Free Launch

Optimizing Dynamic Kernel Launches In GPU Through Thread Reuse

Anuraag Motiwale
Graduate Student - North Carolina State University
asmotiwa@ncsu.edu

Abhishek Singh
Graduate Student - North Carolina State University
aksingh5@ncsu.edu

1. Problem to Address

GPU (Graphical Processing Unit) possess immense potential to accelerate several general purpose computations by exploiting native dynamic parallelism support provided by GPU. At present, there are two ways through which dynamic parallelism can be exploited:

1. Software based approach.
2. Hardware based sub-kernel launch approach.

These two methods come with significant overhead and complexities. The software based approach are complex to code and involves intricate data structures, which comes with their own set of complexities and load imbalance problems. The hardware based approach involves launching child sub-kernels during run-time through programming interface. These sub-kernel launches are subject to large time overheads of maintenance of states of parent threads and, creation of child kernel and threads.

The problem to address is to come up with a novel solution which can address the overheads of above mentioned techniques by capturing the best of both worlds i.e. ease of programming, load balancing and convenience of sub-kernel launches.

CONF 'yy, Month d–d, 20yy, City, ST, Country.
Copyright © 20yy ACM 978-1-nnnn-nnnn-n/yy/mm. . . $15.00.
http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

2. Background and Motivation

GPU shows great potential for accelerating general-purpose computations, and many emerging data intensive applications are trying to exploit GPU for computation accelerations. Dynamic Parallelism support in GPUs enables a kernel to create and synchronize new nested work. Basically, a child kernel can be called from within a parent kernel and then optionally synchronized on the completion of that child kernel. The parent kernel can consume the output produced from the child Kernel, all without CPU involvement. An application can thus launch a coarse-grained kernel which in turn launches finer-grained kernels to do work where needed. This avoids unwanted computations while capturing all interesting details.

Dynamic parallelism is generally useful for problems where nested parallelism cannot be avoided. This includes, but is not limited to, the following classes of algorithms:

- Algorithms using hierarchical data structures, such as adaptive grids.
- Algorithms using recursion, where each level of recursion has parallelism, such as quicksort.
- Algorithms where work is naturally split into independent batches, where each batch involves complex parallel processing but cannot fully use a single GPU.

In this project we are aiming to explore a mechanism to fully synthesize the potential of dynamic parallelism by addressing the overheads associated with software-based and hardware-based approaches.

3. Related work

Following are some of the state of the art works carried out focusing on efficiently exploiting parallelism on GPU:

- NVIDIA introduced the concept of dynamic parallelism on GPU wherein threads can launch sub-kernels dynamically. This approach resulted in significant increase in performance for Quick-sort and other clustering algorithms.
- The negative point about the sub-kernel launch is the overhead time introduced. To reduce this overhead, Wang and Yalamanchili in their recent study, proposed a method called ”Dynamic Thread Block Launch” which is a lightweight executuon of the original sub-kernel launch. This new method requires hardware extension to perform efficiently.
- A new compiler named ”CUDA-NP” developed by Yang and others attempt to exploit nested parallelism by creating a large number of GPU threads initially and using control flows to activate different number of threads for different code sections.
- Wu and others proposed a ”SM-centric transformation” based on the approach of controlling the placement of GPU tasks on GPU streaming multiprocessors enhancing data locality across GPU threads.

4. Solution to the Problem

We are striving to build a mechanism which replaces the sub-kernel launches by re-use of parent threads to complete the task of child kernel. Our software solution will replace all the sub-kernel calls in the code with re-assignment of that task to one of the parent threads. This way programmer can harness the flexibility of sub-kernel launch capability provided by hardware-based approach and avoid the time overheads associated with sub-kernel launches. The parent thread to which the sub-kernels task will be assigned will be based on an adaptive approach which takes into account the number of parent thread kernels and the statistics pertaining to child kernel calls of parent threads. This ensures optimal load balancing considerably superior to its software-based counterparts. Our solution will broadly consist of two parts:

1. Sub-kernel launch transformation: This enables replacement of sub-kernel launch with re-use of parent threads for the same task. Code transformation can be classified into four categories:
   (a) CB* $\rightarrow$ PB*(T1): In this transformation the assignment unit is subtask within a kernel. The assignment is done at thread block level. Thus, any subtask can be assigned to any parent thread block. Global assignment scheme is most intuitive way to achieve load balancing but incurs runtime control overhead.
   (b) CK* $\rightarrow$ PB*(T2): In this transformation the assignment unit is sub-kernel and the assignment is done at thread block level. This scheme can help achieve good load balance and relatively less run-time overhead compared to CB* $\rightarrow$ PB* scheme.
   (c) CKi $\rightarrow$ PBi(T3): In this transformation the assignment unit is sub-kernel. The sub-kernels are locally assigned among threads of parent block whose threads were originally supposed to launch these sub-kernels. This assignment though not as flexible as earlier ones, avoids run-time overhead.
   (d) CKi $\rightarrow$ PTi(T4): In this transformation as well, the assignment unit is sub-kernel. The sub-kernels are assigned to the parent thread which was originally supposed launch this sub-kernel. Key point to be noted is that the sub-kernel is assigned to parent thread rather than the parent thread block.
2. Task assignment selection: The four transformations mentioned above have their own strengths. We will implement a run-time transformation selection algorithm to select appropriate transformation scheme to achieve optimal load balancing. The algorithm uses host side code to select among [T1 or T2], T3 and T4. The host side algorithm computes the maximum number of persistent thread block which can be supported by GPU, if this number is greater than the original number of parent thread blocks in kernel then selection of T3, T4 would render some persistent thread blocks idle hence [T1 or T2] should be selected. Otherwise the imbalance of number of child kernel of the parent threads is examined. If the standard deviation for the number of child kernels among parent threads is greater than the mean of the number of child kernels, this implies global balancing is important and hence [T1 or T2] should be chosen. Else it checks if more than half the parent threads launch sub-kernels. If thats the case, T4

transformation is chosen else T3 is chosen. [T1 or T2] is usually chosen when there is not enough information about number of sub-kernel launches by parent thread or when there is large imbalance in sub-kernel launches among parent threads. Selection between T1 and T2 is done during runtime. The algorithm uses the number of sub-kernel calls to compute the number of extra subtasks the most loaded persistent block is subject to if T2 is applied, compared to load of each persistent thread block if T1 is applied. It then compares the overhead of extra subtasks in T2 with extra control overhead in T1, and chooses the more efficient of the two.

5. Outcome Assessment

To evaluate the working of this source-to-source compiler, we require existing CUDA benchmarks that uses dynamic parallelism. We intend to use the publicly available CUDA benchmarks that uses dynamic parallelism. Some of the benchmarks available are as follows:

- Single-Source Shortest Paths
- Graph Coloring
- Breadth-First Search
- Connected Components Labelling
- Find Minimum Spanning Tree

If the benchmarks used for test cases does not contain dynamic parallelism, then the nested loops present can be transformed into sub-kernel calls. This has to be done by manually altering the code.

For assessing the output generated, following parameters should be considered:

- Correctness of the transformed code when compared with the original code.
- The speed-up achieved by the transformed code with thread reuse as compared to the original code with sub-kernel launches.

The main intention is to reduce the overhead for dynamic parallelism. So, it is important to compare the speed-ups achieved by the transformed versions of the original code in order to get a better idea for choosing a version with higher efficiency.

6. Assignment of Jobs

We are going to do pair programming during the development phase to aid the learning and engagement. Thereafter, in the testing phase we'll split the test cases among us and test independently.

7. Project Time-line

The general time-line to be followed for structured development of the project is as follows:

Dates	Milestones
10/21/2017	Built up background knowledge required to approch this project by reading and understanding the freelaunchPaper_micro2015.pdf paper and going throught the example provided in order to gain essential insights into the transformation done to the original CUDA code.
10/23/2017	Submission of first deliverable. A report to be submitted outlining the overview of the problem faced, its basic solution and its output assessment.
10/28/2017	Learn about ”Clang” and setup the work environment required to develop the compiler.
11/01/2017	Selecting the CUDA benchmarks to be used as test cases.
11/05/2017	Starting to implement the compiler that will work for basic test cases. The complicated conditions will be considered rested.
11/14/2017	Second deliverable due.
11/27/2017	Final stages of the development of the compiler.
12/1/2017	Testing the outputs generated and generating statistical analysis and if possible working on extra features to be supported by the compiler.
12/6/2017	Finishing up the project by adding the readme and test cases to the project.
12/10/2017	Final submission of the project.

8. Solution Implementation

A source-to-source compiler to realize above mentioned transformation seemed to be most intuitive to achieve optimization of sub-kernel calls and load balancing. All four transformations were achieved by materializing all or some of below mentioned alteration in source code:

- Identifying the kernel calls which contained sub-kernel calls, writing handlers, and simultaneously making sure no other kernel calls were altered was pivotal to the project. Thorough testing was done to accomplish this objective.
- Inclusion of required transformation's header file. The corresponding header file housed all the macros which would be instrumental in realization of target transformation. All header files were included at first line of target file, accomplished by leveraging source manager.
- Extraction of number of blocks and number of threads in CUDA kernel call: This was achieved by extracting the pre-arguments from CUDA kernel and sub kernel call expression. These arguments were invariably used in all the transformations mentioned above.
- Addition of declaration for FreeLaunch argument array with each element representing the parameters of a children before the parent kernel call and alteration of kernel call to consume the previously declared FreeLaunch argument array and number of original parent thread blocks as additional arguments.
- Alteration of parent kernel call function declaration to accommodate aforementioned additional arguments added to parent kernel call expression.
- Insertion of associated PreLoop and PostLoop macros. These macros are custom made specific to each transformation to aid in the transformation process.
- Alteration of thread ID calculation by considering the persistent thread usage parent kernel call body.
- Recording sub-kernel call info and computation specific to the required transformation. This step had a lot of applicable static and dynamic component to be handled specific to the required transformation.
- Replacement of the calculation of the ID of the logical child thread in sub-kernel call body and copying the edited child kernel body into parent kernel call body.
- Omission of child kernel call body from the context. Since removal of sub-kernel calls is the aim of this project, it totally makes sense to get rid of the child sub-kernel call's body once it has been copies into parent's body.
- Replacement of all return statements with continue in both kernel and sub-kernel call bodies. It was essential to maintain the continuity of the transformed algorithm.
- Transformation algorithm actively uses the recorded sub-kernel call info for FreeLaunch specific computations. We accomplished this by maintaining a map of the sub-kernel call argument and its data type. These were the arguments of sub-kernel call which were not received as parameters in parent kernel call, but were passed as arguments in sub-kernel call.
- Our end product simply takes a program with sub-kernel calls and generates four transformation mirroring the transformations discussed in Solution section. These transformed program are free of any sub-kernel calls and can be run on GPUs to attest the flow preservation of the transformation.

9. Challenges

- It took us sometime to warm up to development in clang and to gain an understanding of clang AST. The scope of source to source compilation was a little overwhelming.
- Very few clang related tutorials were available at our disposal which seemed to be the bottleneck when we started exploring the source to source compilation paradigm with clang.
- Clang AST's understanding was central for this project. Gaining insight into the same was challenging. Browsing the AST to catch hold of the nodes of interest was a tricky.
- Clang has large array of classes and very intricate class hierarchy which was very abstract in the beginning and it was difficult to find a starting point.
- It took us a lot of literature review to catch hold of class hierarchy and the large code base covered under each class of interest. Galore of methods provided by each class was towering as well as exciting.

- Playing with the source range, which was vital for this project was elusive and took a lot of debugging.
- Writing matchers for specific crucial parts of input program like CUDA sub kernel call expression, CUDA sub-kernel call argument list, thread id calculation expressions in both kernel and sub-kernel calls, kernel call expression with sub kernel calls as descendent took us sometime to zero down on exact requirements.
- Debugging a code which compiles well on GPU but doesn't match the desired output was a humongous task, as it was very hard to pinpoint where everything was going south. It took major chunk of our time to zero in to the pain points.

10. Remaining issues and possible solutions

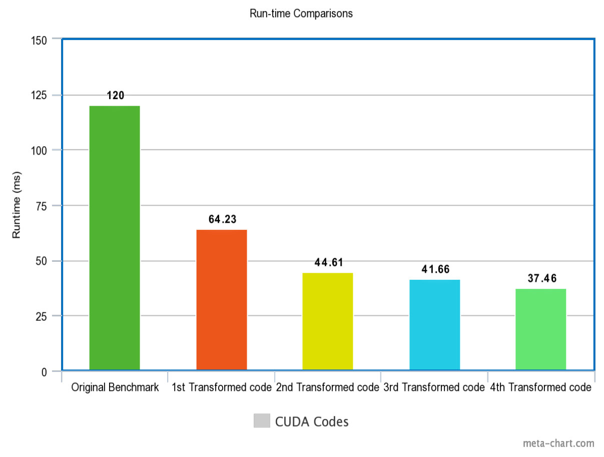
- Due to time constraints, we were not able to implement the the dynamic tool which will choose optimum transformation technique, that should be applied on a given code to achieve best results, based on several deciding factors. This mechanism would run part during compile time and part during run time.
- While matching the thread id calculation expression "threadIdx.x + blockIdx.x*blockDim.x", we only considered single permutation, identical to the one present in benchmark, whereas in real time application there can several other permutations which can be present in the program. To generalize the compiler we can add mechanism to handle all the permutations of the above mentioned expression.
- While identifying the kernel calls with sub-kernel calls we relied on static nesting of CUDA kernel call expression, while a more intuitive way could have been the implementation of a recursive AST node visitor to traverse the clang AST nodes.

11. Lessons and experiences

- Project helped a great to deal to put source to source compilation into perspective.
- Gained deep insight about sub-kernel launches and their optimization by thread re-use to achieve optimal load balancing and efficiency.
- Project well equipped us with skills required for tool development using clang.
- Gained better understanding of Clang AST and libraries, which is a huge take away from this project.
- Experienced real-time prowess of Clang's compilation speed. Clang felt a lot faster to compile than gcc is. Clang's compile times was up to twice as fast as compared to gcc.
- Got acquainted with superior compiler diagnostics of clang. Clang usually provides much better diagnostics in case your code fails to compile, which meant lesser time trying to understand what should we do to fix our code. It even goes further by suggesting you of the most likely fixes.

12. Result

The goal of the project was to develop a source-to-source translator that can convert an input CUDA code that uses dynamic sub-kernel calls to a free launch form. The benchmark used is a Minimum SPanning Tree program. The developed clang tool generated four different CUDA files from a single benchmark CUDA file where, each of the four files represented each of the transformations. Each of the 4 transformed codes were run on the GPU to compare the run-time efficiency after free launch transformations. A free launch form usually makes the code run. much faster. The following graph shows the comparison between the run-times of the original benchmark code and each of the transformed codes.



The above graph clearly shows the vast difference between the run-times of the original benchmark code with dynamic sub-kernel calls and the other free launch transformations. The run-time is the lowest for the fourth transformation of the benchmark which means that the original benchmark has more than half of threads that are launching sub-kernels. In this case the thread-level sub-task assignment will be a good choice as most threads will be busy and the run-time control overhead will be minimized (exact case for T4 transformation).

References

[1] https://github.com/llvm-mirror/clang

[2] https://github.com/llvm-mirror/clang

[3] http://clang.llvm.org/docs/LibASTMatchersReference.html

[4] https://clang.llvm.org/docs/LibASTMatchers.html

[5] https://eli.thegreenplace.net/2014/07/29/ast-matchers-and-clang-refactoring-tools

[6] https://xinhuang.github.io/posts/2015-02-08-clang-tutorial-the-ast-matcher.html

[7] https://github.com/eliben/llvm-clang-samples/blob/master/src_clang/matchers_rewriter.cpp

[8] https://variousburglarious.com/2017/01/18/finding-global-variables-with-clang-ast-matchers/

[9] https://jonasdevlieghere.com/understanding-the-clang-ast/

[9] hhttp://www.goldsborough.me/c++/clang/llvm/tools/2017/02/24/00-00-06-emitting_diagnostics_and_fixithints_in_clang_tools/

[10] https://github.com/loarabia/Clang-tutorial

[11] swtv.kaist.ac.kr/courses/cs453-fall13/Clang%20tutorial%20v4.pdf

[12] https://manu343726.github.io/2017/02/11/writing-ast-matchers-for-libclang.html

[13] https://www.crest.iu.edu/projects/conceptcpp/docs/html-ext/classclang_1_1RecursiveASTVisitor.html

[14] http://bbannier.github.io/blog/2015/05/02/Writing-a-basic-clang-static-analysis-check.html

[15] http://bcain-llvm.readthedocs.io/projects/clang/en/latest/h

[16] Guoyang Chen and Xipeng Shen, ”Free Launch: Optimizing GPU Dynamic Kernel Launches through Thread Reuse”, Computer Science Department, North Carolina State University