SM-centric Transformation Translator

Xiangqing Ding
North Carolina State University
Raleigh, NC, the United States, 27518
xding3@ncsu.edu

Abstract

With the increasing popularity of GPU programming, the need of managing thread scheduling becomes more significant. In this project, a source-to-source translator that can convert CUDA code to an SM-Centric form is designed and implemented.

Keywords
CUDA; SM-centric transformation; Clang;

1. Introduction

In the CUDA programming environment, the thread scheduling is managed by hardware scheduler, which is not exposed to the programmer. As a result, it limits the programmer's ability to exploit workload communication patterns for having no means of intervening how threads are scheduled onto the SM [4]. To overcome this situation, an SM-centric transformation method is proposed, which enables spatial scheduling of GPU tasks for the first time [1].

For the purpose of automatically transforming original CUDA code into SM-centric form, a source-to-source translator is designed, implemented and evaluated in this project.

For minimizing the load of implementation, LLVM is utilized as the infrastructure framework combined with Clang as the front end. In fact, all the transformation and file generation tasks are done within Clang. A Clang tool is implemented in C++ language which traverses and rewrites the AST tree generated by Clang driver.

For the basic evaluation, the translator can be successfully run and should guarantee that generated code could be executed successfully and perform the same functionality as the origin code. As a consequence, some test cases are implemented. In addition, the test cases should guarantee that the translator can handle different situations.

2. Background

2.1 GPU

2.1.1 History

Graphics Processing Units (GPU) was originally designed for graphics processing and firstly launched by NVIDIA in 1999 [2]. Around 2006, GPU became completely programmable with NVIDIA releasing CUDA language, which can be used for writing non-graphics programs that will run on GPUs. Now it gains it popularity for being utilized in computational power for non-graphics applications.

2.1.2 Structure

In GPU, a grid is an executing instance of a kernel and organized as a two-dimension array of blocks. Each block defines an independent task that executes in a task parallel way and is organized as three-dimension array of threads. Each thread can cooperate with each other by synchronizing their execution using barrier and efficiently sharing data through a low latency shared memory. Two threads from two different blocks cannot cooperate with each other and all threads in a grid execute the same kernel function

2.2 CUDA

2.2.1 Overview

CUDA (Compute United Device Architecture) is a parallel computing platform created by NVIDIA that allows software developers to utilize GPU for general purpose processing. In this section, some important CUDA concepts or syntaxes related to this project are described.

2.2.2 Kernel Function

In CUDA programming, functions can be defined as kernels (also called kernel function), which run on the device and can be called from the host. A kernel function is defined using the *__global__* declaration specifier. [10]

2.2.3 Execution Configuration

To identify the call of a kernel function, an execution configuration for that call must be specified, which defines the dimension of the grid and blocks used to execute the function on the device. The execution configuration is specified by inserting an expression (i.e. <<< *Dg, Db, Ns, S* >>>) between the function name and arguments.

In the expression, *Dg* specifies the dimension and size of the grid and *Db* specifies the dimension and size of each block. *Ns* specifies the number of bytes in shared memory that is dynamically allocated per block for this call as well as the statically allocated memory. And *S* specifies the associated stream. Usually *Dg* and *Db* are necessary for the execution configuration and can be of type *int* or *dim3*. [10] However, if an integer value is passed to the execution configuration, it would be cast to the type of *dim3* within the kernel function. And any component left unspecified will be initialized to 1 by default.

2.2.4 Built-in Variables

Each thread executing the kernel is assigned with a unique thread ID, which can be accessed by the built-in variable *threadIdx*. In addition, blocks within the grid are organized into a one- dimensional, two-dimensional, or three-dimensional way, and they can be accessed through the built-in variable *blockIdx*. The number of the threads per block is accessible through the built-in variable *blockDim* while the number of blocks per grid can be accessed through the built-in variable *gridDim*. [10]

2.3 Clang

2.3.1 Overview

This section is mainly about introducing Clang and its AST (Abstract Syntax Tree) structure, based on which this translator is built. It is aimed at helping reader better understand how Clang works.

2.3.2 Introduction

Clang is a compiler front-end for the C language family. It is written in modern C++ and uses LLVM as its backend. Instead of being a monolithic compiler, Clang has a library-based and modular design. It supports different uses such as code refactoring, static analysis and code generation. Among the various dialects of the C language family supported, Clang now allows to parse CUDA, which is a fundamental feature for this project. [9]

2.3.3 Clang AST

An AST (Abstract Syntax Tree) is a tree representation of the abstract syntactic structure of source code written in a programming language.

There are four basic nodes in Clang AST, that is *Decl* (Declaration), *Stmt* (Statement), *Expr* (Expression) and *Type* (Type). Other types of nodes, such as *FunctionDecl* and *CompoundStmt*, are mainly derived from these four node types. To be noted, *Expr* is also derived from *Stmt*.

The main entry point into the Clang AST are the translation units, which is identified by *TranslationUnitDecl* node.

2.3.4 Clang Tools

Clang Tools are standalone command line tools, offering developer-oriented functionality such as syntax checking and refactoring. Clang tools are developed in three components: the underlying infrastructure for building a standalone tool based on Clang, core shared logic used by many different tools in the form of refactoring and rewriting libraries, and the tools themselves. [6]

3. Related Work

Another way of improving GPU scheduling is using a style of programming for the GPU called Persistent Threads. In the persistent threads, threads are kept active throughout the execution of a kernel. In addition, it uses working queues: when a block finishes, it checks the queue for more work and continues doing so until no work left [4]. However, it still does not determine the on which locations tasks run.

There exist several projects that translate from the CUDA to other language. Since these projects deals with the same syntax as this project, sometimes it provides useful solutions to the problems occurred in this project.

Gabriel Martinez, Mark Gardner, and Wu-chun Feng created an automated CUDA-to-OpenCL translator named CU2CL [5]. This project is also built upon the Clang compiler framework. CU2CL takes an application's CUDA source files and rewrites them into equivalent OpenCL host and kernel files. Their implementation of the project is public in GitHub, and their idea about using recursive methods helps me solve the problem of finding member expression within specified function declaration.

4. Design

4.1 Architecture Overview

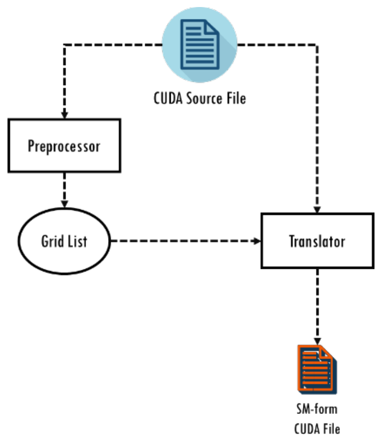


Figure 1. Architecture

The overall translation process is shown in the Figure 1. There are two main components in this process: Preprocessor and Translator. In the implementation, the two components are in the same source file, which would be introduced later. In addition, the translator generates a new file containing the SM-centric form of original CUDA code instead of directly working on the original CUDA file.

Since both Preprocessor and Translator are implemented as Clang tool, they heavily rely on the three interfaces: *Frontend Action*, *ASTConsumer* and *RecursiveASTVisitor*, which form the basic structure of the translator.

When writing a Clang based tool based on *LibTooling*, the common entry point is the *FrontendAction*, which is an interface allowing execution of user specific actions as part of the compilation. [7] By overriding the *EndSourceFileAction()* in *FrontendAction*, the buffer of *Rewriter* is sent to the output file stream when the transformation process is over.

ASTConsumer is an interface for writing actions on an AST, regardless of how the AST was produced. *ASTConsumer* provides many different entry points, but for this use case the only one needed is *HandleTranslationUnit*, which is called with the *ASTContext* for the translation unit. [7]

The *RecursiveASTVisitor* provides hooks of the form *bool VisitNodeType(NodeType *)* for most AST nodes to extract the relevant information. For example, the *VisitDecl(Decl *decl)* visits all the *Decl* nodes in the AST and carries out corresponding actions. [7]

4.2 Preprocessor

In the translation process, the original CUDA file would be firstly processed by the Preprocessor, which is implemented as a Clang tool. The Preprocessor does no refactoring tasks and is added to retrieve all the argument name of *Dg* (See *2.2.3*

Execution Configuration) in execution configuration before the Translator is run.

The Preprocessor traverse the AST of original CUDA code generated by Clang driver, and visits all the node of calls for CUDA kernel function and their execution configurations. Then the variable names of *Dg* (See *2.2.3*) will be stored in a global string list. Then the list will be used by the Translator to do transformation tasks.

4.3 Translator

Translator carries out all the transformation tasks of the SM-transformation, which will be described in next section. It also takes in the original CUDA file and The AST is traversed again in a recursive descent way. Since it is implemented within the same source file as the Preprocessor, it can access the global grid list. And in this step, a new file will be created for holding the transformed CUDA code.

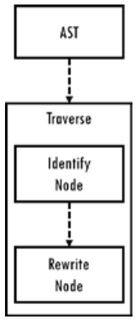


Figure 2. Traversing stages

5. Implementation

5.1 Overview

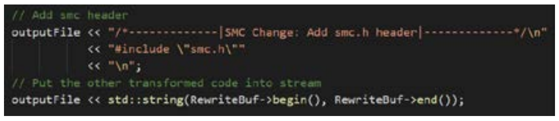
In this section, the whole implementation process will be described task by task. Experiences will also be descripted in this section

5.2 Tasks

5.2.2 Adding Header

The first task of SM-centric transformation is adding the *#include "smc.h"* to the beginning of the original CUDA file.

When the generated file is created, the statement is firstly added to the output file stream so that the generated CUDA file always begins with this statement.



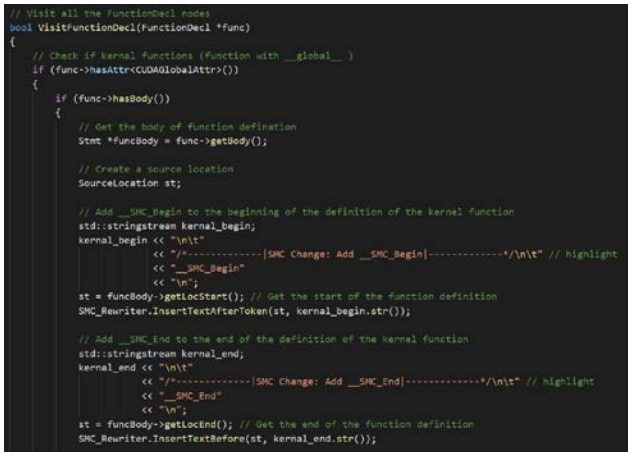
```
// Add smc header
outputFile << "/*-------------|SMC Change: Add smc.h header|-------------*/\n"
           << "#include \"smc.h\""
           << "\n";
// Put the other transformed code into stream
outputFile << std::string(RewriteBuf->begin(), RewriteBuf->end());
```

Figure 3. Snippet

5.2.3 Adding __SMC_Begin and __SMC_End

The next transformation is adding *__SMC_Begin* to the beginning and *__SMC_End* to the end of the definition of the kernel function. To do this task, the first thing is to locate the CUDA kernel function declaration. As stated before, in CUDA programming, the kernel function has a *__global__* attribute. To check whether current function declaration is a kernel function declaration, *FuncDecl->hasAttr<CUDAGlobalAttr>()* is called when a *FunctionDecl* node is visited.

After the declaration of kernel function is located, it is simple to retrieve the start and the end of the function definition as well as inserting text.

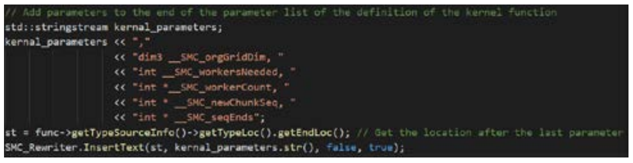


```
// Visit all the FunctionDecl nodes
bool VisitFunctionDecl(FunctionDecl *func)
{
    // Check if kernal functions (function with __global__ )
    if (func->hasAttr<CUDAGlobalAttr>())
    {
        if (func->hasBody())
        {
            // Get the body of function defination
            Stmt *funcBody = func->getBody();

            // Create a source location
            SourceLocation st;

            // Add __SMC_Begin to the beginning of the definition of the kernel function
            std::stringstream kernal_begin;
            kernal_begin << "\n\t"
                         << "/*-------------|SMC Change: Add __SMC_Begin|-------------*/\n\t" // highlight
                         << "__SMC_Begin"
                         << "\n";
            st = funcBody->getLocStart(); // Get the start of the function definition
            SMC_Rewriter.InsertTextAfterToken(st, kernal_begin.str());

            // Add __SMC_End to the end of the definition of the kernel function
            std::stringstream kernal_end;
            kernal_end << "\n\t"
                       << "/*-------------|SMC Change: Add __SMC_End|-------------*/\n\t" // highlight
                       << "__SMC_End"
                       << "\n";
            st = funcBody->getLocEnd(); // Get the end of the function definition
            SMC_Rewriter.InsertTextBefore(st, kernal_end.str());
```

Figure 4. Snippet

5.2.4 Adding Parameters

Continuing from the previous task, the next step is adding the parameters to the end of the argument list of the definition of the kernel function. To locate the parameters of function, we firstly get the *TypeSourceInfo* of the function. And then the location after the last parameter can be retrieved by calling *TypeSourceInfo->getTypeLoc().getEndLoc()*. In the end, new parameters could be appended.



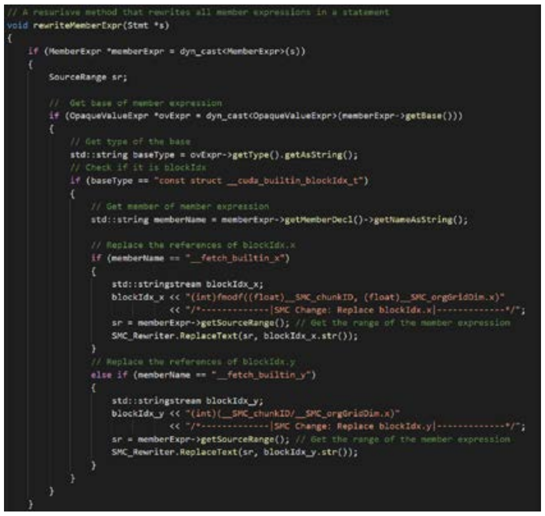
```
// Add parameters to the end of the parameter list of the definition of the kernel function
std::stringstream kernal_parameters;
kernal_parameters << ","
                  << "dim3 __SMC_orgGridDim, "
                  << "int __SMC_workersNeeded, "
                  << "int *__SMC_workerCount, "
                  << "int * __SMC_newChunkSeq, "
                  << "int * __SMC_seqEnds";
st = func->getTypeSourceInfo()->getTypeLoc().getEndLoc(); // Get the location after the last parameter
SMC_Rewriter.InsertText(st, kernal_parameters.str(), false, true);
```

Figure 5. Snippet

5.2.5 Replacing the References of Built-in Variables

The last step of modification within CUDA kernel function declaration is replacing all the references of *blockIdx.x* and *blockIdx.y*, which are apparently member expressions. As stated before, these two variables are also built-in variables and have specified type.

To ensure that only the references in the kernel function are replaced, a recursive function is implemented to visit every child node of the *FuncDecl* node. In the recursive method, it would firstly try cast the input, which is a *Stmt* object, to the *MemberExpr* object. If it succeeds, it means the object is a member expression and then the expression would be judged whether it is the *blockIdx.x* or *blockIdx.y*. After locating the references, it is simple to replace them with target expression.

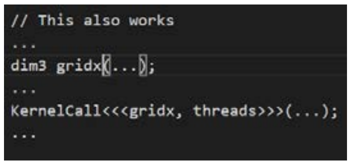


```
// A recursive method that rewrites all member expressions in a statement
void rewriteMemberExpr(Stmt *s)
{
    if (MemberExpr *memberExpr = dyn_cast<MemberExpr>(s))
    {
        SourceRange sr;

        //  Get base of member expression
        if (OpaqueValueExpr *ovExpr = dyn_cast<OpaqueValueExpr>(memberExpr->getBase()))
        {
            // Get type of the base
            std::string baseType = ovExpr->getType().getAsString();
            // Check if it is blockIdx
            if (baseType == "const struct __cuda_builtin_blockIdx_t")
            {
                // Get member of member expression
                std::string memberName = memberExpr->getMemberDecl()->getNameAsString();

                // Replace the references of blockIdx.x
                if (memberName == "__fetch_builtin_x")
                {
                    std::stringstream blockIdx_x;
                    blockIdx_x << "(int)fmodf((float)__SMC_chunkID, (float)__SMC_orgGridDim.x)"
                               << "/*-------------|SMC Change: Replace blockIdx.x|-------------*/";
                    sr = memberExpr->getSourceRange(); // Get the range of the member expression
                    SMC_Rewriter.ReplaceText(sr, blockIdx_x.str());
                }
                // Replace the references of blockIdx.y
                else if (memberName == "__fetch_builtin_y")
                {
                    std::stringstream blockIdx_y;
                    blockIdx_y << "(int)(__SMC_chunkID/__SMC_orgGridDim.x)"
                               << "/*-------------|SMC Change: Replace blockIdx.y|-------------*/";
                    sr = memberExpr->getSourceRange(); // Get the range of the member expression
                    SMC_Rewriter.ReplaceText(sr, blockIdx_y.str());
                }
            }
        }
    }
}
```

Figure 6. Snippet

5.2.6 Replacing Grid Declaration

This is the most difficult task of the implementation. The problem is that the grid variable to be used in execution configuration is not necessarily named as "grid". For instances, following statements show in Figure 7 also works.

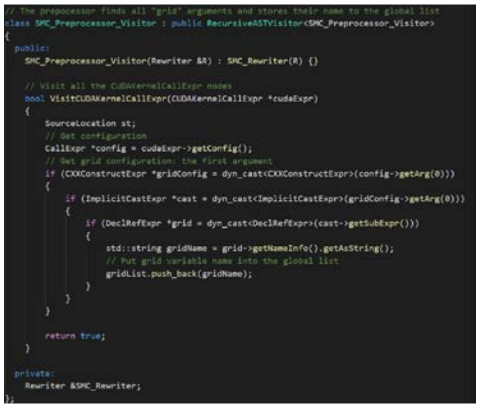


```
// This also works
...
dim3 gridx(...);
...
KernelCall<<<gridx, threads>>>(...);
...
```

Figure 7. Example

As a consequence, we need to retrieve the grid variable name from each call for CUDA kernel functions. And then we check each *VarDecl* node with the grid name list to see whether it is the corresponding declaration of the grid variable.

To solve this problem, a Preprocessor is implemented, which is run before the Translator, to put all grid variables name in a global string list. Then when the translator runs, it can access the complete grid list.

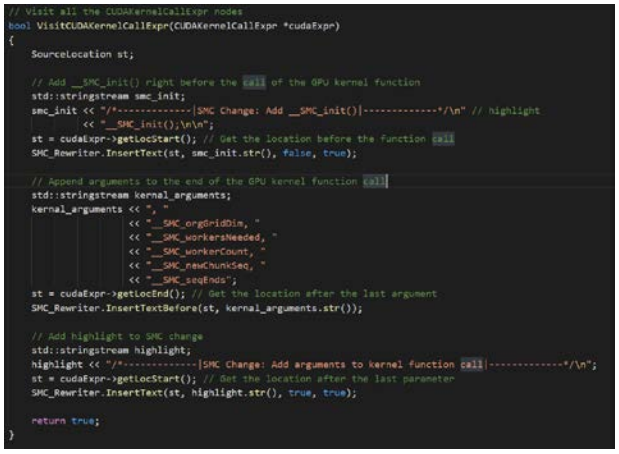


```
// The prepocessor finds all "grid" arguments and stores their name to the global list
class SMC_Preprocessor_Visitor : public RecursiveASTVisitor<SMC_Preprocessor_Visitor>
{
  public:
    SMC_Preprocessor_Visitor(Rewriter &R) : SMC_Rewriter(R) {}

    // Visit all the CUDAKernelCallExpr nodes
    bool VisitCUDAKernelCallExpr(CUDAKernelCallExpr *cudaExpr)
    {
        SourceLocation st;
        // Get configuration
        CallExpr *config = cudaExpr->getConfig();
        // Get grid configuration: the first argument
        if (CXXConstructExpr *gridConfig = dyn_cast<CXXConstructExpr>(config->getArg(0)))
        {
            if (ImplicitCastExpr *cast = dyn_cast<ImplicitCastExpr>(gridConfig->getArg(0)))
            {
                if (DeclRefExpr *grid = dyn_cast<DeclRefExpr>(cast->getSubExpr()))
                {
                    std::string gridName = grid->getNameInfo().getAsString();
                    // Put grid variable name into the global list
                    gridList.push_back(gridName);
                }
            }
        }

        return true;
    }

  private:
    Rewriter &SMC_Rewriter;
};
```

Figure 8. Preprocessor Snippet

5.2.7 Adding __SMC_init() and Appending Arguments

The next task is adding *__SMC_init()* before each call for the kernel functions and adding arguments to the call. The task becomes easy since there is specified node for the calls, that is *CUDAKernelCallExpr*. We just need to finish the hook function and get corresponding location for inserting.



```
// Visit all the CUDAKernelCallExpr nodes
bool VisitCUDAKernelCallExpr(CUDAKernelCallExpr *cudaExpr)
{
    SourceLocation st;

    // Add __SMC_init() right before the call of the GPU kernel function
    std::stringstream smc_init;
    smc_init << "/*-------------|SMC Change: Add __SMC_init()|-------------*/\n" // highlight
             << "__SMC_init();\n\n";
    st = cudaExpr->getLocStart(); // Get the location before the function call
    SMC_Rewriter.InsertText(st, smc_init.str(), false, true);

    // Append arguments to the end of the GPU kernel function call
    std::stringstream kernal_arguments;
    kernal_arguments << ", "
                     << "__SMC_orgGridDim, "
                     << "__SMC_workersNeeded, "
                     << "__SMC_workerCount, "
                     << "__SMC_newChunkSeq, "
                     << "__SMC_seqEnds";
    st = cudaExpr->getLocEnd(); // Get the location after the last argument
    SMC_Rewriter.InsertTextBefore(st, kernal_arguments.str());

    // Add highlight to SMC change
    std::stringstream highlight;
    highlight << "/*-------------|SMC Change: Add arguments to kernel function call|-------------*/\n";
    st = cudaExpr->getLocStart(); // Get the location after the last parameter
    SMC_Rewriter.InsertText(st, highlight.str(), true, true);

    return true;
}
```

Figure 9. Snippet

5.2.9 Extra Features

In our implementation, there are other features achieved though not required. One interesting feature is that the generated file is in the same path as the original CUDA file. And in the generated CUDA file, highlight comments are added around the SM-centric changes to notify readers.

Also, some assumptions are relaxed. For example, the grid in the original program now could be one-dimension, two-dimension or three-dimension. In addition, the translator can work on the original file containing multiple kernel functions.

6. Challenges and Solutions

The biggest challenge is definitely using Clang. Learning how to program within Clang costs most of time of this project, even though the examples provided does show a basic structure of the translator. There are so many Clang classes and methods. The hierarchies and relationships between classes are complex especially for the multiple inheritance feature of C++. Furthermore, the official Clang documentation is not detailed in many parts such as method description. To overcome this, I tried to collect unofficial document and had to utilize them, such as blog, course slides and so on.

Another challenge is the understanding of SM-centric transformation. The project document describes how the code are transformed step by step, so the implementation is mainly based on the description instead of understanding. To make sure that the project meets the requirement, a deep study about SM-centric is needed in future development.

There are also many small problems. One of the problem is locating CUDA kernel function declaration, since there are no specified types of node for it. However, as stated before, the kernel function is specified with qualifier *__global__*. The solution is stated in the previous part (See *5.2.2*).

Another problem I met is the implementation of replacing grid declaration, which is stated in the previous section (See *5.2.6*).

7. Limitations and Future Work

For current implementation, the only limitation is that the *#include "smc.h"* statement is added at the beginning of the generated file instead of being inserted after all existing include statements. This doesn't make any difference to current test cases. However, it can be achieved in future work by extending *PPCallBacks* and overriding the *IncludeDirective()* method.

Reference

[1] B. Wu, G. Chen, D. Li, X. Shen and J. S. Vetter, "SM-centric transformation: Circumventing hardware restrictions for flexible GPU scheduling," 2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT), Edmonton, AB, 2014, pp. 497-498

[2] Anon. NVIDIA Launches the World's First Graphics Processing Unit: GeForce 256. Retrieved October 24, 2017 from http://www.nvidia.com/object/IO_20020111_5424.html

[3] Anon. 2017. Performance Analysis Tools. (April 2017). Retrieved October 24, 2017 from https://developer.nvidia.com/performance-analysis-tools

[4] K. Gupta, J. A. Stuart and J. D. Owens, "A study of Persistent Threads style GPU programming for GPGPU workloads," 2012 Innovative Parallel Computing (InPar), San Jose, CA, 2012, pp. 1-14.

[5] Martinez, G., Gardner, M., & Feng, W. C. (2011, December). CU2CL: A CUDA-to-OpenCL translator for multi-and many-core architectures. In Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on (pp. 300-307). IEEE.

[6] Anon. Clang 6 documentation. Retrieved December 6, 2017 from https://clang.llvm.org/docs/ClangTools.html

[7] Anon. Clang 6 documentation. Retrieved December 6, 2017 from https://clang.llvm.org/docs/RAVFrontendAction.html

[8] Anon. Clang 6 documentation. Retrieved December 6, 2017 from http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[9] Anon. clang: a C language family frontend for LLVM. Retrieved December 10, 2017 from https://clang.llvm.org/

[10] Anon. CUDA C Programming Guide. Retrieved December 10, 2017 from http://docs.nvidia.com/cuda/cuda-c-programming-guide

[11] LLVM, \The LLVM Compiler Infrastructure", online: http://llvm.org/,
last access: June 2012.