Implementing Free Launch Source To Source Compiler

Jalandhar Singh
North Carolina State University
jsingh8@ncsu.edu

Rajeev Menon Kadekuzhi
North Carolina State University
rkadeku@ncsu.edu

ABSTRACT

Dynamic parallelism in a GPU is currently provided using either a software based approach or a hardware based approach. The software based approach is currently difficult to develop and maintain for the developers and could be inefficient in load balancing among threads. The hardware based approach has an overhead when there are sub-kernel launches. The system has to save the state of the parent kernel and then create new threads for the child kernel and once their execution is complete, the system has to restore the state of the parent kernels. The hardware based approach is relatively easier for a developer to develop.

The problem here is to address the inefficiency and the shortcomings that are present in both the approaches.

1. BACKGROUND AND MOTIVATION

Dynamic Parallelism is important for GPU to benefit a broad range of applications. As we have mentioned above, currently it is provided as two different approaches, Software based and Hardware based approach. Neither is satisfactory. Software based approach is complicated to program and is often subject to some load imbalance. Hardware based approach suffers large run time overhead.

To overcome the main drawbacks of both the approaches, we can make use of a new software based approach, *free launch*. With this approach, it would be possible for the developers to develop code that contain sub-kernel launches, and then the code is automatically transformed in such a manner that the sub kernel launches are removed and the parent threads can be reused for the execution that was supposed to be done by the sub-kernel launches. A performance gain of 37 times is expected for the transformed source code compared to the code with sub-kernel launches.

2. OBJECTIVES

The research paper *Free Launch: Optimizing GPU Dynamic Kernel Launches through Thread Reuse*[1] discusses about the above mentioned software based approach. The paper discusses about four different types of transformations and an algorithm to select a transformation that would perform the best for a particular input code and hardware. The following are the different transformations provided in the research paper.

Notations: C for Child, P for parent, B for Thread Block, T for Thread and $\rightarrow$ for Assignment.

1. $CB_* \rightarrow PB_*$: In this transformation, the source code would be transformed such that any sub-task from any of the children can be assigned to any parent thread block. This transformation would be referred as T1 Transformation for the rest of the document.

2. $CK_* \rightarrow PB_*$ In this transformation, the source code would be transformed such that any sub-kernel from any of the children can be assigned to any parent thread block. That is, all the sub-tasks of a sub-kernel is restricted to be assigned to only a parent thread block. This transformation would be referred as T2 Transformation for the rest of the document.

3. $CK_i \rightarrow PB_i$ In this transformation, the source would be transformed such that the child sub-kernel can be assigned to only its parent thread block. That is, any the sub-tasks of a sub-kernel is restricted to be assigned to only its parent thread block. This transformation would be referred as T3 Transformation for the rest of the document.

4. $CK_i \rightarrow PT_i$ In the transformation, the source code would be transformed such that the child sub-kernel can be assigned to only its parent thread. That is, all the sub-tasks of a sub-kernel is restricted to be assigned to only its parent thread. This transformation would be referred as T4 Transformation for the rest of the document.

The objective of the team is to implement the above mentioned transformations.

Section 3 of the document describes the challenges that were faced by the team during the course of the project. Section 4 describes the Solution that the team has come up with. Section 5 describes the lessons and experiences gained while Section 6 specifies the Results from the test cases. Section 7 documents the pending issues and the possible solutions for those issues.

3. CHALLENGES

There were multiple challenges that the team faced during the development phase of the project. The team was new to CUDA. It took some time to understand CUDA and mechanism of using it. Team struggled to get started with CUDA code with sub-kernel launches as it had required special parameters and architecture flags for compilation. Using special parameters and architecture flags required specific hardware. Due to the lack of understanding, it was really difficult to select and use the appropriate hardware for testing the solutions that the team came up with. Understanding the nitty-gritties made it really difficult to get going. Additionally, the lack of understanding in CUDA made it difficult to transform CUDA related parameters like threadIdx.x, blockDim.x and blockIdx.x in the transformations.

Clang/LLVM was also new to the team members. Even though the samples and the tutorial that was provided in the class helped to provide a basic understanding on how to get going, the team found it difficult to resolve multiple issues during development. Once such issue was using Clang on CUDA source code that had sub-kernel launches. The Clang source code does not have support for parsing CUDA code that has sub-kernel launches. The team had to update the Clang source code in order to parse the CUDA source code with sub-kernel launches.

One other challenge that the team had to face with Clang was finding appropriate utility functions in Clang for the desired use cases. There were cases where it was required to retrieve the code between two source locations. As transformations mainly used the Rewriter and Matcher module, much of the concentration for figuring out the ideal utility function for this use case was spend in these classes. Eventually, the team figured out a utility method from the Lexer module.

There isn't much reference material available online only for a source to source Clang compiler. This made it difficult for the team while it faced issues during development. There were use cases to have one matcher in place to match multiple child nodes of a parent that were of same type. For example, there was a need to match two nodes of type 'CXXConstructExpr' that were the descendant of the node 'CUDAKernelCallExpr'. The team could not find a solution with the matcher and had to work around by extracting the information while executing the matcher call back.

Along with the lack of online reference material, there were some Clang material present online that used modules that are now obsolete. These reference materials relied on classes and methods that are currently removed from the Clang source code. Referring to these materials resulted in some confusion and eventually wasted some of the development time.

4. SOLUTION IMPLEMENTED

The solution developed has 6 different components. They are:

1. Sub-kernel Call Matcher Component
2. T1 Transformation Component
3. T2 Transformation Component
4. T3 Transformation Component
5. T4 Transformation Component
6. Parent Call Transformation Component
7. Execution Script Component

The first six components are build with the help of Clang/LLVM while the Execution Script Component is developed utilizing bash script. The Clang components are used to generate binary for 4 different tools:

- free-launch-1: Clang tool for T1 Transformation.
- free-launch-2: Clang tool for T2 Transformation.
- free-launch-3: Clang tool for T3 Transformation.
- free-launch-4: Clang tool for T4 Transformation.

It needs to be noted that the Rewriter and Matcher modules were used for transforming the source code. A design decision was made to utilize the Rewriter module to overwrite the update files. This was done to reduce the complexity when the kernel definitions and the kernel invocations are in different files. This design decision also helps in reducing the complexity of running the tool as with this decision, the tool required only the source file that was dependent on all the other files.

Additionally, it needs to be noted that ReplaceText, InsertTextBefore and InsertTextAfter methods of Rewriters are utilized to replace text between two source locations, insert specific text before a source location and insert specific text after a provided source location respectively.

4.1 Sub-kernel Call Matcher Component

This component pertains to the code that is responsible for matching sub-kernel launches in the source code. The design decision was to come up with one matcher, that would match for any function definition that contains a CUDA kernel launch. This decision meant that the matcher would match for any function definition that has a kernel launch within it. This required the matcher call back to filter the CUDA function definition from normal function definition. The necessary changes for this has been added in the transformation component for each of the four transformation.

The following was the matcher that was used in this component.

```
functionDecl(
    hasDescendant(
        cudaKernelCallExpr(
            allOf(
                hasDescendant(
                    declRefExpr(
                    ).bind("childKernel")
                ),
                hasDescendant(
                    callExpr(
                    ).bind("dimExpr")
                )
            )
        ).bind("kernelCall")
    )
).bind("parentKernel")
```

In order to retrieve the nodes in match call back components, string ids were bound to some of the node matchers that were used in the match finder. The string IDs that were bound are:

1. childKernel: This string ID is used in the subsequent components to identify the child kernel source code location.

2. dimExpr: This string ID is used in the subsequent components to retrieve the CUDA kernel parameters like the grid dimension and block dimension.

3. kernelCall: This string ID is used in the subsequent components to retrieve the location of child kernel invocation within the parent kernel. This ID also helps in retrieving information like the arguments that are passed to the child kernel.

4. parentKernel: This string ID is used in the subsequent components to retrieve the location of the parent kernel. This information would help in updating the parent kernel signature to support more arguments and to translate the parent kernel code as described in the subsequent sections.

All the transformations use the same matcher. To transform the code as per the transformation requirement, the match call back is provided uniquely for each transformation.

4.2 T1 Transformation Component

This component refers to the match call back for the T1 transformation. This component:

1. Makes the parent threads persistent.
2. Transforms the CUDA parameters like threadIdx.x that is used in the parent kernel as desired for the code logic.
3. Saves the arguments and CUDA kernel parameters like block dimension and grid dimension to be saved in some memory location.
4. Adds code to make sure that a parent kernel can execute any sub-task.
5. Adds the sub-kernel source code to the parent kernel, along with code for the parent kernel to select any pending sub-tasks.
6. Transforms the return statement to continue statement in the parent source code so that the parent kernel execution does not complete once it is done with its tasks.
7. Includes the appropriate header file for T1 Transformation.
8. Includes additional parameters for the parent kernel to make it capable for caching child kernel arguments.

The header file that was provided contained multiple macros that are helpful for the transformation. Parent threads were made persistent using the appropriate macros that were provided in the header file. The transformations for CUDA parameters like threadIdx.x and return statements were done by retrieving the parent kernel function body as string and then replacing the keywords/statements with appropriate keywords/statements. The source code that invokes the child kernel was replaced with code that would cache the CUDA kernel parameters for the kernel invocation and the arguments for it. The source code from the child kernel was then inserted along with appropriate macros and code to retrieve the cached arguments at the end of the parent kernel. All these changes were incorporated by utilizing Rewriter module of Clang. Once the source file parsing was complete, this component would add the header file with the macros as the first line for the source code.

4.3 T2 Transformation Component

This component refers to the match call back for the T2 transformation. This component:

1. Makes the parent threads persistent.
2. Transforms the CUDA parameters like threadIdx.x that is used in the parent kernel as desired for the transformation.
3. Saves the arguments and CUDA kernel parameters like block dimension and grid dimension to be saved in some memory location.
4. Adds code to make sure that a parent kernel can execute any sub-kernel. The transformation should be such that the parent kernel would execute the whole set of sub-tasks for a sub-kernel.
5. Adds the sub-kernel source code to the parent kernel, along with code for the parent kernel to select any pending sub-kernel.
6. Transforms the return statement to continue statement in the parent source code so that the parent kernel execution does not complete once it is done with its tasks.
7. Includes the appropriate header file for T2 Transformation.
8. Includes additional parameters for the parent kernel to make it capable for caching child kernel arguments.

The implementation of this component is similar to the T1 Transformation Component defined above. The difference in the transformations is in point 4 mentioned above. In this transformation, the parent kernel source code needs to be transformed such that the parent kernel can execute any pending sub-kernel. It has the restrictions that all the subtasks of a sub-kernel is required to be assigned to only a parent thread block. This requirement is achieved by adding sufficient code while including the child kernel code in the parent kernel code.

4.4 T3 Transformation Component

This component refers to the match call back for the T3 transformation. This component:

1. Makes the parent threads persistent.
2. Saves the arguments and CUDA kernel parameters like block dimension and grid dimension to be saved in some memory location.
3. Adds code to make sure that a parent kernel executes only its sub-kernel.
4. Adds the sub-kernel source code to the parent kernel, along with code for the parent block to execute only its sub-kernels.
5. Transforms the *return* statement to *goto* statement in the parent source code so that the parent kernel execution does not complete once it is done with its tasks. This transformation is required so that on completion of parent kernel tasks, it can execute the appropriate child sub-tasks.

6. Includes the appropriate header file for T3 Transformation.
7. Includes additional parameters for the parent kernel to make it capable for caching child kernel arguments.

The implementation of this component is similar to the T1 Transformation Component defined above. The difference in the transformations is in point 4 mentioned above. In this transformation, the parent kernel source code needs to be transformed such that the parent kernel can execute any pending sub-kernel that are spawned by that parent block. It has the restrictions that the sub-tasks of a sub-kernel is required to be assigned to its parent thread block. This requirement is achieved by adding sufficient code while including the child kernel code in the parent kernel code.

4.5 T4 Transformation Component

This component refers to the match call back for the T4 transformation. This component:

1. Transforms return statements to continue statements in child kernel definition. This transformation is required so that the parent kernel execution does not complete until all the child kernel sub-tasks are completed.
2. Transforms the CUDA kernel parameters like threadIdx.x, blockDim.x, and blockIdx.x.
3. Includes the child kernel definition inside the parent kernel definition.
4. Includes code to execute all the child kernel sub-tasks as part of that parent kernel.

The child invocation is used to retrieve the child kernel definition node. Using these nodes, the child kernel invocation code is replaced with the child kernel definition that is retrieved with the child kernel definition node.

4.6 Parent Call Transformation Component

The transformations components T1 Transformation Component, T2 Transformation Component and T3 Transformation Component updates the signature for the parent kernel. This was done to cache the arguments that were used by the sub-kernel. The transformation components maintained a list of parent kernels in which the transformations were done.

A matcher was added to match any CUDA kernel call. The list maintained by the transformation components is used by the match call back to identify if the signature of the parent kernel call has been updated or not. In the case that the list contains the kernel name, then additional arguments are added to the CUDA kernel invocation. As the additional arguments are used as cache for saving the child kernel arguments, code for a buffer memory declaration and allocation is added before the invocation and code for deallocation is added after the kernel call.

4.7 Execution Script Component

This component is developed using bash script. This component is used to run the Clang source to source compiler tool. A user of the source to source compiler is expected to run this script to run the tool. This component takes 5 arguments.

1. binary: The compiled binary generated by the clang tool is expected for the transformation execution.
2. input: The input source file that needs to be transformed.
3. transformation: The transformation type that needs to be executed.
4. header: The folder where the header files for the transformations are present.
5. compiler_args: The extra arguments that are required for Clang compiler to compile the input source file.

As discussed above, Rewriter modules are used to make the transformations in the code and the transformations are made in the source file as per the design decisions made. This component, on execution would create a new directory at the same level as that of the parent directory of the input file. The contents of the folder are copied over to the new folder and the transformation is executed. This would make sure that the actual source code is not over-written even when the Clang tool is designed to overwrite the changes. The new folder will have the same name as the folder with the input file, with a suffix identifying the transformation.

5. LESSONS AND EXPERIENCES

This project gave the team very valuable lessons. The team came across many new technologies while working in this project. The project gave the team a fast tracked learning curve in CUDA and helped to appreciate the advantages of GPU programming. Clang/LLVM was also new to the team and the team is now comfortable working in Clang tooling. Creating tools in an existing structured framework like Clang provided great lesson to the team on developing projects that are maintainable. The disciplined approach of adding and using utilities in a structured framework was a valuable lesson for the team.

The team had no prior hands-on experience in working with compilers. A totally new experience was provided by this project as we got to work in a source to source compiler. Even though the team members knew about the existence of these types of compilers, we were unaware of the different use cases of utilizing these compilers. The project also provided first hand experience on how a compiler can be utilized to optimize code. The team is now in a position to assume how the industry standard compilers like gcc optimizes code.

Even though the intermediate representation and AST concepts were introduced in the class, working with them in industry standard compiler gave good experience. It helped to consolidate the understanding of parsers and intermediate representations that were introduced in the class. The concept of having matchers and rewriters were also something that the team appreciated in this project.

Lastly, the project introduced a wide variety of tools that are used in the industry like C++ 11, ninja and CMake. Developing code using these helped the team gain good experience in these tools eventually making us better developers than what we were when we started the project.

6. RESULTS

This section covers the results obtained after testing all the transformations. We have divided the tests into two parts:

1. Functional Testing
2. Performance Testing

6.1 Functional Testing

In functional testing, we have tested the correctness of each transformation. We had **identified and added 8 different** test cases apart from the sample tests that were provided to us as part of project material. The following are the different test cases.

Test Id	Test Name
Test 1	Sanity Test
Test 2	Child kernel call with return statement
Test 3	Parent kernel call with return statement
Test 4	Parent kernel call with different type of function arguments
Test 5	Multiple parent function calls inside main function
Test 6	Include multiple files directly into main file
Test 7	Include multiple files recursively into main file
Test 8	Parent function call utilizing multiple grids

1. **Sanity Test (Test 1)**: The basic correctness of each tool is tested with this test case. A CUDA program is written to add two matrices with two kernels such that the one of the kernel(parent) invokes the other kernel(child). The expectation of the test was to transform the parent kernel definition as per the transformation tool that was used. In case of T1, T2 and T3 transformations, the tool was expected to transform the code snippet that invoked the parent kernel to support for the new arguments that would be used to buffer the child kernel arguments. On execution of the tool on the CUDA program, it was verified that the transformed code produced was as expected.

   It needs to be noted that all the subsequent tests specified here, extend this sanity test as per the test case.

2. **Parent or child kernel call with return statement (Test 2 and 3)**: After testing the basic functionality, we have moved to next step where we have added return statements in the child and parent kernel. As we know, each transformation is expected to transform different differently for return statements. T1 and T2 transformation tools are expected to transform return statement as continue statements in parent kernel. T3 transformation tool is expected to transform *return* statement as *goto* statement while T4 transformation tool is expected to not update the return statements in the parent kernel. In case of child kernel, it is expected to transform the return statements in the code to continue for all the transformations. We have verified that both child and parent kernel code transformations and results were as expected.

3. **Parent kernel call with different type of function arguments (Test 4):** In this test, we have added multiple arguments to the child kernel call. The expected behavior is that the parent kernel should correctly buffer the arguments that are provided to child kernel for transformation tools T1, T2 and T3. In case of T4 transformation tool, the arguments need not be saved to any buffer. We have verified that the tools behave as per the expectation.

4. **Multiple parent function calls inside main function (Test 5)**: In this test, we have added multiple parent kernel invocation. T1, T2 and T3 transformation tools allocates a buffer for caching the child kernel arguments. The expectation of the test was to verify that the cache is allocated for all the parent calls, and are reset after the parent kernel completion. We have verified that the tools behave as per the expectation.

5. **Include multiple files directly or recursively into main file (Test 6 and 7)**: Each transformation tries to match for child or parent kernel call. As a result, if main file includes multiple files then each transformation should check for child or parent kernel call in all the included files. Now user can include these files directly or by including one file into another (recursively). In both cases, all the transformations should match and replace the parent or child kernel code with appropriate transformed code. The observed behaviour was as expected.

6. **Parent function call utilizing multiple grids (Test 8)**: In this test, we have added code in the parent kernel to utilize the kernel parameters like blockDim.x and blockIdx.x. T1 and T2 transformation tools are expected to translate these parameters. From the test, it was verified that the behavior of the tools were as expected.

6.2 Performance Testing

This section gives the performance comparison between several set of programs that were generated using different transformation techniques. We have used the original program as a benchmark. As part of the performance testing, we ran a couple of tests to check which transformation is performing better and for the cases where it shows better performance.

For the performance tests, we created the executable for each transformation using nvcc compiler. After creating the binary, we executed the binary with a graph as an input file. Internally each program tries to create a minimum spanning tree based on the content of the graph input file and prints the runtime taken to calculate the minimum spanning tree on the console.

System Environment used for performance testing: We used the ARC cluster to reserve the compute node to execute the performance test. The command used to reserve the compute node is ”srun -p srun -p titanx –pty /bin/bash”. Test code requires architecture compute_35 or greater.

To emulate different scenarios, we executed the binaries with a set of graphs. The following is the list of graphs which were used to run the performance test.

1. rmat12.sym.gr: The graph contains 4096 nodes interconnected by 59320 edges.
2. USA-road-d.NY.gr: The graph contains 264346 nodes interconnected by 730100 edges.
3. USA-road-d.FLA.gr: The graph contains 1070376 nodes interconnected by 2712798 edges.
4. r4-2e20.gr: The graph contains 1048576 nodes interconnected by 4194304 edges.

Note: These graphs were obtained from lonestargpu-2.0 library. User can generate these input files by running "make inputs" command in the lonestargpu directory.

We have used these graphs as input to compare the performance of all the transformations against the original code that was provided. The following table specifies the combination of tests that were run.

Test Id	Test Name
Test 1	Original vs T1 vs T4 with rmat12.sym.gr as the input
Test 2	Original vs T1 Vs T2 vs T3 vs T4 with USA-road-d.NY.gr as a input
Test 3	Original vs T1 vs T2 vs T3 vs T4 with USA-road-d.FLA.gr as a input
Test 4	Original vs T1 vs T2 vs T3 vs T4 with r4-2e20.gr as a input

We have discussed each test case individually as mentioned below:

1. **Original vs T4 vs T1 with rmat12.sym.gr as a input**: In this performance test, we have considered the rmat12.sym.gr graph as an input file to the executable. Here, we have only compared the performance between original code, T1 Transformed and T4 Transformed code. We can see that T4 has performed much better as compared to T1 transformed and Original code. However the performance observed for the original code is slightly better than T1.

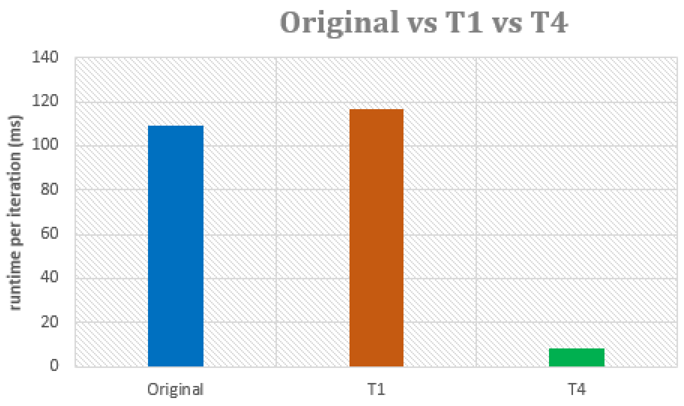

Figure 1: Flowchart - Performance comparison for Original vs T1 vs T4 with rmat12.sym.gr as a input

2. **Original vs T1 Vs T2 vs T3 vs T4 with USA-road-d.NY.gr as a input**: In this test, we have considered all the programs generated using all four transformations for the performance comparison with original code as a benchmark. In this case, T2 has performed slightly better than the original code. Moreover, the performance of T2, T3 and T4 are almost similar, with T4 showing the best result. T1 is the only model that provides poorer performance than the original.

   However, the runtime per iterations for all models seen in this test are significantly more as compared to the previous test. This is because the *USA-road-d.NY.gr* graph contains more nodes and edges as compared to the *rmat12.sym.gr* graph.

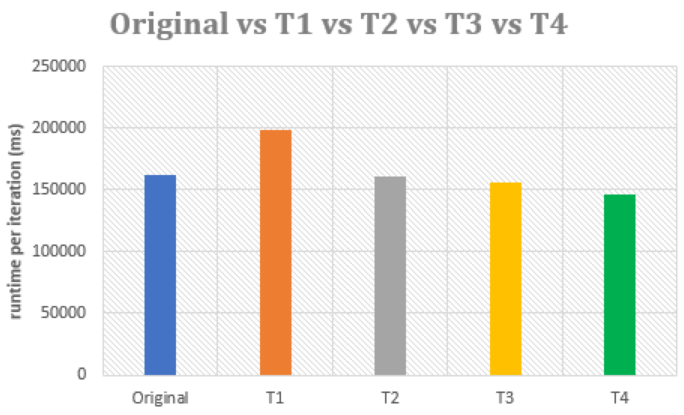

Figure 2: Flowchart - Performance comparison for Original vs T1 vs T2 vs T3 vs T4 with USA-road-d.NY.gr as a input

3. **Original vs T1 vs T2 vs T3 vs T4 with USA-road-d.FLA.gr as a input**: Here it can be observed that T1 and T2 provide a marked improvement over the original code and is the best model. T4, on the other hand, performs poorer.

   As the graph *USA-road-d.FLA.gr* contains nearly ten times as many nodes as *USA-road-d.NY.gr* graph, the runtime per iterations is almost tripled.

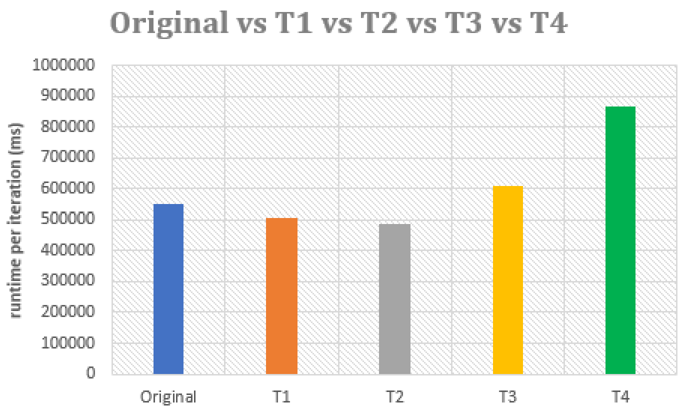

Figure 3: Flowchart - Performance comparison for Original vs T1 vs T2 vs T3 vs T4 with USA-road-d.FLA.gr as a input

4. **Original vs T1 vs T2 vs T3 vs T4 with r4-2e20.gr as a input**: The performance comparison for *r4-2e20.gr* graph is as shown in Figure 4. T3 is the best model for this graph and is nearly 10000ms faster than the next better model (T1).

From the tests performed above, it can be said that the performance of the model is closely related with the testing parameters such as the various inputs that can be provided to the model.

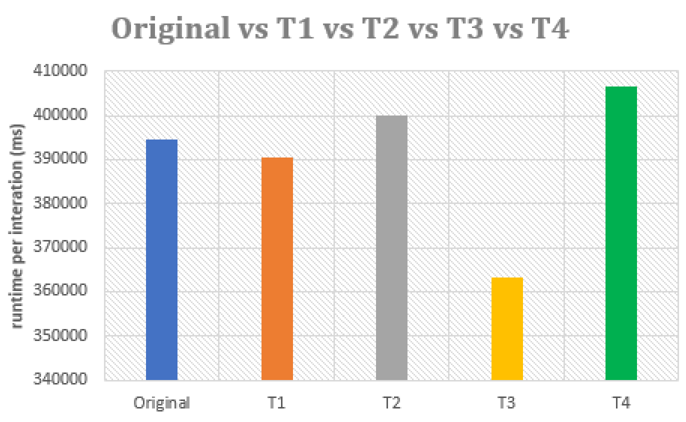

Figure 4: Flowchart - Performance comparison for Original vs T1 vs T2 vs T3 vs T4 with r4-2e20.gr as a input

7. REMAINING ISSUES AND POSSIBLE SOLUTIONS

There are four remaining known issues in the project:

1. Improving Tool Performance: The current implementation of the tool utilizes two matchers and two match call backs. This would result in the tool trying to match two patterns in the AST. The performance of the system can be improved by optimizing two matchers to be combined to one matcher.

2. Transformation Selection: Based on the hardware and the input code, one transformation can be selected to be having the best performance. Currently, development has not been done for dynamically selecting the appropriate transformation.

3. Function Declaration: Clang Tool build fails to transform source code that was function declaration for the parent kernel. As the function signature is updated, every declaration of the parent kernel needs to be translated to account for the new arguments that are added. This issue can be fixed by extending the tool to iterate through all the declarations of the function and then translating it. It would ideally require a new matcher and call back pair to fix the issue.

4. Multidimensional Grid And Thread Blocks: Clang tool build fails to transform source code that utilizes multiple dimensions for CUDA kernel parameters like threadIdx.x, threadIdx.y etc. This issue can be fixed by processing the grid and thread block arguments provided for the sub-kernel launches.

8. REFERENCE

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