The source to source transformation from CUDA to SM-Centric*

†

Haoxu Ren
Department of Computer Science
North Carolina State University
hren3@ncsu.edu

Abstract

This project is aimed to implementing the source to source transformation from CUDA code to SM-Centric code. To achieve this transformation, this report choose the AST matcher provided by Clang. With the matcher and rewriter, transform the input file which means the CUDA code file to SM-Centric code file. In the new SM-Centric file, it includes the new format code, and compared with the original CUDA code, it implements a flexible control of GPU scheduling at the program level becomes feasible, which opens up new opportunities for GPU program optimizations.

Keywords CUDA code, SM-Centric, Clang, LLVM

1 Introduction

Because of the high computational cost of current compute-intensive applications, many scientists view graphic processing units (GPUs) as an efficient means of reducing the execution time of their applications. High-end GPUs include an extraordinary large amount of small computing units along with a high bandwidth to their private on-board memory

The goal of this project is to develop a source-to-source translator that can convert an input CUDA code to a SM-Centric form code. An SM-Centric form has the advantage that it allows the execution of the GPU code to be flexibly controlled in schedule, such as assigning to the same SM a certain set of tasks that share lots of data.

2 Background

A GPU's computing power lies in its abundant memory bandwidth and massive parallelism. However, its hardware thread schedulers, despite being able to quickly distribute computation to processors, often fail to capitalize on program characteristics effectively, achieving only a fraction of the GPU's full potential. Moreover, current GPUs do not allow programmers or compilers to control this thread scheduling, forfeiting important optimization opportunities at the program level.

SM represents Streaming Multiprocessors, with this technique, flexible control of GPU scheduling at the program level becomes feasible, which opens up new opportunities for GPU program optimizations.

An SM-Centric form has the advantage that it allows the execution of the GPU code to be flexibly controlled in schedule, such as assigning to the same SM a certain set of tasks that share lots of data.

In order to implement the automatic tool that transforms source code employing CUDA extensions into plain C code, a source-to-source transformation framework has been leveraged. Different options for this class of source transformations are available nowadays, from simple pattern string replacement tools to frameworks which parse the source code into an Abstract Syntax Tree (AST) and transform the code using that information. Given that our tool needs to do complex transformations involving semantic C++ code information, we have selected the latter.

3 Objective

The objective of this project is to achieve the transformation from CUDA code to SM-Centric code.

This part briefly explains the code transformations involved in the SM-Centric Transformation. In detail, some assumptions taken by us in this explanation are as follows:

* The program contains only one CUDA kernel function, and its definition and invocation are in a single file.

* The grid(...) call is inside the same function that calls the CUDA kernel function.

* The grid in the original program is one or two dimensional.

And in detail, the detailed requirements of this project shown as following:

1. Add the following line to the beginning of the CUDA file after all existing #include statements:

   #include "smc.h"

2. add the following line to the beginning of the definition of the kernel function

   _SMC_Begin

3. add the following line to the end of the definition of the kernel function:

   _SMC_End

4. add the following arguments to the end of the argument list of the definition of the kernel function:

   dim3 __SMC_orgGridDim,
   int __SMC_workersNeeded,
   *int * __SMC_workerCount,*
   *int * _SMC_newChunkSeq,*
   *int * __SMC_seqEnds*

5. in the definition of the kernel function, replace the references of blockIdx.x with

   (int)fmodf((float)__SMC_chunkID, (float)__SMC_orgGridDim.x);

6. in the definition of the kernel function, replace the references of blockIdx.y with

   (int)(__SMC_chunkID/__SMC_orgGridDim.x);

7. Replace the call of function grid(...) with

   dim3 __SMC_orgGridDim(...)

8. add the following right before the call of the GPU kernel function:

   __SMC_init();

9. append the following arguments to the end of the GPU kernel function call:

   __SMC_orgGridDim,
   __SMC_workersNeeded,
   __SMC_workerCount,
   __SMC_newChunkSeq,
   __SMC_seqEnds

4 The Related Work

SM-centric transformation includes two essential techniques. The first is SM-based task selection. In a traditional GPU kernel execution, with or without persistent threads, what tasks a thread executes are usually based on the ID of the thread. While with SM-based task selection, what tasks a thread executes is based on the ID of the SM that the thread runs on. By replacing the binding between tasks and threads with the binding between tasks and SMs, the scheme enables a direct, precise control of task placement on SM.

The second technique is filling-retreating scheme, which offers a flexible control of the amount of active threads on an SM. Importantly, the control is resilient to the randomness and obscuration in GPU hardware thread scheduling. It helps SM-centric transformation in two aspects. First, it ensures an even distribution of active threads on SMs, which is vital for guaranteeing the correctness of SM-centric transformations. Second, it facilitates online determination of the parallelism level suitable for a kernel, which is especially important for the performance of multiple-kernel co-runs, a scenario benefiting significantly from SM-centric transformation.

5 Challenges and Solutions

In this project, I met some challenges during the project period, in this project, I need to use Clang and LLVM, and it's totally new to me, so at the first time, I need to learn some background knowledges about Clang and LLVM.

To solve this challenge, the professor sends me the tutorials about the set-up process of the development environment.

```
mkdir ~/clang-llvm && cd ~/clang-llvm
git clone http://llvm.org/git/llvm.git
cd llvm/tools
git clone http://llvm.org/git/clang.git
cd clang/tools
git clone http://llvm.org/git/clang-tools-extra.git extra
```

Next you need to obtain the CMake build system and Ninja build tool. You may already have CMake installed, but current binary versions of CMake aren't built with Ninja support.

```
cd ~/clang-llvm
git clone https://github.com/martine/ninja.git
cd ninja
git checkout release
./bootstrap.py
sudo cp ninja /usr/bin/

cd ~/clang-llvm
git clone git://cmake.org/stage/cmake.git
cd cmake
git checkout next
./bootstrap
make
sudo make install
```

Figure 1: *The installment of Clang*

After the installation, we will start creating a Clang tool.

First, we'll need to create a new directory for our tool and tell CMake that it exists. As this is not going to be a core clang tool, it will live in the `tools/extra` repository.

```
cd ~/clang-llvm/llvm/tools/clang
mkdir tools/extra/loop-convert
echo 'add_subdirectory(loop-convert)' >> tools/extra/CMakeLists.txt
vim tools/extra/loop-convert/CMakeLists.txt
```

CMakeLists.txt should have the following contents:

```
set(LLVM_LINK_COMPONENTS support)

add_clang_executable(loop-convert
  LoopConvert.cpp
  )
target_link_libraries(loop-convert
  clangTooling
  clangBasic
  clangASTMatchers
  )
```

With that done, Ninja will be able to compile our tool. Let's give it something to compile! Put the following into `tools/extra/loop-convert/LoopConvert.cpp`. A detailed explanation of why the different parts are needed can be found in the LibTooling documentation.

```
// Declares clang::SyntaxOnlyAction.
#include "clang/Frontend/FrontendActions.h"
#include "clang/Tooling/CommonOptionsParser.h"
#include "clang/Tooling/Tooling.h"
// Declares llvm::cl::extrahelp.
#include "llvm/Support/CommandLine.h"

using namespace clang::tooling;
using namespace llvm;

// Apply a custom category to all command-line options so that they are the
// only ones displayed.
static llvm::cl::OptionCategory MyToolCategory("my-tool options");

// CommonOptionsParser declares HelpMessage with a description of the common
// command-line options related to the compilation database and input files.
// It's nice to have this help message in all tools.
static cl::extrahelp CommonHelp(CommonOptionsParser::HelpMessage);

// A help message for this specific tool can be added afterwards.
static cl::extrahelp MoreHelp("\nMore help text...");

int main(int argc, const char **argv) {
  CommonOptionsParser OptionsParser(argc, argv, MyToolCategory);
  ClangTool Tool(OptionsParser.getCompilations(),
                 OptionsParser.getSourcePathList());
  return Tool.run(newFrontendActionFactory<clang::SyntaxOnlyAction>().get());
}
```

Figure 2: *The procedure of creating a ClangTool*

After finishing the set-up procedure of the development environment, the next challenge is to build and run the SM-Centric transformation codes.

And there are some pre-assumptions of this project:

1. The transiator is built and the same environment as the LLVM Virtual Machine provided. Otherwise, the instructions are guaranteed to work.

2. The original CUDA file to be transformed should be with the suffix ".cu" to ensure the naming of generated file.

For Build part:

1. Direct to the Clang tool source folder(/home/ubuntu/llvm/tools/clang/tools)

2. Create a folder named sm-centric

3. Put the SM_centric_transformation.cpp(source code) and CMakeLists.txt(Dependency file) under the sm-centric folder. Both files can be found in the Source Code folder.

4. Modify the CMakeLists.txt in the clang tool folder. Append one line to the end of the file. *add_clang_subdirectory(sm-centric)*

5. Direct to the build-release folder (/home/ubuntu/llvm/build-release)

6. Run *ninja sm-centric*

Once the translator is built successfully, message like following will be displayed.

```
ubuntu@ubuntu:~/llvm/build-release$ ninja sm-centric
[2/2] Linking CXX executable bin/sm-centric
```

Figure 3: *The success of translator built*

For Run part:

We put the transform-needed code into a folder, in my computer, the folder is */home/ubuntu/Downloads,* so we can run the transformation code as following:

```
ubuntu@ubuntu:~/llvm/build-release$ bin/sm-centric  /home/ubuntu/Downloads/matrixMul_org.c
u -- --cuda-host-only
/home/ubuntu/Downloads/matrixMul_org.cu:36:10: fatal error: 'helper_functions.h' file not
found
#include <helper_functions.h>
         ^~~~~~~~~~~~~~~~~~~~
1 error generated.
Error while processing /home/ubuntu/Downloads/matrixMul_org.cu.
```

Figure 4: *The running of the transformation code*

For the transformation code part, to achieve these requirements, I need to use the AST matcher provided by Clang and llvm.

So, I met the first challenge of the code part, I need to write the different matcher for different objective. When analyze the project requirements, for these 9 requirements, it can be divided into serval parts. One is handle for cuda kernel function declaration, one is handle for the call of cuda kernel function, one is handle for member expression declaration, and another is handle for variable declaration. So, I have a basic knowledge of these matchers as following shown:

```
public:
    MyASTConsumer(Rewriter &R) : cudaPrinter(R), HandleForFunction(R),HandleForDecl(R) {
        // The matcher of kernel function
        Matcher.addMatcher(
                functionDecl().bind("kernalfunc"),&HandleForFunction);
        // The matcher of blockIdx.x
        Matcher.addMatcher(
                memberExpr().bind("Vardecl"),&HandleForDecl);
        // The matcher of grid() declaration
        Matcher.addMatcher(
                varDecl().bind("grid"),&HandleForDecl);
        // The matcher of the call of Cuda kernel function
        Matcher.addMatcher(
                cudaKernelCallExpr().bind("cudaCall"), &cudaPrinter);
```

Figure 5: *The original matcher design*

As the figure 5 shown, I design these matchers to achieve the requirements of the project, but it's just the original version, what's next is to perfect these matchers and implement these requirements.

With AST matcher, every statement and declaration in the transform-needed file can be treated as different nodes. If I can get the nodes, then I can get the important source location for the next replacement and inserting. With these information, I can work on implementing these matchers.

```
Matcher.addMatcher(
        functionDecl().bind("kernalfunc"),&HandleForFunction);
// The matcher of blockIdx.x
Matcher.addMatcher(
        memberExpr(hasObjectExpression(hasType(asString("const struct __cuda_builtin_blockIdx_t")))).bind("Vardecl"),&HandleForDecl);
// The matcher of grid() declaration
Matcher.addMatcher(
        varDecl(hasType(asString("dim3")),hasName("grid")).bind("grid"),&HandleForDecl);
// The matcher of the call of Cuda kernel function
Matcher.addMatcher(
        cudaKernelCallExpr().bind("cudaCall"), &cudaPrinter);
```

Figure 6: *The implementation of matcher*

From the figure above, we can see I design these 4 matchers for all requirements, include *functionDecl, memberexpr, vardecl, cudakernelcallexpr.* With these original matcher, I narrow these matchers and get the nodes which met the requirements. Then in the handle implementation part, I use Result.Nodes.getNodeAs() function to locate the chosen node, and then invoke the Rewriter to implement the replacement and inserting.

The following figure shows the one of implementation:

```
class FunctionDeclHandler : public MatchFinder::MatchCallback {
public:
    FunctionDeclHandler(Rewriter &Rewrite) : Rewrite(Rewrite) {}

    virtual void run(const MatchFinder::MatchResult &Result) {
        //get the node 'kernelfunc'
        if(const FunctionDecl *FuncDecl = Result.Nodes.getNodeAs<FunctionDecl>("kernalfunc")){
            // if this function is cuda kernel function
            if(FuncDecl->hasAttr<CUDAGlobalAttr>()){
                // get the function body of this cuda kernel function
                const Stmt *FuncBody = FuncDecl->getBody();
                // Insert the "_SMC_Begin" to the top of the cuda kernel function
                Rewrite.InsertTextAfterToken(FuncBody->getLocStart(), "\n__SMC_Begin:\n");
                // Insert the "_SMC_End" to the bottom of the cuda kernel function
                Rewrite.InsertText(FuncBody->getLocEnd(), "\n__SMC_End: \n", true, true);
                // Initialize a size_t for storing the number of current parameters
                size_t a;
                a=FuncDecl->param_size();
                // get the current parameters of this kernel function
                const ParmVarDecl *para = FuncDecl->getParamDecl(a-1);
                //Add the "dim3 __SMC_orgGridDim, int __SMC_workersNeeded, int *__SMC_workerCount, int * __SMC_newChunkSeq, int * __SMC_seqEnds"
                // to the end of the argument list of the definition of the kernel function
                Rewrite.InsertTextAfterToken(para->getLocEnd(),", dim3 __SMC_orgGridDim, int __SMC_workersNeeded, int *__SMC_workerCount, int * __SMC_newChunkSeq, int * __SMC_seqEnds");
            }
        }
    }

private:
    Rewriter &Rewrite;
};
```

Figure 7: *The implementation of FunctionDeclHandle*

In this implementation, it implements the location of the declaration of CUDA kernel function. The method *hasAttr<CUDAGlobalAttr>()* is the most important part in this location process. This method narrows the range of existing function declaration to the function declaration started with "_global_", which means the CUDA kernel function declaration.

It's exactly what we want. When locate the CUDA kernel function declaration, what the next is to get the body of the function, it can implement by the method: *getbody()*, and with this function, I get the location inside this function which means inside the compounds, so I can add the *"__SMC_Begin"* and the *"__SMC_End"* for this kernel function. In detail, I need to invoke the Rewriter's *InsertText()* method and *InsertTextAfterToken()* method to implement these 2 requirements. In the next, I need to locate the parameter declaration part of this function, so I can use the function *getParamDecl()* to achieve this. Then as the same above, invoke the *InsertTextAfterToken()* method to insert new parameters to the end of parameter declaration location. The new added parameter is of the SM-Centric format, which is like:

*dim3 __SMC_orgGridDim, int __SMC_workersNeeded, int *__SMC_workerCount, int * __SMC_newChunkSeq, int * __SMC_seqEnds*

With these descriptions above, I implemented the transformation for the CUDA kernel function declaration part. And I will show the result part in the next Result part.

And in this part, I will continue show the resting implementation of other matcher handlers.

For Variable Declaration Handler:

```
class VarDeclHandler : public MatchFinder::MatchCallback {
public:
    VarDeclHandler(Rewriter &Rewrite) : Rewrite(Rewrite) {}

    virtual void run(const MatchFinder::MatchResult &Result) {
        //get the node "Vardecl"
        if(const MemberExpr *Var = Result.Nodes.getNodeAs<MemberExpr>("Vardecl")){
            //get the right side name of blockIdx.x
            std::string name = Var->getMemberDecl()->getNameAsString();
            //if the right side's name is x
            if(name == "__fetch_builtin_x")
                //replace the "blockIdx.x" with "(int)fmodf((float)__SMC_chunkID, (float)__SMC_orgGridDim.x)"
                Rewrite.ReplaceText(Var->getLocEnd().getLocWithOffset(-9),11,"(int)fmodf((float)__SMC_chunkID, (float)__SMC_orgGridDim.x);\n");
            // if the right side's name is y
            if(name == "__fetch_builtin_y")
                //replace the "blockIdx.y" with "(int)(__SMC_chunkID/__SMC_orgGridDim.x)"
                Rewrite.ReplaceText(Var->getLocEnd().getLocWithOffset(-9),11,"(int)(__SMC_chunkID/__SMC_orgGridDim.x);\n");
        }

        //get the node "grid"
        if(const VarDecl *Var = Result.Nodes.getNodeAs<VarDecl>("grid")){
           // get the sourcelocation of the call of function "grid(...)"
            SourceLocation sr = Var->getLocStart().getLocWithOffset(4);
            // replace the 'grid(...)' with "dim3 __SMC_orgGridDim(...)"
            Rewrite.ReplaceText(sr,Var->getNameAsString().length()," __SMC_orgGridDim");
        }
    }

private:
    Rewriter &Rewrite;
};
```

Figure 8: *The implementation of VariableDeclarationHandler*

For the call of CUDA kernel function Handler:

```
class CudaCallPrinter : public MatchFinder::MatchCallback {
public :
    CudaCallPrinter(Rewriter &Rewrite) : Rewrite(Rewrite) {}

    virtual void run(const MatchFinder::MatchResult &Result) {
        //get the node 'cudaCall'
        if (const Stmt *S = Result.Nodes.getNodeAs< clang::Stmt>("cudaCall")){
            // Insert the "_SMC_init()" right before the call of the GPU kernel function
            Rewrite.InsertText(S->getLocStart(), "\n __SMC_init();\n",true,true);
            // if this function call is the call of CUDA kernel function
            if( const CUDAKernelCallExpr *cudafunc =  dyn_cast< CUDAKernelCallExpr>(S)){
                // get the location of this call's arguments
                const Stmt *arg = cudafunc->getArg(cudafunc->getNumArgs()-1);
                // append the following arguments to the end of the GPU kernel function call
                Rewrite.InsertText(arg->getLocStart().getLocWithOffset(1), ", __SMC_orgGridDim, __SMC_workersNeeded, __SMC_workerCount, __SMC_newChunkSeq, __SMC_seqEnds",true,true);
            }
        }

    }
private:
    Rewriter &Rewrite;
};
```

Figure 9: *The implementation of the call of CUDA kernel function*

6 The remaining issues and possible solutions

During the period of the whole project, there is a remaining issue left.

For the matcher of the call of CUDA kernel function, it implements the matcher for finding the position of call of CUDA kernel function.

```
// The matcher of the call of Cuda kernel function
Matcher.addMatcher(
        cudaKernelCallExpr().bind("cudaCall"), &cudaPrinter);
```

Figure 10: *The matcher of the call of CUDA kernel function*

```
class CudaCallPrinter : public MatchFinder::MatchCallback {
public :
    CudaCallPrinter(Rewriter &Rewrite) : Rewrite(Rewrite) {}

    virtual void run(const MatchFinder::MatchResult &Result) {
        //get the node 'cudaCall'
        if (const Stmt *S = Result.Nodes.getNodeAs< clang::Stmt>("cudaCall")){
            // Insert the "_SMC_init()" right before the call of the GPU kernel function
            Rewrite.InsertText(S->getLocStart(), "\n __SMC_init();\n",true,true);
            // if this function call is the call of CUDA kernel function
            if( const CUDAKernelCallExpr *cudafunc =  dyn_cast< CUDAKernelCallExpr>(S)){
                // get the location of this call's arguments
                const Stmt *arg = cudafunc->getArg(cudafunc->getNumArgs()-1);
                // append the following arguments to the end of the GPU kernel function call
                Rewrite.InsertText(arg->getLocStart().getLocWithOffset(1), ", __SMC_orgGridDim, __SMC_workersNeeded, __SMC_workerCount, __SMC_newChunkSeq, __SMC_seqEnds",true,true);
            }
        }

    }
private:
    Rewriter &Rewrite;
};
```

Figure 11: *The implementation of the call of CUDA kernel function*

According to these figures shown above, we can see the detailed implementation of the call of CUDA kernel function.

The CUDA code need to be transformed is showed as following:

```
// Performs warmup operation using matrixMul CUDA kernel
if (block_size == 16)
{
    matrixMulCUDA<16><<< grid, threads >>>(d_C, d_A, d_B, dimsA.x, dimsB.x);
}
else
{
    matrixMulCUDA<32><<< grid, threads >>>(d_C, d_A, d_B, dimsA.x, dimsB.x);
}
```

Figure 11: *The code need to be transformed*

With the requirements, the after-transform code will add the *"__SMC_init()"* above the call of CUDA kernel function, which means the *matrixMulCUDA<16* and *matrixMulCUDA<32>*, and what's more? I also need to add the following arguments to the end of the GPU kernel function call:

__SMC_orgGridDim, __SMC_workersNeeded, __SMC_workerCount,__SMC_newChunkSeq,__SMC_seqEnds

And what we need to get is as the following:

```
/* SMC change: a line added */
__SMC_init();

// Performs warmup operation using matrixMul CUDA kernel
if (block_size == 16)
{
  /* SMC change: added four parameters to the call at the end */
  //debug
  //printf("call kernel block_size=16\n");
  matrixMulCUDA<16><<< grid, threads >>>(d_C, d_A, d_B, dimsA.x, dimsB.x, __SMC_orgGridDim, __SMC_workersNeeded, __SMC_workerCount, __SMC_newChunkSeq, __SMC_seqEnds); /* SMC change:five extra parameters are used at the end*/
}
else
{
  /* SMC change: added four parameters to the call at the end */
  //debug
  //printf("call kernel block_size=32\n");
  matrixMulCUDA<32><<< grid, threads >>>(d_C, d_A, d_B, dimsA.x, dimsB.x, __SMC_orgGridDim, __SMC_workersNeeded, __SMC_workerCount, __SMC_newChunkSeq, __SMC_seqEnds); /* SMC change: five extra parameters are used at the end*/
}
```

Figure 12: *The expected result*

When I run my sm-transformation code, I will get the following result:

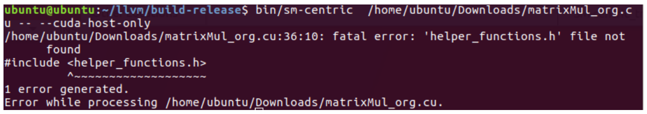

Figure 13: *The screen shot when running*

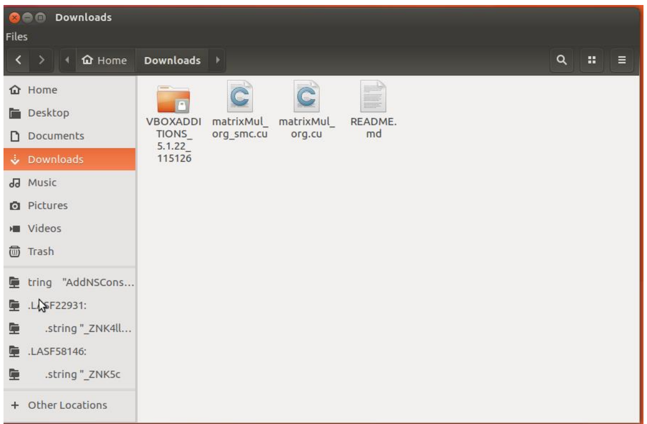

Figure 14: *The screen shot after running*

From these screen shots, we can see that there is an error generated, it said the compiler can't find the helper_functions.h file. But it still output the after-transform codes, which named "matrixMul_org_smc.cu". When I open this file, and I found this transformation is not successful, the output codes shown as following:

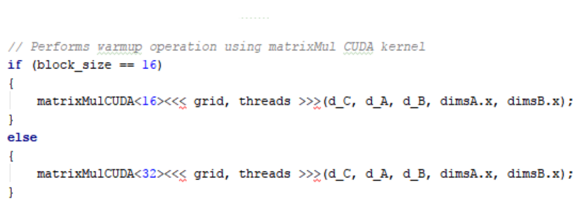

Figure 15: *The screen shot of the result file*

So, to solve this problem, I find the *helper.h* files from the professor. And move these all .h files to the /usr/include folder:

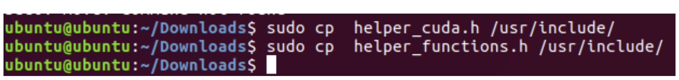

Figure 16: *The screen shot of move files*

After that, re-running the codes, as the following figures show, there is no error happened.

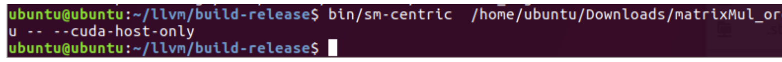

Figure 17: *The screen shot of re-running codes*

And we open the newly output file: matrixMul_org_smc.cu, check the output file:

Figure 18: *The result file*

From this output file, we can see this transformation is successful.

7 Results

I use a CUDA file named *matrixAdd_org.cu* for test case, and the original cuda code is showed as following:

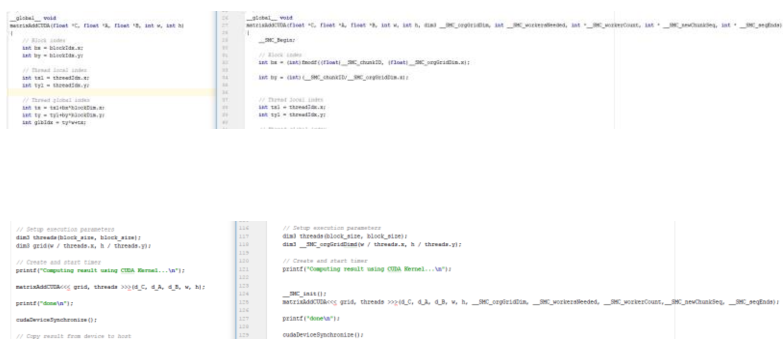

Figure 19: *The result file*

According to the output file, we can see the all transformation-needed part is successfully transformed. That means my transformation codes is useful and can run correctly.

References

[1] NVIDIA CUDA. http://www.nvidia.com/cuda.

[2] J. T. Adriaens, K. Compton, N. S. Kim, and M. J. Schulte. The Case for GPGPU Spatial Multitasking. In Proceedings of HPCA, 2012.

[3] Wu B, Chen G, Li D, et al. Enabling and Exploiting Flexible Task Assignment on GPU through SM-Centric Program Transformations[C]//ACM on International Conference on Supercomputing. ACM, 2015:119-130.

[4] Wu B, Chen G, Li D, et al. SM-centric transformation: Circumventing hardware restrictions for flexible GPU scheduling[C]// International Conference on Parallel Architecture and Compilation Techniques. IEEE, 2017:497-498

[5] The tutorial of Clang and LLVM, https://clang.llvm.org/docs/LibASTMatchersTutorial.html

[6] Carlos Rea˜no, A CUDA-to-rCUDA Converter, Master's Thesis in Software Engineering, Formal Methods and Information Systems Department of Information System and Computation.

[7] T. Aila and S. Laine. Understanding the eciency of ray traversal on gpus. In Proceedings of the Conference on High Performance Graphics 2009, HPG '09, 2009.

[8] G. Chen, B. Wu, D. Li, and X. Shen. Porple: An extensible optimizer for portable data placement on gpu. In Proceedings of MICRO, 2014.

[9] L. Chen, O. Villa, S. Krishnamoorthy, and G. Gao. Dynamic load balancing on single-and multi-gpu systems. In IPDPS, 2010.

[10] R. Nasre, M. Burtscher, and K. Pingali. Morph algorithms on gpus. In PPOPP, 2013.

[11] M. E. Belviranli, L. N. Bhuyan, and R. Gupta. A dynamic self-scheduling scheme for heterogeneous multiprocessor architectures. ACM Trans. Archit. Code Optim., 9(4), Jan. 2013.