Final Write Up for SM Centric Program Transformation

Joymallya Chakraborty
Computer Science
North Carolina State University
jchakra@ncsu.edu

Abstract

A Graphics Processing Unit (GPU) is a specialized electronic circuit designed to rapidly manipulate and alter memory to accelerate the creation of images in a frame buffer intended for output to a display device. With hundreds of cores integrated, the GPU often creates tens of thousands of threads for an application. The computation power of GPU depends on the degree of parallelism. To utilize maximum potential of GPU, thread scheduling should be done in proper way. There are two ways of doing this scheduling –Hardware Scheduling and Software Scheduling.
As performance of Hardware Scheduling is not very satisfactory, researchers mainly try to concentrate on software scheduling. Here, we will describe SM (Streaming Multiprocessor)-centric transformation method [5] to achieve program level scheduling. As an extra advantage, this method opens up the scope of GPU program optimization. SM centric transformation makes affinity based scheduling possible. The performance metrics will be system throughput and average turnaround time.
In recent times, researchers have tried to use persistent threads to solve the scheduling problem. But this is not sufficient as even using persistent threads also, we cannot achieve location control. SM-centric transformation not only provides the freedom of spatial control but also helps to significantly improve performance in case of multiple kernel co-runs. This transformation comprises two steps –SM based task selection and Filling Retreating Scheme.

1 Background

GPU consists a number of Streaming Multiprocessors(SM), each of which contains tens of cores. At launch of a kernel, a huge number of threads are generated. Thirty-two threads compose a warp and many warps compose a thread block or CTA, many CTAs compose a Grid. After launching of a kernel, GPU hardware scheduler assigns each CTA to one of its SMs. But this assignment algorithm has not been disclosed to the public. It differs in each generation of GPU. If we compare two basic scheduling schemes in terms of advantage with respect to GPU performance, then spatial scheduling is more beneficial for GPU than temporal scheduling .
From resource sharing aspect, GPU shows non-uniformity. A CTA may share different amounts of data with different CTAs. In multi kernel co-run scenario, CTAs from the same kernel often share more instructions and data than CTAs from different kernels do. So, if we can achieve spatial scheduling properly, certainly we will observe performance speedup.

CourseProject for CSC-512, 1
2017. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM. . . $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2 SM Centric Transformation

SM centric transformation comprises two steps –SM based task selection and Filling Retreating Scheme.

2.1 SM-Based Task Selection

SM centric task selection is replacing job-worker binding with a binding between job and SM. Job means operation conducted by kernel group and worker means CTA. However, how many workers are assigned to an SM is determined by hardware scheduler, which is unpredictable. The non-determinism of hardware scheduler seriously affects the performance of SM centric task selection.

2.2 Filling Retreating Scheme

This scheme offers a simple way to precisely control the number of active workers on each SM. Let say, one SM can support at most m active workers at the same time. In the filling-retreating scheme, a total of $m * M$ workers are created at a kernel launch, where M is the number of SMs in the GPU. So, each SM gets m workers assigned. This step is the filling part of the scheme. The retreating facilitates flexible adjustment of the number of active workers on an SM.

3 Approach

To implement SM-centric program transformation, four macros will be used mainly.

1> _SMC _init - Before the invocation of a GPU kernel (CPU side)
2> _SMC _Begin and _SMC _End(GPU side)
3> _SMC _chunkID - Will replace CTA ID (GPU side)

These can be done easily by the compiler in one pass over the original GPU program. So, I have to develop a source to source compiler that will convert an input CUDA code to a SM-centric form. Initially I have considered some assumptions before implementing this. The initial assumptions are -

1> The program contains only one CUDA kernel function, and its definition and invocation are in a single file.
2> The grid(...) call is inside the same function that calls the CUDA kernel function.
3> The grid in the original program is one or two dimensional.

I have to use CLANG for this source to source transformation. Clang is a library to convert a C program into an abstract syntax tree (AST) and manipulate the AST. LibTooling, ASTMatcher, Rewriter will also be used for implementation part.

4 Task

I have to implement the following eight code transformations to convert an input CUDA code into a SM centric form.

1> Addition of the following line at the beginning of the CUDA file after all existing include statements: include "smc.h".
2> Addition of the following line at the beginning of the definition of the kernel function: _SMC _Begin.
3> _SMC _End will be added at the end of CUDA kernel function.
4> Addition of five new arguments - dim3 __SMC _orgGridDim, int __SMC _workersNeeded, int * __SMC _workerCount, int * __SMC _newChunkSeq, int * __SMC _seqEnds to the end of the argument list of the definition of the kernel function.
5> Occurrences of blockIdx.x and blockIdx.y will be replaced. blockIdx.x will be replaced by (int)fmodf((float) __SMC_chunkID, (float) __SMC __orgGridDim.x);
blockIdx.y will be replaced by (int)(__SMC _chunkID/ __SMC _orgGridDim.x);
6> Call of function grid() will be changed to dim3 _SMC __orgGridDim().
7> Addition of the following right before the call of the GPU kernel function: _SMC __init();.
8> Five new arguments - __SMC _orgGridDim, __SMC _workersNeeded, __SMC _workerCount, __SMC _newChunkSeq, __SMC _seqEnds will be added at the end of the GPU kernel function call.

5 Implementation

I have successfully implemented the above mentioned eight code transformations. Here I would describe my approach of implementations and some assumptions which I made while implementing. I have first read the Clang AST documentation [2] and learn about LibTooling, ASTMatcher and Rewriter [4] . If we closely observe above mentioned eight modifications, then seven among them are related to code rewriting. The only one which is different is adding a new header file. Because we know, preprocessor directives and header files are not part of the clang AST. So, the handling of this modification differs from rest.
For those seven similar kind of modifications, I have written a ASTMatcher for each one of them. ASTMatcher is like the query string based on which AST node of interest will be matched. Then, I have created child class inheriting MatchFinder parent class. Inside newly created class, virtual run method has been implemented. We know in C++, virtual function is such kind of function which is been declared in parent class and been defined in child classes. Inside this run method, the rewriting part has been done.

1> Finding the declaration of CUDA kernel function - I have used CUDAGlobalAttr property of FunctionDecl class to identify the CUDA kernel function.
2> Finding the CUDA kernel call - I have used cudaKernelCallExpr class to identify CUDA kernel call.
3> varDecl class was used to modify several variable declarations.

4> For the header file addition part, I have used PPCallbacks class. I have created a new class - Find _Includes which inherits PPCallbacks class [3]. Inside that class, I have implemented InclusionDirective() method. Inside this method, I have added the line "include"smc.h". Then inside MyFrontendAction class which inherits ASTFrontendAction class, I have implemented BeginSourceFileAction() method. Inside that method, I have created a std::unique _ptr referencing Find _Includes class. This ptr has been passed as an argument for addPPCallbacks() method.

6 Evaluation

For the evaluation part, I have used three CUDA programs as input. One is a normal program containing one CUDA kernel function, a code of matrix addition and another one of matrix multiplication. In all the cases, I got expected output. The input programs were successfully transferred to SM centric form.

7 Scope of Improvement

My implementation is based on some assumptions. If I could get more time, I would have implemented relaxing those assumptions. Here I am making a list of assumptions I made while implementing.

1>I have added "include"smc.h" at the end of all other includes. For this implementation, I needed to know the name of the last header file in the program. In all the test programs I used, helper _cuda.h was the last header file. So, after getting the location of last header file , I added an extra header file after that. But I think if I would get more time, I could have implemented it without knowing the name of the last header file in the input program.
2>For detection of CUDA kernel function declaration, I used CUDAGlobalAttr property. I am not sure whether all CUDA kernel function can be identified by this.
3>In all the input programs which I used as test cases, blockIdx.x and blockIdx.y have been assigned to two separate variables. Looking into those examples, I thought that if I get the declaration of those two variables, then I can easily change the assigned value. So, I followed this approach and successfully replaced blockIdx.x and blockIdx.y as I was supposed to. But then I realized , there could be some cases where blockIdx.x and blockIdx.y have been used without being assigned. In those cases, my implementation would fail to rewrite these two variables. I realized it too late to change my previous implementation. Then I tried to carefully look into the AST dump of input CUDA programs, I observed that blockIdx.x and blockIdx.y are not normal variables. They are GPU CUDA variables. So, they are structurally not like simple variables in the clang AST. If I could get more time, then I could implement this portion differently.

8 Learning Outcome

I was familiar with C++ before this project. Still while coding , I gained some more knowledge about inheritance,virtual function implementation, function overriding. I made myself familiar with Clang LibTooling and Rewriting [1]. Though this project was not much related to GPU CUDA, still I got an idea about GPU programming.

9 Comments

The project aimed at flexible task assignment on GPU. The procedure to achieve this goal is to use SM centric program transformation. I have implemented the program transformation part. It would be great if I could get more time to observe how this transformed code enables better task assignment and helps to achieve software level scheduling on GPU. In future, I would like to continue my research work in the domain of GPU,Clang and compiler.

References

[1] 2014. C++ Analysis using Clang. (2014). https://ehsanakhgari.org/blog/2015-12-07/c-static-analysis-using-clang
[2] 2014. Clang Tutorial. (2014). http://swtv.kaist.ac.kr/courses/cs453-fall13/Clang%20tutorial%20v4.pdf
[3] 2014. Preprocessor Github Repo. (2014). https://github.com/xaizek/self-inc-first/
[4] 2016. Clang 6 Documentation. (2016). https://clang.llvm.org/docs/
[5] Dong Li Xipeng Shen Bo Wu, Guoyang Chen and Jeffrey Vetter. 2015. Enabling and Exploiting Flexible Task Assignment on GPU through SM-Centric Program Transformations. (2015). https://people.engr.ncsu.edu/xshen5/Publications/ics15.pdf