by the Ministry of Education, Youth and Sports of the Czech Republic under the research intent No. 0021622419 which is highly appreciated.

For functions that cannot utilize per-thread either per-block implementation efficiently, we introduce an implementation pattern, where each function is performed by a few threads and a few functions run within a thread block. We call it *medium-grained* pattern, as it does not fit to both fine-grained per-thread and coarse-grained per-block implementation. This pattern brings some issues in efficient usage of the shared memory, which are addressed in this paper.

We have introduced a medium-grained pattern in [3], where it has been used mainly to implement kernels for mapped matrix or tensors multiplications allowing to speed-up matrix assembly in finite elements method. However, there was no deeper analysis of this pattern and its relation to specifics of GPU hardware as well as no example of implementation or performance evaluation of any medium-grained function. Independently, Klöckner at al. [4] used the same pattern in acceleration of discontinuous Galerkin method. Although some general characteristics of medium-grained implementation is discussed, it is mainly focused at problems related with applications in discontinuous Galerkin.

All performance evaluations mentioned in this paper has been performed using GeForce GTX 280.

II. The Space for Medium Granularity

In this section, the gap between areas where fine- and coarse-grained functions perform well as well as the performance impact of medium-grained implementation filling this gap is shown.

We have chosen the matrix matrix multiplication as a well-known task to demonstrate the impact of medium-grained implementations, i.e. $map(\cdot, A, B)$ is computed (A and B are lists of input matrices $A_1, .., A_n$ and $B_1, .., B_n$ respectively). The very fast implementation of matrix multiply kernel is known [2], nevertheless, it is usable only for large matrices and cannot be efficient if it is mapped to many very small matrices in parallel.

We have implemented four matrix multiplications – a fine-grained, a coarse-grained and two medium-grained ones. The fine-grained matrix multiplication implementation performs the whole multiplication in one thread, the coarse grained computes one element of resulting matrix in one thread. The first medium-grained implementation is analogous to the coarse-grained one, but there are more matrices multiplied within thread block. In the second medium-grained implementation, one thread computes one row of resulting matrix.

All matrices in our example are stored in continuous blocks in global memory, thus its loading into shared memory is necessary to maintain coalesced memory access for all implementations.

The Figure 1 left shows the performance of mapped matrix multiplication using different granularities. The fine-grained implementation yields better performance using small matrices, because of little resource usage, but descends quickly with growing size of matrices.

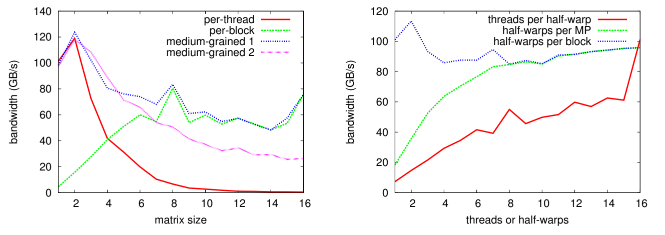

Fig. 1. Left: the performance of mapped matrix-matrix multiplications in different granularities, Right: the performance of global memory transfers.

The coarse-grained one performs well for larger matrices, which can be solved in sufficient number of threads in parallel and does not yield significant number of unused threads in warp. The first medium-grained implementation fills the performance gap between coarse- and fine-grained ones and for larger matrices behaves similarly to coarse-grained one. The second medium-grained implementation uses more shared memory per thread, thus its performance is lower for larger matrices, but for small ones its performance is higher due to faster access to shared memory, as it is mentioned below.

The size of the performance gap depends on the problem type – if more resources per thread are used, the performance of fine-grained implementation descends faster with growing size of input, but the coarse-grained implementation does not scale, thus the gap is wider. This occurs e.g. by using rectangular matrices.

III. GPU Performance Characteristics

In this section, the performance specifics of GPUs important for function mapping are discussed. The global memory performance for different warp and multiprocessor occupancies is benchmarked and the slowdown of some suboptimal shared memory patterns unavoidable in medium-grained implementations is evaluated.

A. Shared Memory

In case of medium-grained implementations, the bank conflicts can occur even if the mapped function accesses shared memory without bank conflicts, when more instances of the function run within single half-warp. We can recognize two sources of this inter-functions bank conflicts:

- threads of one function instance access data in exclusive memory banks in parallel, but the mapping of data addresses to banks overlaps for multiple elements accessed by multiple functions instances within half-warp
- all threads within the single function instance access data in same location, but when multiple functions runs in half-warp, multiple locations are accessed forbidding single broadcast

In the first mentioned case, the memory bank conflicts can be removed by padding or storing data elements interleaved (allowing to align particular data elements so the memory bank access exclusivity is reached both within function as well as among multiple functions). However, in the second mentioned case, the bank conflicts cannot be removed, because only one broadcast can be performed in time.

The slowdown of memory access with bank conflicts without any broadcasting is determined by the number of threads accessing different addresses in the same bank. The determination of slowdown when some threads reads the same location in memory is not so straightforward. The NVIDIA documentation describes a non-deterministic algorithm choosing randomly the broadcasting bank and performing normal data access to the rest of memory banks [1]. The choice of the broadcasted address can affect bank conflicts

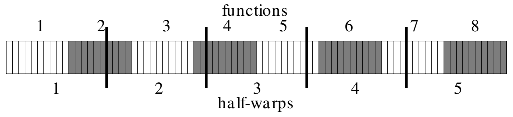

Fig. 2. The mapping of function instances to threads. Threads are depicted as bars, bold lines divides particular half-warps and the color differentiates particular functions, in this case 10 threads performs single function instance.

degree, thus all possible choices should be take into account where the most probable bank conflicts degree is evaluated. According to our microbenchmarks, the bank conflicts degree estimation matches experimental results quite tightly when perfectly random choice of broadcasted address is expected, thus we have not noticed any strong evidence to search for more precise prediction method. Since the adjacent half-warps can access shared memory differently, it is necessary to study sufficient number of half-warps (the access pattern repeats after at most n half-warps where the n is number of threads reading the same value).

The Figure 2 illustrates example of mapping the function instances to threads. All threads within single function instance reads single value. The first half-warp generates 2-way bank conflict because it needs two broadcasts to read all requested data, second half-warp generates 2-way bank conflict if data for second and third function are broadcasted before data for fourth function (two threads from fourth function gets data in parallel with this two broadcasts), otherwise three broadcasts are needed yielding 3-way bank conflict. The third half-warp generates 2-way bank conflict, the fourth and fifth are symmetric to second and first, respectively. Taking all possible choices of broadcasts into account, the average bank conflict is $2.2\overline{6}$-way in this case.

B. Global Memory

To be able to estimate global memory bandwidth, we have benchmarked the GPU for different warp and multiprocessor occupancy, and different block sizes. The benchmarks loads 16 floats into shared memory, synchronize and stores it back to global memory, which is the characteristic behavior of our functions. At Figure 1 right, the performance of various number of active threads in half-warp as well as various block sizes when single or maximal number of blocks runs at multiprocessor are depicted.

IV. Performance Prediction

Having knowledge of how the GPU performs in tasks specific for medium-grained implementations, the developer can predict the performance of particular implementations allowing him or her to implement only a version with should reach the best performance. In this section, we demonstrate the performance prediction on mapped matrix multiplication presented in Section II, namely the multiplication of 4×4 matrices.

Two tasks are evaluated separately – the global memory transfers needed to perform the given function, and the floating point instructions throughput of the function. When the GPU occupancy is good and sufficient number of blocks run at multiprocessor concurrently, the computation and memory transfers can overlap and the function runs at speed determined mainly by slower of this tasks.

The 4×4 matrix multiplication performs 128 floating point operations and moves 48 floats from/to global memory, giving the flop-to-world ratio $2.\overline{6}$. The function computes $c_{i,j} = \sum_{k=0}^{3} a_{i,k} \cdot b_{k,j}$ where i determines column and j determines row of the matrix.

The fine-grained implementation needs 48 floats in shared memory per thread, thus only about 85 threads can run in multiprocessor (the

	fine	coarse	medium 1	medium 2
GMEM estimated	70	101	114	114
GMEM measured	49.9	71.6	98.2	104.5
on-chip estimated	< 339	93.3	93.3	128.7-162.3
on-chip measured	143.9	85.4	90.3	135.3
total measured	41.8	40.9	80.6	88.8

TABLE I
ESTIMATED AND MEASURED BANDWIDTH OF MATRIX MULTIPLICATION.

exact number is influenced by compiler using shared memory to store function parameters and by the block size), restricting the memory bandwidth to about 70 GB/s. The computation performs one load instruction to one multiply-add (MAD) instruction (the arithmetic instructions can work only with one operand in shared memory [2]). To omit bank conflicts in accessing multiple matrices in parallel, the shared memory space allocated for each matrix is enlarged by 1 float. To perform single function, the 64 MAD, 64 load and 16 store instructions is used yielding about 226.2 GFlops on GTX 280 (the MAD with one operand in shared memory needs additional 2 cycles per warp [2]). Nevertheless, because of very poor occupancy, the real performance can be significantly lower (at least 192 threads is recommended to hide registry read-after-write latency [1]).

The coarse-grained implementation has sufficient GPU occupancy and the half warp occupancy has not significant impact to global memory bandwidth, which is bounded by 101 GB/s in this case. However, there are numerous memory bank conflicts. Within thread block, 4 threads in the same row simultaneously reads the same number from a, and 4 threads in the same column the same number from b. This yields 4-way bank conflict slowing the load and MAD instructions by factor of 4. The large degree of serialization caused by bank conflicts yield good GPU instruction pipeline occupancy even when only half of threads in warp is utilized. Altogether, the 64 MADs and 64 loads (all of them running at 25 % of maximal speed due to bank conflicts), has to be performed per function. The shared memory stores are not necessary – resulting elements can be stored to global memory from registers. The estimation of computation without global memory transfers is 62.2 GFlops.

The first medium-grained implementation can utilize full warps, thus the global memory can perform at maximal speed about 114 GB/s. The estimation for computation without global memory transfers is same as for coarse-grained implementation.

The second medium-grained implementation still keeps good GPU occupancy (about 341 threads can run at multiprocessor), thus the estimation of global memory bandwidth is same as in previous case. The reading of elements of b yields 4-degree bank conflicts as in previous medium-grained implementation, but the reading of a can be conflict-free when each row of a is enlarged by one unused float. The storing results into shared memory is necessary here to allow efficient transfer to global memory (bank conflicts can be removed by same padding as in a). This implementation needs 64 MADs and 64 loads, one of them running at 25 % of maximal speed and 16 stores. The performance estimation for computation is from 85.8 GFlops to 108.2 GFlops depending of compiler choice of elements used by load instructions and used by MADs.

Table I compares estimated and measured performance (to be easy comparable, performance of on-chip computation has been expressed as corresponding bandwidth). Both memory bandwidth and instruction throughput estimations are quite accurate excepting the case when GPU is heavily underutilized (the case of fine-grained implementation). The total performance estimation needs to estimate both memory bandwidth and instruction throughput, because it is limited by slower one. Although estimation of this two particular values is quite accurate, the total performance is also limited by the ability of GPU to overlap memory transfers and computation, which is obviously worse in the coarse grained implementation when only blocks of half-warp size are utilized.

Although we cannot estimate overall performance very precisely, the particular estimations gives a good comparison of possible implementations allowing us to implement one with best chance to outperform the others. Moreover, the estimation of instruction throughput is important when more complex function is developed reusing many data already stored in on-chip memory (e.g. a fusion of few algebraic routines).

We note in the case of 4×4 matrix multiplication, the broadcasts producing bank conflicts can be completely removed. In the case of coarse and first medium-grained implementation, the computation of dot product of vectors from input matrices can start at $(i + j) \bmod 4$ for element i, j of resulting matrix. For second medium-grained implementation, we can start dot product computation at $j \bmod 4$ for j-th row of resulting matrix. Although these modifications yields better results for 4×4 matrices, we decided to not discuss them here, because they remove bank conflicts only for this specific size and for most sizes of matrices reduce the performance.

V. CONCLUSION AND FUTURE WORK

In this paper, we have argued that some mapped functions cannot perform optimally using both fine-grained and coarse-grained implementations. The medium-grained pattern has been introduced and its usability has been demonstrated using example implementation of matrix-matrix multiplication. We have analyzed main GPU characteristics important for medium-grained implementations. This knowledge have been demonstrated at performance estimation of mapped matrix multiplication function.

The usability of the medium-grained pattern is restricted to functions with specific size and level of parallelism. However, the size of data elements processed by mapped function as well as a exploitable parallelism are fixed for specific task and thus independent to task size , so the utilization of medium-granularity is necessary to exploit maximal available GPU computational power in some cases.

The development of mapped functions becomes significantly more difficult when the function is more complex, i.e. performing complex task using several simpler algorithms in serial to transform input to output. In this case, the choice of proper parallel granularity as well as functions complexity is challenging. In [3], we have proposed the decomposition-fusion approach, when as simple as is meaningful functions are implemented and later some of them are fused into more complex ones. Currently, we are working on automatic tool searching the state space of possible fusions and choosing the functions with proper granularity to be fused.

REFERENCES

[1] NVIDIA CUDA Programming guide version 2.3. NVIDIA, 2009.
[2] Vasily Volkov, James Demmel. Benchmarking GPUs to tune dense linear algebra. *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, 2008.
[3] Jiří Filipovič, Igor Peterlík, Jan Fousek. GPU Acceleration of Equations Assembly in Finite Elements Method – Preliminary Results. *Symposium on Application Accelerators in High-Performance Computing 2009.*
[4] Andreas Klöckner, Tim Warburton, Jeffrey Bridge, Jan S. Hesthaven. Nodal Discontinuous Galerkin Methods on Graphics Processors. *Journal of Computational Physics, Volume 228, Issue 21*, 2009.