SM-Centric Transformation

Henil Shah
Department of Computer Science,
North Carolina State University, USA
hkshah5@ncsu.edu

1 INTRODUCTION

Recently, the field of High-Perforamance Computing (HPC) has grown from being the domain of theoretical scientists and computer developers to an enormous field with applications ranging from modeling of complex physical phenomena such as weather, fluid dynamics to deep learning, Big Data, semantic analysis etc. With its massively parallel architecture consisting of thousands of small - efficient cores, general Purpose GPUs are one of the most prominent architectures used to run HPC applications.

The massively parallel architecture of GPU can run thousands of thread concurrently. Managing and scheduling these threads imposes grand challenges while optimizing performance of an application on the GPUs. Scheduling these threads to map memory access with the underlying architecture is necessary to achieve full potential of the GPU. Unlike CPU, GPU does not provide a software API to manage them. This is done by hardware at runtime to decrease scheduling overhead for such a large number of threads.

Many of the applications that GPU is used for are data-intensive tasks. Many applications have parallel tasks that do not have non-uniform amount of data with other tasks. Also, SMs on GPUs have non-uniform resources at hand. These are the two major reasons why introducing spatial locality can help gain major performance gain. But, due to our inability to control the scheduler, we lose out on a lot of performance because of lack of spatial locality in our program.

One major effort done to resolve the issue of controlling scheduling on GPU is Persistent Threads. In this approach, we create a small bunch of threads that run concurrently on the processor. Unlike normal threads, they are not killed once their task is done; rather they stay in execution throug out the program execution and keep fetching tasks from the task queue. This solves the problem to some extent, as in we have the control on the order of sequential execution of tasks on a particular thread. But this does not allow us to decide which tasks to run simultaneously. SM-Centric transformation of CUDA code helps us monitor this aspect of execution.

There are consistent proofs that introducing spatial locality through SM-Centric program transformation shows solid performance gain. There are results indicating increment in speedup and decrease in L1 cache misses. Such metrics show the potential of SM-centric transformation in performance optimization. Such performance improvement can help many data intensive applications that use GPU programming model. But transforming applications to valid SM-Centric transformed programs can be inconvenient for many CUDA developers. Therefore, the motivation behind developing a source to source translator to transform a CUDA code to SM-Centric tranformed program is that developing such a tool that allows them to leverage high-performance without any extra effort can be very helpful.

2 OBJECTIVE

Streaming Multiprocess (SM) Centric transformations are very useful in increasing efficiency of a program on GPU architecture. This software approach allows us to have flexible program-level control on scheduling jobs among workers and lets us introduce spatial locality to improve performance significantly. For this approach to work we need to modify the CUDA program such that it aligns with the SMC standards. In this project, we parse it and transform it in such a way that it can be compiled to make optimized executions with SM-Centric program transformation.

SM Centric transformation gives precise control over parallelism on GPU by giving control over number of active workers on a Streaming Multiprocessor. The code produced as a part of project should take a CUDA program as input and produce an output program with SM-Centric transformations. This requires us to make changes to the originial CUDA code based on the following 8 rules:

First two changes are made on the CPU side of code before GPU code is invoked

1. Adding *SMC_Init* right before the function call to CUDA Kernel.
2. Some arguments (*__SMC_orgGridDim*, *__SMC_workersNeeded*, *__SMC_workerCount*, *__SMC_newChunkSeq*, *__SMC_seqEnds*) are added to the end of the GPU Kernel Call.
   The *SMC_Init* call initiatest the addition variables passed to the kernel call in SMC form. For example, it initiates number of workers needed, the array of the desired sequence of IDs of job chunks, and an all-zero counter array to count active workers. The functions used to initiate the variables regarding workers and sequence of IDs can be provided by the programmer or the optimizing compiler; their definitions depend on the purpose of the specific application of the SM-centric transformation.

3. On the GPU side of code, the function definition reflects the arguments corresponding to the extra SMC variables passes to the kernel call. For that we add *dim3 __SMC_orgGridDim, int __SMC_workersNeeded, int *__SMC_workerCount, int * __SMC_newChunkSeq, int * __SMC_seqEnds* to the function definition.
4. At the beginning of the function definition we add *__SMC_Begin*. This macro works on assigning jobs to an SM by getting its ID and checking whether it has enough active co-operative thread arrays (CTA).
5. At the end of the kernel function definition we add __SMC_End. This is a trivial macro that puts and end bracket to the for loop that __SMC_Begin runs in order to do its job.
   In SMC for all the occerences of CTA (co-operative thread arrays) ID should be replaced with __SMC_chunkID. Which leads to the following two changes in our CUDA code.
6. The references of blockIdx.x are replaced with $(int)fmodf((float)\_\_SMC\_chunkID, (float)\_\_SMC\_orgGridDim.x)$
7. References of blockIdx.y are replaced with $(int)(\_\_SMC\_chunkID/\_\_SMC\_orgGridDim.x)$
8. the call of function grid(...) is replaced with *dim3 __SMC_orgGridDim(...)*. This assumes that the grid is 3 dimensional. For two dimensional grid it will be *dim2 __SMC_orgGridDim(...)*.

To achieve this objective, a program has been implemented using Clang that takes the input code and performs requires modifications throught the generated Abstract Syntax Tree (AST).

Following sections discusses the program implemented to perform this task, the challenges associated with, the solutions incorporated in the implementation and achievement and limitations of the resulting product.

3 RELATED WORK

Due to advances in various fields such as Big Data Analysis, Artificial Intelligence etc that demand large amount of computation power, demand of high-performance computing architecture has been increased significantly. GPU has become a very popular choice in such applications due to its massive level parallelism and large memory bandwidth. Still, most of the time, full potential of a GPU is not achieved because of its inaccessible hardware level scheduler.
Many of the applications that GPU is used for are data-intensive tasks. Many applications have parallel tasks that do not have non-uniform amount of data with other tasks. Also, SMs on GPUs have non-uniform resources at hand. These are the two major reasons why introducing spatial locality can help gain major performance gain. But, due to our inability to control the scheduler, we lose out on a lot of performance because of lack of spatial locality in our program.

4 CHALLENGES AND SOLUTIONS

Following are the major challenge faced during the development of the Clang solution to implement the project.

1. The biggest challenge in writing in creating the solution was to understand the AST dump of the input code. To make the modifications, the "Re-writer" code first required to find the appropriate node and insert / replace text there. But even trivial code snippets have complex and huge AST underneath. Therefore brosing through the huge AST dump of the entire code and analyze it is practically impossible.
   To solve this problem, before starting to write the rewriter, I wrote Finder codes that dumped only some specific nodes instead of the entire code. This made understanding structure of the relevant nodes very convenient.
2. During the implementation of the re-writer, a major problem that I faced was multiple detection of CUDA function definition. This problem initially seemed unpredictable but after testing it with a couple of codes I observed that this was happening with function that have multiple call expression for the CUDA kernel function. For example, one of the benchmark code *MM* had multiple CUDA call expression in an *if* condition with different parameters for different cases. In such cases the function detection occurred multiple times and which lead to a couple of problems. The SMC statements *__SMC_Begin* and *__SMC_End* and the additional SMC arguments in the function declaration got added multiple times in the function. Other modifications to the kernel function definition required me to replace existing text with the SMC suitable text. Therefore those changes did not pass the condition of identification after the first time and I did not face any issues with that. Only the part where I had to insert the text without modifying already existing code faced issues.
   To resolve this issue I added a global variale to keep check of the number of times the function is detected. After one time the modifications are made to the kernel function definition the condition to insert text in that function definition turns false and even though the function is detected multiple times, the text is added only once.
3. The most challenging rewriter was the rewriter for blockIdx.x and blockIdx.y. From looking at the code, these two seem like variables but since they are inbuilt values refering to ID of a block, they had rather complicated AST. To get the Calle with Member Expression "blockIDx" and after getting that, to differentiate between blockIDx.x and blockIDx.y, a lot of encapsulating AST expression nodes had to be condition checked and removed.

To implement this, I fist got a good picture of what the AST node for blockIdx looked like and then wrote conditions for the AST nodes - starting from the outermost to moving inwards. At each point, the inner expression was casted to the desired statement class of the AST of blockIdx. But casting without checking the type of the expression resulted in segmentation fault because downcasting through *static_cast* is allowed in C++ but when we try using method of that statement type, a segmentation occures for statements that are originally of not that type. Therefore, at each point, before downcasting the expression, I checked the Statment Class using getStmtClass() method.

4. Creating the callback function to add #include "smc.h" statement

5 LESSONS AND EXPERIENCES

The biggest learning product of this project was a deep and thorough understanding of the Clang tool and the AST class. It also turned out to be a very good experience and lead to a better understanding of code parsing. As an inexperienced person in this area, the subtle details of a code structure like the AST of blockIdx do not come intuitively. Now after working with such programs, understanding of the code structure has been enhanced significantly. Details like Attributes, Arguments etc of a statment and how they are arranged in the AST. I now have a better understanding of the differences between different types of declarations, expressions, statements and what kind of methods that are accessible for those classes.

Another part of the project where I struggles initially and benefited a lot later was the understanding of the data types in clang. Clang expressions have different types of tokens and pointers like iterators, Source Range, Source Manager etc have similar properties but working through the project gave a better understanding of the nuances between them and a better understanding of when which type will be used.

Working on modifying an input program required me to find, locate and modify a specific pattern. These different functionalities have different structures to implement them. After working through the project now I have a good hold of writing finders, matchers, visitors, call-backs and a good intuition of when to use them.

6 TEST FILES

For an SMC transformed code to run soundly on GPU, it needs to meet two conditions:

1. The operations by different threads are discriminated only by the global thread ID.
2. The execution order of the CTAs does not disturb the correctness of the kernel.

This is to ensure that SM-centric transformation does not change integrity of a job even though all appearances of the CTA ID in a kernel are replaced with the SMC chunkId and that the new order of execution maintains the meaning of the program. The two conditions hold for well-formed GPU programs, due to the nature of GPU execution models. for a GPU program to work properly, it should not rely on the execution order of CTAs, because due to the non-determinism in CTA scheduling on the GPU, it is hard to know what order would be taken in a run. Free from data race helps ensure the conditions hold.

The two programs used to verify the output of the program are benchmark codes MM and MAM. MM implements Matrix Multiplication and MAM implements Matrix Addition. These two codes follow the given two conditions.

These two codes have some similar patterns:

1. Both had only one CUDA kernel function.
2. Occurence of blockIdx in both the codes are on RHS of expressions as a simple delcaration expression. They are not a part of any binary or other complex expression
3. The kernel function declaration, definition and call expression are all in the same file.

The only major difference between both of them is that MM has two CUDA call expression statement (in an if-else condition) while MAM has only one CUDA call expression statement. Because of this difference, the matchere of CUDA kernel function defintion is called more than once in MM but not in MAM.

7 EVALUATION CRITERIA AND RESULTS

The SMC output for the input CUDA program should be syntactically and semantically correct.

For it to be syntactically correct it should have the right syntax, i.e. it should follow the 8 rules given in section-2.

For it to be semantically correct it should follow the following measures:

1. "smc.h" file should be included after all the header files in the source code.
2. No matter how many CUDA kernel call are made, *__SMC_init*() should be added only once.
3. The parameters should match between the CUDA call expression and CUDA Kernel declaration.

The code implemented for this project makes all 8 modifications to these two input files and gives the correct output in a file with name modified with "_smc.cu".

8 REMAINING ISSUES

The submission covers all the modifications given in the project description. It gives correct output with the given test-cases.
But this implementation doesn't include the functionality where it includes __SMC_Init() each time there's a CUDA call.

This can be avoided by keeping a global variable that goes to "False" after first time we add "__SMC_Init()". But in this case we have to be careful that the expression is not added inside any if condition, switch or any loop body. Otherwise the implementation will be wrong.

References

[1] [n. d.]. Clang CUDA call expression. https://clang.llvm.org/doxygen/classclang_1_1CUDAKernelCallExpr.html. ([n. d.]).

[2] [n. d.]. Clang Lexer. https://clang.llvm.org/doxygen/classclang_1_1Lexer.html. ([n. d.]).

[3] [n. d.]. Clang PPCallBacks. https://clang.llvm.org/doxygen/classclang_1_1PPCallbacks.html. ([n. d.]).

[4] [n. d.]. Clang SourceLocation. https://clang.llvm.org/doxygen/classclang_1_1SourceLocation.html. ([n. d.]).

[5] [n. d.]. Compiler Instance. https://clang.llvm.org/doxygen/classclang_1_1CompilerInstance.html. ([n. d.]).

[6] [n. d.]. Extra Clang Tools 6 documentation. http://clang.llvm.org/extra/pp-trace.html. ([n. d.]).

[7] [n. d.]. Source Manager. https://clang.llvm.org/doxygen/classclang_1_1SourceManager.html. ([n. d.]).

[8] Eli Bendersky. [n. d.]. Basic source-to-source transformation with Clang. https://eli.thegreenplace.net/2012/06/08/basic-source-to-source-transformation-with-clang. ([n. d.]).

[9] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, and Kevin Skadron. 2008. A performance study of general-purpose applications on graphics processors using CUDA. *Journal of parallel and distributed computing* 68, 10 (2008), 1370–1380.

[10] Kayvon Fatahalian and Mike Houston. 2008. A closer look at GPUs. *Commun. ACM* 51, 10 (2008), 50–57.

[11] M. Garland, S. Le Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, and V. Volkov. 2008. Parallel Computing Experiences with CUDA. *IEEE Micro* 28, 4 (July 2008), 13–27. https://doi.org/10.1109/MM.2008.57

[12] Kshitij Gupta, Jeff A Stuart, and John D Owens. 2012. A study of persistent threads style GPU programming for GPGPU workloads. In *Innovative Parallel Computing (InPar), 2012*. IEEE, 1–14.

[13] Olaf Krzikalla. [n. d.]. Performing Source-to-Source Transformations with Clang. https://llvm.org/devmtg/2013-04/krzikalla-slides.pdf. ([n. d.]).

[14] Nathan Otterness, Ming Yang, Sarah Rust, Eunbyung Park, James H Anderson, F Donelson Smith, Alex Berg, and Shige Wang. 2017. An evaluation of the NVIDIA TX1 for supporting real-time computer-vision workloads. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2017 IEEE*. IEEE, 353–364.

[15] Sagar Pandya. [n. d.]. "What's the right way to match includes (or defines) using Clang's libtooling?". https://stackoverflow.com/q/27029313. ([n. d.]).

[16] Xipeng Shen. [n. d.]. Slides of CSC 512: Compiler Construction. ([n. d.]).

[17] The Clang Team. [n. d.]. Clang Documentation. https://clang.llvm.org/docs/LibASTMatchersTutorial.html. ([n. d.]). Accessed: 2010-09-30.

[18] Bo Wu, Guoyang Chen, Dong Li, Xipeng Shen, and Jeffrey Vetter. 2015. Enabling and exploiting flexible task assignment on GPU through SM-centric program transformations. In *Proceedings of the 29th ACM on International Conference on Supercomputing*. ACM, 119–130.