Free Launch

Bijo Joseph
North Carolina State University
bjoseph4@ncsu.edu

Karan Jadhav
North Carolina State University
kjadhav@ncsu.edu

Abstract

The hardware based approach of achieving dynamic parallelism in GPUs via sub-kernel launches is very intuitive to programmers. And the software based approach of load balancing helps achieve efficiency. In this paper we propose implementing Free Launch[1], a solution that avoids the overhead of sub-kernel launches by making reuse of parent threads, yet at the same time allows programmers to write in the intuitive sub-kernel way.

Categories and Subject Descriptors D.3 [*Programming Languages*]: Processors-optimization,compilers

Keywords GPU, Dynamic Parallelism, Thread reuse, sub-kernel, load balancing, Clang, LibTooling.

1. Introduction

"Early CUDA programs had to conform to a flat, bulk parallel programming model. Programs had to perform a sequence of kernel launches, and for best performance each kernel had to expose enough parallelism to efficiently use the GPU. For applications consisting of parallel for loops the bulk parallel model is not too limiting, but some parallel patterns such as nested parallelism cannot be expressed so easily." [2]

CUDA GPUs with capability 5.0 and above, allowed for dynamic parallelism by introducing subkernel launches which allow threads to launch more kernels to do finer grained work. This allows programmers to express dynamic parallelism intuitively.

CONF 'yy, Month d–d, 20yy, City, ST, Country.

http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

Existing software and hardware based techniques to utilize dynamic parallelism on GPUs have certain shortfalls. The software based approach requires programmers to write complex code to manage worklists that can efficiently load balance tasks across GPU threads for parallel execution. The hardware based approach uses subkernel launches, which have a high runtime overhead associated with them. This is on account of the time to save parent thread states, create child kernel and threads and restore states of parent threads.

2. Background and Motivation

Sophisticated implementations of the software based approach still incur load imbalance among threads. In the hardware based approach, hardware extensions have been proposed to fix the problem, but lack popularity. The Free Launch[1] paper proposed four different source to source transformations of programs with dynamic parallelism, that do not have the shortcomings of the other implementations. At the same time, it fixes the issues with the software and hardware based issues of expressing dynamic parallelism. The subkernel launch removal is achieved using four substasks assignment schemes. The first is assigning a subtask to any parent thread block. The second is assigning a task set of a subkernel to any parent thread block. The third is assigning a task set of a subkernel specifically to its parent thread block. And the last scheme is assigning a child task set specifically to its parent thread.

In this paper we mention how to utilize Clang's LibTooling library to generate the four transformed versions[1] after subkernel launch removal. Libtooling allows us to create a source to source transformation tool for each version. Clang does the preprocessing, scanning and generates the AST, saving us from the effort of writing the scanner and parser for CUDA code. In addition, Clang's Matcher and Rewriter functionality

provide us the necessary APIs for source to source transformation of CUDA code.

3. Objectives

Our objectives in this project were to implement the four source to source transformations[1], each of which would transform an input program that used dynamic parallelism, into one of the four versions. Each of these transformation versions had certain intricacies, that led to implicit sub-objectives and/or requirements in our transformations.We state these below.

1. Making parent threads persistent : Because of the global nature of certain transformation it is necessary to have synchronization across thread blocks, which is only possible if GPU kernel code uses persistent threads. Hence, we do this by keeping threads resident on SMs until the kernel finishes.

2. Parameters and shared memory : Every child kernel makes use of some parameters which also include the launch configuration. These parameters need to be recorded by parent threads to successfully fulfill child subtasks. Shared memory in GPUs is on-chip memory, which is accessible by threads in the same thread block. It helps improve performance as it is faster than using global memory.

3. Control flows and subtask assignment : Control flows are required for certain scenarios, for example when child kernel launches need to be synchronous. In this case the parent finishes child subtasks and then resumes its operations. Another example is synchronizing threads to avoid data races on the set of subtasks. Subtask assignment is utilized to ensure even partition of the subtasks among parent thread blocks.

4. Challenges

The steps needed to achieve the points mentioned in Objectives have been crafted as challenges and the solution for each challenge in mentioned in the section Solutions

1. Adding the header files required for each transformation.
2. In order to generate the transformed code that was void of sub kernel launches, our first task was to identify the exact location of the sub kernel launch in the source code. As we wanted to keep our implementation generic, it was important to not have any limitation on how a kernel call was called within a kernel call nor be dependent on any naming convention for parent and child kernel functions.
3. After identifying the parent kernel call and the child kernel call, the next challenge is to find the function definitions of the parent and child. We need the location of the parent function definition to add two new parameters required for transformation. The first parameter to get the number of parent thread blocks, before using persistent threads and the second parameter, an array that consisted of all the parameters of the child kernel.
4. Initialize the Free Launch arguments and assign it memory.Then in the parent kernel call pass in the new parameters. Finally after the call free the memory assigned for the Free Launch arguments
5. We also need to add additional code to parent function definition like macros relevant to each transformation, the calculation of thread ID considering persistent thread usage.
6. The transformed source code also needs to have the appropriate ”goto” and ”continue” statements in the parent function body to meet the control flow and subtask assignment scheme requirements of each transformation.
7. In the parent function body we also need to record kernel information, to do this first we needed to extract and store all the variable declaration types and variable names in the parent function body, these stored values should then used during code generation to record kernel information.
8. Our objective is to remove child kernel calls, so there is no use of a child function definition as all the code of the child function definition will be moved to the parent function definition body.Removing the child function definition will be a challenge, as the keyword global is a macro present in a different file. To remove this macro we need to go through Clang documentation and figure out how the Source Manager keeps track of files currently being accessed.
9. We also need to modify the existing control flow and sub task assignment in the child function body before moving all the child function body code to parent function body.

5. Solutions

We faced the same challenges while generating each transformation. So in this section we will only mention the solutions to challenges with respect to transformation T1 on mst-dp-modular. The additional steps apart from the common steps will be stated explicitly.

We added the headers required for the free launch transformation to the beginning of the main file as seen in Figure 1.

```
SourceLocation startOfFile =
Rewrite.getSourceMgr().getLocForStartOfFile(
  Rewrite.getSourceMgr().getMainFileID()
);
Rewrite.InsertText(startOfFile,
  "#include \"freeLaunch_T1.h\"\n",
  true,false
);
```

Figure 1. Header

To identify the parent kernel call and the child kernel call our Clang tool makes use of a Matcher that identifies a CUDA kernel call expression within a CUDA kernel call expression. We also bind these AST nodes in order to retrieve them in the respective CallBacks.The Matcher is given in Figure 2. The logic is as follows, we first identify a CUDA kernel call expression and then check for the callee, the callee method redirects us to the parent function declaration, once we get to the parent function declaration we search its body for a CUDA kernel call expression, the presence of a CUDA kernel call indicates a subkernel launch and this is the exact information we require to address challenge two. We say this is a generic solution as we do not depend any specific naming convention for parent and child kernel functions, nor are we confined to where in the source code there is a kernel call within a kernel call. Our Matcher is capable of identifying all such instances as long as we don't have any indirect child kernel calls.

```
StatementMatcher CudaKernelCallMatcher =
cudaKernelCallExpr(
  callee(
    functionDecl(
      hasDescendant(
        cudaKernelCallExpr().bind("childCudaCall")
      )
    ).bind("parentFuncDecl")
  )
).bind("parentCudaCall");
```

Figure 2. Cuda Kernel Call Matcher

The Matcher that helps address challenge two, gives us the location of the parent function declaration, now to address challenge three, i.e to add two new parameter to the parent function declaration. We first get the last parameter name as a string, and calculate it's length and use it as a offset to insert the two new parameters.We follow this approach as Clang does not provide a direct access to the end location of the parameter list,the end location provided by Clang's getLocEnd will always point to the location of the second last token.So we get the location of the second last token using getLocEnd(), and then add the offset, now we have the end location of the parameters for inserting the new parameters.The code that achieves this functionality is given in Figure 3.

```
string lastParam = parentFuncDecl->
  parameters()[parentFuncDecl->getNumParams() - 1]->
    getQualifiedNameAsString();
int offset = lastParam.length();
SourceLocation endParamLoc = parentFuncDecl->
  parameters()[parentFuncDecl->getNumParams() - 1]->
  getLocEnd().getLocWithOffset(offset);
string newParameters =  ", int blocks, char *FL_Args";
Rewrite.InsertText(endParamLoc, newParameters,true,true);
```

Figure 3. Appending Parameters

To initialize and free Free Launch arguments we just insert the appropriate code as strings, we believe the initialization will be the same across all programs. This can be seen in Figure 4. We also need to add two extra arguments to the parent kernel call, seen below in Figure 5, for the first we need to pass in the number of blocks, we can get this from the parent kernel calls execution configuration and then pass it as a parameter to the parent kernel call, the second parameter which is the Free launch argument array is inserted as a string as this will be same across all transformations.

```
string intializeBeforeParentKernelCall =
  "char *FL_Arguments;\n
  cudaMalloc((void **)&FL_Arguments,MAX_FL_ARGSZ);\n
  cudaMemset(FL_Arguments,0,MAX_FL_ARGSZ);\n";
Rewrite.InsertText(
  parentKernelCall->getLocStart(),
  intializeBeforeParentKernelCall,true,false
);
```

Figure 4. Free Launch args

The challenge five was addressed by using the two Matchers the first Matcher as per Figure 2 and the Matcher shown in Figure 6, this matcher is similar to Figure 2, the only difference is that this Matcher will trigger a call back for every variable declaration encountered in the parent function body. In the call back we match for the variable declaration that calculates the

```
SourceRange range =
  parentKernelCall->getConfig()->getArg(0)->getSourceRange();
llvm::StringRef numBlocks =
  Lexer::getSourceText(
    CharSourceRange::getTokenRange(range),
    *SM, LangOptions()
  );
string newParametersForParentCall =
  ", " + numBlocks.str() + ", FL_Arguments";
Rewrite.InsertText(
  parentKernelCall->getLocEnd(),
  newParametersForParentCall,
  true,true
);
```

Figure 5. Adding args to parent kernel call

thread ID and replace it as per each transformation as seen below in Figure 7. This replacement is done only for T1 and T2. It is in this callback that we also store all the variable in the parent function body that will be used to address challenge seven.After the callbacks of Matcher in Figure 6 complete then in the callback of Mather in Figure 2 we insert the appropriate macros needed for each transformation.

```
StatementMatcher ParentVarDeclMatcher =
cudaKernelCallExpr(
  callee(
    functionDecl(
      hasDescendant(
        cudaKernelCallExpr().bind("childCudaCall")),
        hasBody(
          forEachDescendant(
            varDecl().bind("parentVarDecl")
        )
      )
    ).bind("parentFuncDecl")
  )
).bind("parentCudaCall");
```

Figure 6. Parent Var Decl Matcher

```
if(parentVarStr.find(
  "blockIdx.x * blockDim.x + threadIdx.x"
  )!=std::string::npos) {
  replace(
    parentVarStr,
    "blockIdx.x * blockDim.x + threadIdx.x",
    "(blockIdx.x+FL_y) * blockDim.x + threadIdx.x"
  );
  Rewrite.RemoveText(parentVarRange);
  Rewrite.InsertText(
    parentVarRange.getBegin(),
    parentVarStr,
    false, false
    );
}
```

Figure 7. Parent ID calculation

In order to have the appropriate control flow and subtask assignment scheme in each transformation the return statement in the parent function body needs to replaced with goto/continue, we have a separate Matcher shown in Figure 8, that identifies all the return statements in the parent function body and in its MatchCallBack we replace it with either goto/continue statement relevant to each transformation. This helps us complete challenge six.

```
StatementMatcher ParentReturnStmtMatcher =
cudaKernelCallExpr(
  callee(
    functionDecl(
      hasDescendant(
        cudaKernelCallExpr().bind("childCudaCall")
      ),
      forEachDescendant(
        returnStmt().bind("parentReturnStmt")
      )
    ).bind("parentFuncDecl")
  )
).bind("parentCudaCall");
```

Figure 8. Parent Var Decl Matcher

In the parent function body we need to record the kernel information, for this we need the variables declared in the parent function body, and as we store these variables when we enter the CallBack, seen below in Figure 9, of Matcher(Figure 6).Then in the CallBack of Cuda Kernel Call Matcher we only need to generate the appropriate code, code generation is straight forward as we only have to create strings that resemble the exact code but in places that require the parent kernel information we substitute it the value we had previously stored. Thus achieving challenge seven.

```
parentVars.insert(
  pair<string, string>(
    parentVarDecl->getQualifiedNameAsString(),
    parentVarDecl->getType().getAsString()
  )
);
parentVar.push_back(
  parentVarDecl->
    getType().getAsString()+
  ";"+parentVarDecl->
    getQualifiedNameAsString()
);
```

Figure 9. Record parent variables

In challenge eight we were supposed to move the code in the child's function body to the end of parent's function body and also add the relevant code to retrieve the child kernel modified arguments, this process is straight forward as it again makes use of the variables we have stored initially and has a lot of code that will not change across programs, this subpart can be achieved in a similar way as solution to challenge

seven.To remove the child function definition we had to make use of the Source Manager. This was not very intuitive as the child kernel call made use of the keyword global. As global was considered a macro, to delete it we needed it's start location in our main file and not in the file where the macro actually existed. To achieve this we made use of the Source Manager.The Source Manager provides the location just before the start of the macro, and using the startLoc and endLoc methods of the function declaration helped us remove the entire child function definition during rewriting. This can be seen in Figure 10 below.To to remove the child kernel call we only need to extract the source range of the child kernel call and use the Rewriter to remove that code.

```
SourceLocation startChildFuncDecl =
  Rewrite.getSourceMgr().getFileLoc(
    childFuncDecl->getLocStart()
  );
SourceLocation endChildFuncDecl =
  Rewrite.getSourceMgr().getFileLoc(
    childFuncDecl->getLocEnd()
  );
Rewrite.RemoveText(
  SourceRange(
    startChildFuncDecl, endChildFuncDecl
  )
);
```

Figure 10. Remove Child Function

To modify the control flow in the child function body and calculate the global thread ID, we use string manipulation to replace the code with code required by each transformation. This transformed string is then inserted at the end of the parent function body using the Rewriter.This helps us complete the ninth challenge.

6. Lessons and Experiences

We first learned about LibTooling in Clang, how we can create standalone tools that use Clang's API to analyze and work on input source code. Next, our understanding of Abstract Syntax Tree was strengthened based on Clang's AST syntax, which we learned through the documentation and also by visualizing via dumping nodes that we wished to understand and work with. These were tied with the Matcher and Rewriter, which were features available in the Clang API that helped us write rules for matching nodes in the AST and then modifying them respectively.

Specifically because of the Free Launch topic, we learned about CUDA programs. Additionally, we understood the concept of dynamic parallelism and how it is different from the traditional flat origramming model.

7. Results

We transformed the provided mst-dp-modular into the four different versions using our source to source transformers. We then ran each of these versions, including the original source file against the rmat12.sym.gr test case provided in the lonestar package. The runtimes of each version can be seen in Figure 11 below. As we can see, the original version which uses the sub-kernel launch, takes the longest owing to the overhead associated with nested kernel launches. We notice that performance improves as we go from version T1 to T4. Ideally performance should improve as we go from T4 to T1 because of the increasing granularity and proper distribution of subtasks. However we speculate that in this test, this was not the case due to one or more of two reasons.

First, versions towards T4 take a performance hit when the amount of work done by child kernels varies greatly among parents who launch them. Due to the inequal work, some parent threads end up doing more work than others. In this case, the work being done by parent threads could have been comparably similar. The second reason could be that in cases where the amount of data being worked on is less, versions towards T1 take longer time owing to the overhead of recording child kernel parameters and granular work distribution. In this case, the data could have been so little, that the "better" versions simply spent more time in optimizing work distribution and comparatively lesser in doing the actual work.

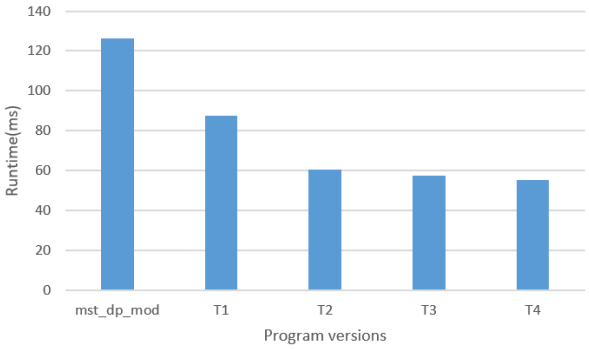

Figure 11. Runtimes for input rmat12.sym.gr

8. Issues

Our Clang tool has only been tested on the code in mst-dp-modular package due the time limitations, so its behaviour on other code bases cannot be accurately predicted.We are confident that our tool will be able to remove subkernel calls as long as there aren't any indirect subkernel calls. To be able to extract indirect subkernel call, we will need to modify our Matchers accordingly.

We have not given special importance to the aesthetics of the code, as our main goal was to get our tool to give the correct transformation with minimal emphasis on formatting and comments in the transformed code. An example could be that be that we just insert all the headers to the beginning of the main file.

Currently we output all the transformed code generated by reading from a single file to another file.This is a limitation not an issue of our tool, that we require all the code to be in a single file.We have stuck with this implementation as we were facing difficulty modifying the code present in different files, during the Rewriting phase. We could have addressed this limitation if we knew how Source Manager keeps track of the current location code in different files and the use of File ID to modify the code in the appropriate files or to create new files for each transformed code.

We were also facing issues when we passed inputs present in the lonestars package to our transformed versions even the transformed versions given as reference failed for these inputs. Version T4 is the only version that successfully gives an output that matches with the reference results for all the inputs, for version T3 only inputs rmat12.sym.gr(I1) and 2d-2e20.sym.gr(I2) give the results and fails for USA-road-d.FLA.sym.gr(I3) similarly for the transformed version given as reference.For version T2 and T1 we get the correct output only with I1 similar results for the transformed version given as reference. Since we are short on time we do not go further with fixing our transformations to yield the correct output, but these could be fixed given we debug our transformed code.

References

[1] Guoyang Chen, Xipeng Shen - Free Launch: Optimizing GPU Dynamic Kernel Launches through Thread Reuse

[2] https://devblogs.nvidia.com/parallelforall/introduction-cuda-dynamic-parallelism/