Scientific Research Publishing

Journal of Software Engineering and Applications, 2024, 17, 89-108
https://www.scirp.org/journal/jsea
ISSN Online: 1945-3124
ISSN Print: 1945-3116

Developing a Clang Libtooling-Based Refactoring Tool for CUDA GPU Programming

Kian Nejadfard[1], Janche Sang[2]

[1]Champ Titles, Inc., Cleveland, USA
[2]Department of Computer Science, Cleveland State University, Cleveland, USA
Email: j.sang@csuohio.edu

How to cite this paper: Nejadfard, K. and Sang, J. (2024) Developing a Clang Libtooling-Based Refactoring Tool for CUDA GPU Programming. *Journal of Software Engineering and Applications*, **17**, 89-108.
https://doi.org/10.4236/jsea.2024.172005

Received: January 11, 2024
Accepted: February 26, 2024
Published: February 29, 2024

Abstract

Refactoring tools, whether fully automated or semi-automated, are essential components of the software development life cycle. As software libraries and frameworks evolve over time, it's crucial for programs utilizing them to also evolve to remain compatible with modern advancements. Take, for example, NVIDIA CUDA's platform for general-purpose GPU programming. Embracing the more contemporary unified memory architecture offers several benefits, such as simplifying program source code, reducing bugs stemming from manual memory management between host and device memory, and optimizing memory transfer through automated memory handling. This paper describes our development of a refactoring tool based on Clang's Libtooling to facilitate this transition automatically, thereby relieving developers from the burden and risks associated with manually refactoring large code bases.

Keywords

Refactoring, CUDA, Unified Memory, Clang, Libtooling

1. Introduction

In the code base of a software product, whether it is a new development from the ground up or an existing product that has endured for decades, change is always happening. The more active a project, the more changes are applied to the code base. Furthermore, changes to a code base may occur for various purposes, such as adding a new feature, fixing a bug, and refactoring existing code. Refactoring can generally be described as making source code changes while preserving the external behavior [1]. In other words, refactoring should not change the program's conformance to functional requirements [2].

One important reason to refactor source code is to keep up with newer ver-

sions of frameworks, libraries, and language standards over time. Aside from the obvious choice of refactoring source code by hand, automated refactoring is a good alternative that can help save developers' time and therefore reduce costs associated with maintaining a software project. In addition, using an automated refactoring tool can reduce the chance of introducing bugs when compared against manual refactoring. Of course, this is likely only true if the refactoring tool has been tested rigorously and is robust in its decision making. As an example, 2 to 3 is a refactoring tool designed to translate Python2 to Python3 [3], showcasing the efficiency and reliability that automated tools can bring to the process.

Thinking about implementation details, there are two general approaches to performing automated refactoring: the search-and-replace method and the context-aware method. In the search-and-replace method, the refactoring tool searches source files for occurrences of specific string values and performs modifications as needed. This method has a potentially big flaw: due to lack of awareness of the programming language and its semantics, the purely text-based modifications can have a significant number of false-positives and false-negatives, either leading to a failed refactoring operation, or worse, a seemingly successful refactoring operation that has introduced hidden bugs.

In the context-aware method, the refactoring tool is able to understand the semantics of the language, and build an understanding of the program source code prior to deciding on what can be refactored. While this approach is superior to the previous method, it is much more complex to employ, especially for a language as complex as C++.

In recent years, GPUs have been used for improving the performance of various computational intensive applications in the areas of mathematics [4] [5], artificial intelligence [6], simulation [7], etc. *Compute Unified Device Architecture* (CUDA) is NVIDIA's software platform for writing programs that can harness the parallel computation power of the GPU [8]. It is important to note that only a part of such programs are executed on the GPU, while the rest of the program is by the CPU and main memory, as is usual. Because of this hybrid model of execution, it is necessary to have a data transfer pipeline between main memory and the memory on the graphics card, as shown in **Figure 1**. We will refer to the memory on the graphics card as *device memory* from now on.

With the introduction of CUDA version 6, NVIDIA introduced the *Unified Memory* model. Prior to unified memory, programmers had to distinguish between data from main memory versus data from device memory and specifically implement the means to transfer such data between the two locations whenever necessary [9]. Having to perform this task manually increases the complexity of program implementation, as well as introducing a higher chance of creating bugs. The use of unified memory allows programmers to get free of having to distinguish between memory from these two locations, and treat both the same way. Essentially, unified memory abstracts main and device memory away from the programmer and lets the system handle this automatically.

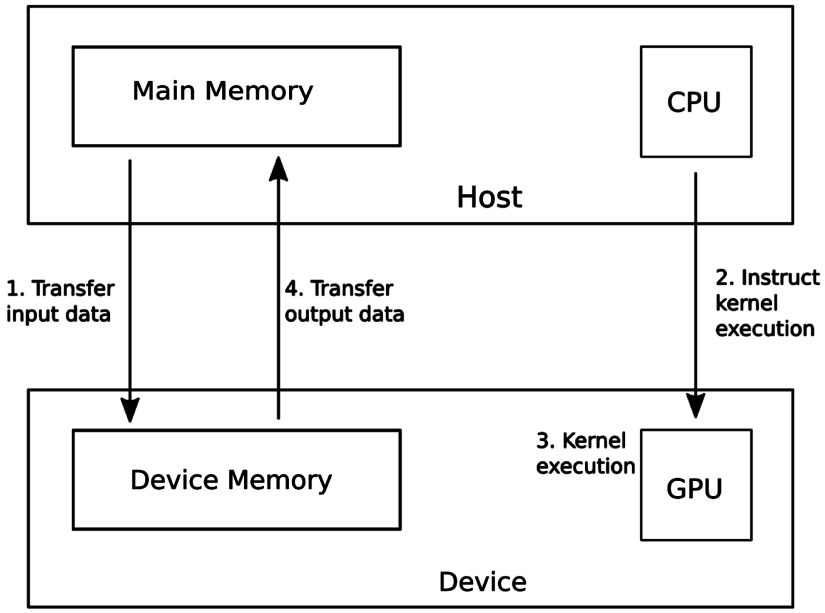

Figure 1. Typical data and processing flow in a CUDA GPU program.

Given the inherent benefits of refactoring CUDA programs to use the unified memory management API, this paper describes our development of an automatic refactoring tool that is able to update CUDA source code that is using manual memory management between host and device to utilize the more modern unified memory API. This has the potential to save valuable time and money that is well spent in other areas of projects.

The refactoring tool we developed has the ability to use the Abstract Syntax Tree (AST) representation of the program source code to perform *context-aware* refactoring on it, and write the updated code to the output source file. This is in contrast to a simple *find-replace* method that is easier to implement, however *is not smart enough* to perform advanced refactoring that requires inferring information from the context and acting accordingly.

Once an abbreviation for *Low-Level Virtual Machine*, LLVM is a collection of modular compiler and toolchain technologies [10]. The LLVM infrastructure has various *front-end* compilers for languages such as C, C++, Swift, Rust, D, and CUDA. The front-end compilers translate source code from each of these languages to a lower-level, intermediate representation called *LLVM IR*. Optimizers are then run on this intermediate representation and in the end, using one of the back-ends of LLVM, the optimized intermediate representation is translated to machine code for a specific target instruction set architecture. The LLVM compiler infrastructure and its C/C++/Obj-C/CUDA front-end compiler named Clang, have a modular and library-based design, which allows developers to develop tools that can reuse critical parts of this ecosystem, such as the pre-processor, lexer, parser, and so on, without having to re-invent the wheel. This is especially important for languages such as C++ that have a complicated syntax tree.

For the purpose of a refactoring tool that performs source-to-source translation only a portion of the compiler toolchain pipeline needs to be used, since we are not interested in generating the intermediate representation. Rather, the focus is put on the original source code being refactored and also the AST that the front-end compiler produces from the source code. *LibClang* is a stable high-level C interface to Clang that provides powerful abstractions for iterating through an AST using a cursor without getting into too much details of Clang's AST. *LibTooling*, on the other hand, is a C++ interface to Clang that allows full control over the AST and is aimed at writing standalone tools that are run on a single file or a specific set of files outside the build system. The downside of libTooling is that it's not guaranteed to be stable, therefore it is subject to change with future releases of Clang [11].

The organization of this paper is as follows. Section 2 describes the related work in source-to-source translation. In Section 3, a CUDA traditional program is used to illustrate some key points with regard to refactoring it for using unified memory. Section 4 describes the transformation framework based on our experience of manual translation. The detailed design and implementation issues of our automatic refactoring tool are discussed in Section 5. Finally, we give a short conclusion and future work in Section 6.

2. Related Work

To date, the only refactoring tool specific to CUDA that was found is the work of Damevski and Muralimanohar [12] for transforming a pure C loop into a parallel CUDA kernel function to be executed on the GPU. Such transformation can be categorized as a source-to-source translation from C to CUDA. While this is very valuable, it is not able to fulfill the need for a refactoring tool specifically for CUDA, where both source and target of the translation are CUDA.

Fortunately, it is a very good time for working on a refactoring tool for programs written in CUDA, due to the work of Wu *et al.* on GPUCC, an open-source General-Purpose GPU (GPGPU) compiler [13]. Since the publishing of their paper in 2016, GPUCC has been merged into and become a part of the LLVM infrastructure's Clang front-end compiler and therefore, all of the libraries and tooling available in this infrastructure such as LibTooling can now be used for CUDA source code as well.

Given the complexity of the C++ language, it would be a tremendous effort to develop a lexer and parser for it from scratch. Moreover, it would be a waste of efforts if every refactoring/source analysis tool for the C++ language has to implement its own limited parser to be able to work on source code. Luckily, with the inception of the *LLVM* project [10], the tooling around the C++ language has been significantly improved. *LibTooling* is a library to support development of standalone tools based on *Clang*, which is a language front-end and tooling infrastructure for the C family of languages (C, C++, Objective C/C++, OpenCL, CUDA, and RenderScript) for the LLVM project [14].

The use of this library is very useful for writing tools that are able to use Clang's C++/CUDA parser to obtain an abstract syntax tree representation, process the tree and run queries against it to infer information about the structure of the program, and run *front-end actions* on it according to the context. This allows for performing source-to-source transformation (refactoring) on existing source code.

OP2 is a domain-specific language embedded in C/C++ that aims for decoupling of scientific specification of an application from its parallel implementation by transforming application code written using the OP2 DSL into the desired form that can be executed on various backend hardware platforms such as OpenMP, CUDA, and OpenCL [15].

Balogh *et al.* [16] used Clang's LibTooling to implement a source-to-source translator called *OP2-Clang*. The tool was designed to significantly reduce maintenance and to make it easy to be extended to generate new parallelization strategies and optimizations for different hardware platforms.

3. Unified Memory Refactoring in CUDA

In this section, a few snippets of a CUDA program are reviewed in order to emphasize a few key points with regards to refactoring CUDA programs for using the unified memory API. This program showcases an important algorithm in parallel computing, called reduction, which can be described as extracting a single value from an input sequence of data by applying a binary operation [17]. In other words, the input sequence is divided into smaller sequences, and the binary operation is applied in parallel to the smaller sequences, and in the end the result of this operation is collected back as a single value.

Without further ado, let's take a look at a portion of the source code (**Listing 3.1**), courtesy of NVIDIA [18].

On lines 3 and 14 host memory is allocated using malloc and assigned to h_idata and h_odata. Device memory is allocated using cudaMalloc on lines 19 and 20, and assigned to d_idata and d_odata. After input data has been initialized on host (contents of h_idata), they are copied over to the corresponding device memory counterpart. In the end, the allocated host and device memory is released using calls to free and cudaFree. On line 36 there is a call to benchmarkReduce. The snippet in **Listing 3.2** shows the definition of this function.

Looking at the signature of function benchmarkReduce, it can be seen that three memory pointers are being passed in as arguments: h_odata, d_idata, and d_odata. On line 10, a block of memory is allocated on device memory using pointer d_intermediateSums, which is later used in a device-to-device memory copy operation. The unified memory API allows for allocating memory once, and using the same pointer in either host or device code. Transferring data between host and device is left to the GPU driver and is handled automatically, although there are ways to provide hints to the driver in order to potentially improve its throughput. In order to use unified memory, it is important to first

```
// create random input data on CPU
unsigned int bytes = size * sizeof(T);
T *h_idata = (T *)malloc(bytes);
for (int i = 0; i < size; i++) {
    ...  // initialize h_idata
}
int numBlocks = 0, numThreads = 0;
getNumBlocksAndThreads(whichKernel, size, maxBlocks,
        maxThreads, numBlocks, numThreads);
if (numBlocks == 1) {
  cpuFinalThreshold = 1;
}
// allocate mem for the result on host side
T *h_odata = (T *)malloc(numBlocks * sizeof(T));
// allocate device memory and data
T *d_idata = NULL;
T *d_odata = NULL;

checkCudaErrors(cudaMalloc((void **)&d_idata, bytes));
checkCudaErrors(cudaMalloc((void **)&d_odata,
        numBlocks * sizeof(T)));
// copy data directly to device memory
checkCudaErrors(cudaMemcpy(d_idata, h_idata, bytes,
        cudaMemcpyHostToDevice));
checkCudaErrors(cudaMemcpy(d_odata, h_idata, numBlocks * sizeof(T),
        cudaMemcpyHostToDevice));
// warm-up
reduce<T>(size, numThreads, numBlocks, whichKernel,
          d_idata, d_odata);

int testIterations = 100;
StopWatchInterface *timer = 0;
sdkCreateTimer(&timer);

T gpu_result = 0;
gpu_result = benchmarkReduce<T>(size, numThreads, numBlocks,
        maxThreads, maxBlocks, whichKernel, testIterations,
        cpuFinalReduction, cpuFinalThreshold, timer, h_odata,
        d_idata, d_odata);

double reduceTime = sdkGetAverageTimerValue(&timer) * 1e-3;

// compute reference solution
T cpu_result = reduceCPU<T>(h_idata, size);

... // print results and compare the gpu result with the cpu result

// cleanup
sdkDeleteTimer(&timer);
free(h_idata);
free(h_odata);
checkCudaErrors(cudaFree(d_idata));
checkCudaErrors(cudaFree(d_odata));

... // return true or false
```

Listing 3.1. Part of the reduction example, runTest function.

identify memory pointers that are eligible for this migration. Candidates can be picked up by looking at calls to cudaMalloc. In the case of reduction program, candidates are:

- d_intermediateSums
- d_idata in runTest
- d_odata in runTest
- d_idata in benchmarkReduce
- d_odata in benchmarkReduce

It is also very important to point out that, there are two different sets of d_idata and d_odata, defined in functions benchmarkReduce and runTest. This emphasizes an advantage of using a context-aware refactoring tool which understands the semantics of the source code (C++/CUDA) and is able to differentiate between two variable references, both named d_idata or d_odata. A context-aware tool is able to track the definition of these variables, and perform

```
template <class T>
T benchmarkReduce(int n, int numThreads, int numBlocks,
     int maxThreads, int maxBlocks, int whichKernel,
     int testIterations, bool cpuFinalReduction,
     int cpuFinalThreshold, StopWatchInterface *timer,
     T *h_odata, T *d_idata, T *d_odata) {
 T gpu_result = 0;
 bool needReadBack = true;
 T *d_intermediateSums;
 checkCudaErrors(cudaMalloc((void **)&d_intermediateSums,
           sizeof(T) * numBlocks));

 for (int i = 0; i < testIterations; ++i) {
  gpu_result = 0;
  cudaDeviceSynchronize();
  sdkStartTimer(&timer);
  // execute the kernel
  reduce<T>(n,numThreads,numBlocks,whichKernel,d_idata,d_odata);
  // check if kernel execution generated an error
  getLastCudaError("Kernel execution failed");

  if (cpuFinalReduction) {
    // copy result from device to host
    checkCudaErrors(cudaMemcpy(h_odata, d_odata,
          numBlocks * sizeof(T), cudaMemcpyDeviceToHost));
    ... // sum partial sums from each block on CPU

    }
    needReadBack = false;
  } else {
    int s = numBlocks;
    int kernel = whichKernel;

    while (s > cpuFinalThreshold) {
      int threads = 0, blocks = 0;
      getNumBlocksAndThreads(kernel, s, maxBlocks, maxThreads,
               blocks, threads);
      checkCudaErrors(cudaMemcpy(d_intermediateSums, d_odata,
               s * sizeof(T), cudaMemcpyDeviceToDevice));
      reduce<T>(s,threads,blocks,kernel,d_intermediateSums,d_odata);

      ... // calculate s based on the kernel value
    }
    if (s > 1) {
      // copy result from device to host
      checkCudaErrors(cudaMemcpy(h_odata, d_odata,
               s * sizeof(T), cudaMemcpyDeviceToHost));
      ... // sum partial block sums on GPU
      needReadBack = false;
    }
  }
  cudaDeviceSynchronize();
  sdkStopTimer(&timer);
 }
 if (needReadBack) {
  // copy final sum from device to host
  checkCudaErrors(cudaMemcpy(&gpu_result, d_odata, sizeof(T),
           cudaMemcpyDeviceToHost));
 }
 checkCudaErrors(cudaFree(d_intermediateSums));
 return gpu_result;
}
```

Listing 3.2. Part of the reduction example, benchmarkReduce function.

refactoring operations on them separately. Using a string-based search/replace method in this scenario will likely result in unpredictable results.

After identifying device pointer candidates, it is time to find out how data is copied from/to these pointers. For this purpose, calls to cudaMemcpy must be found that either their source or target pointer (the first two arguments) are one of the device pointers that have been marked down. In the example scenario above, these calls are:

- Line 23 of runTest snippet—copy from h_idata to d_idata
- Line 25 of runTest snippet—copy from h_idata to d_odata
- Line 23 of benchmarkReduce snippet—copy from d_odata to h_odata

- Line 38 of benchmarkReduce snippet—copy from d_odata to d_intermediateSums
- Line 45 of benchmarkReduce snippet—copy from d_odata to h_odata
- Line 56 of benchmarkReduce snippet—copy from d_odata to gpu_result which is a variable defined on the stack of the host

Out of all the copy operations identified above, the copy from d_intermediateSums to d_odata is a device-to-device copy, as both pointers are allocated on device. Therefore, logically speaking, when migrating to unified memory, this copy call should not be touched as this data transfer is not between host and device.

The cudaMemcpy calls above also reveal another important information: the match between a host memory pointers and a corresponding device memory pointers. This information is important in order to find out how the memory pointers can be replaced with unified memory pointers. Then, calls to malloc must be identified that allocate host memory for pointers that have been identified in relation to device memory pointers of interest:

- Line 3 of **Listing 3.1** runTest snippet—allocating memory for h_idata

At this point, the most important data for refactoring has been collected. Either the host memory h_idata or its corresponding device memory d_idata can be converted to use cudaMallocManaged for its allocation, while replacing its counterpart references with the newly added/refactored pointer.

4. Transformation Framework

After performing transformations by hand on a number of test programs, as it has been shown in the previous section of this section, it appears that we may be able to formulate a common framework for making such transformations. The following describes the steps of this framework in a mostly procedural method:

1) Identify device-specific memory allocations by locating calls to cudaMalloc. This reveals the device pointer allocations that may be candidates for refactoring into unified memory.

2) Identify calls to cudaMemcpy that are copying data across host and device, in either direction (third argument must not be cudaMemcpyDeviceToDevice). These calls must have the device memory pointers identified in the previous step as either their source or destination pointers. When picking up such cudaMemcpy calls, store the host and device pointers in such a way that they can be associated/looked up.

3) Remove calls to cudaMalloc that allocate device memory for a pointer that has been identified in the previous step. If there is a call to cudaMalloc that allocates memory for a device pointer that has a device-to-device copy, or no copies at all, then it must be left asis. We are only interested in removing cudaMalloc calls that are being replaced with unified memory management.

4) Remove cudaMemcpy calls that were identified in the previous step. They are no longer needed, as we are replacing their corresponding device memory

pointers with unified memory. If a cudaMemcpy that is being removed, is copying data from device to host, then we should replace it with a call to cudaDeviceSynchronize. Without doing so, there will likely be undefined behavior in the refactored code (unless a cudaDeviceSynchronize) already exists.

5) If possible, remove the error-checking code for each cudaMemcpy invocation that gets removed. This will either be in the form of an if-statement with LHS or RHS being the returned value of the cudaMemcpy invocation, or a function call taking the returned value of cudaMemcpy as one of its arguments.

6) Identify calls to malloc that allocate memory for host pointers that have been identified in step 2 and refactor them each to an initialization of the pointer and then a call to cudaMallocManaged. Similar to the following diff:

```
< float * host_A = (float *) malloc(size);
---
> float * host_A = (float *) nullptr;
> cudaMallocManaged(&host_A, size);
```

7) Rename references to device memory pointers with what is being used as the first argument to cudaMallocManaged from step 6. The association between host and device pointers that has been identified in step 2 is useful here.

8) Add a call to cudaDeviceSynchronize after executing the kernel. This was not an issue previously because of an explicit call to cudaMemcpy to transfer the results back to host memory. However, since that is now removed, invoking cudaDeviceSynchronize is necessary to ensure code execution on host machine does not continue further until the kernel execution is complete.

9) Remove calls to free if their first argument is one of the host memory pointers that we have identified in step 2.

10) Refactor calls to cudaFree with the first argument being one of the device pointers from step 2 such that the first argument is replaced with the corresponding host pointer name.

We manually converted a few example CUDA programs and the data for manual conversion is shown in Table 1. Time measurements in the table include getting familiar with the program's context, it's general flow, and performing bare-minimum changes that are needed to convert the program to use the unified memory API. While recompiling and testing each program for correctness has been performed, the time it took for these two operations are not included in the values above.

5. Design and Implementation

The refactoring tool takes one or more C++/CUDA source files as input. Each file given to the refactoring tool is considered for analysis as a separate compilation unit. Therefore, all context information that is gathered by the tool and all the refactoring decisions made are local to each translation unit. Figure 2 shows a high level view of the interaction between program source code, the front-end parser, the generated AST along with the replacement actions, and the output refactored source code.

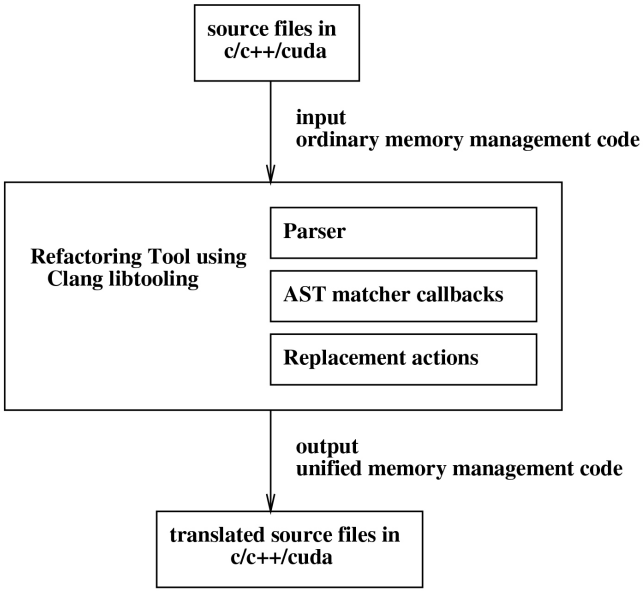

Figure 2. High level operation of the refactoring tool.

Table 1. Statistics and manual refactoring time measurement for a selected number of CUDA programs.

Program Name	Original (bytes)	Refactored (bytes)	Refactored Lines	Time to Refactor
vectorAdd	3395	2452	11	9 m 53 s
matrixMul	13,736	12,535	13	8 m 43 s
mergeSort	4065	3548	17	12 m 34 s
sortingNetworks	5356	4630	12	9 m 8 s
transpose	19,877	19,674	8	7 m 22 s
scalarProd	5364	5172	7	6 m 5 s
reduction	18,204	18,110	18	31 m 36 s

It is beneficial to get acquainted with the Clang AST nodes most relevant to this work, in order to better understand how different variations of CUDA function calls of interest may be represented in the AST for the purpose of refactoring. Without further ado:

- **DeclRefExpr**—Expression that references a variable declaration.
- **CallExpr**—Expression that is a function call.
- **VarDecl**—Variable declaration statement.
- **BinaryOperator**—Binary operator, such as = that takes a left-hand side and right-hand side.
- **UnaryOperator**—Unary operator, such as & used to retrieve the address of a variable.
- **FunctionDecl**—Function declaration. This is what a CallExpr could reference.

- **ParenExpr**—Parenthesis expression.
- **ImplicitCastExpr**—Represents an implicit type conversion that does not have any representation in the source code. For example, when calling a function using its name, there is an implicit cast to a pointer to the function (also referred to as function-to-pointer decay) under the hood.
- **CStyleCastExpr**—Represents a C-style cast in C++ code.

In Clang there are two methods of working with the AST and its nodes: *visiting*, and *matching* [19]. Visiting nodes can be achieved by inheriting from the RecursiveASTVisitor base class using the *curiously recurring template pattern* (*F-bound polymorphism*) and overriding the desired Visit*() methods. This approach visits AST nodes using a depth-first pre-order (default) or post-order traversal algorithm [20] and whenever the type of a node being visited matches one of the Visit*() methods, that method is called.

As an alternative to the visiting approach, Clang's LibASTMatchers provides a domain-specific language written in C++ that allows defining predicates and corresponding callback handlers on AST nodes. Section takes a closer look at using LibASTMatchers and the domain-specific language for working with AST nodes combined with LibTooling.

In comparison, LibASTMatchers and its domain-specific language enables writing more expressive code that is easier to read and understand, compared to an equivalent implementation using RecursiveASTVisitor. Given the capability of AST matchers to match and handle nodes of interest for this refactoring tool, along with the added benefit of producing more readable code, they have been preferred over an implementation using RecursiveASTVisitor.

5.1. AST Matchers

The refactoring tool is developed by using Clang's LibTooling, and its extensive collection of AST node matchers. These matchers allow the refactoring tool to get a handle to AST nodes of interest, and observe the details of each node, in order to infer and gather information to perform context-aware refactoring decisions. Refactoring actions are then applied to each relevant source file.

When registering each AST matcher, LibTooling allows specifying bind points to details of each node. Moreover, various different matchers can be linked together to match number of relevant nodes of the AST, while capturing the important parts of it using the bind points. For example, when matching a call to cudaMemcpy, bind points are set for its arguments, as well as the binary operator or variable declaration statement that is above this call in the AST.

The refactoring tool registers the following two AST matchers for calls to cudaMalloc (**Listing 5.1**) and cudaMemcpy (**Listing 5.2**).

A number of variations of AST when calling cudaMalloc are discussed below.

1) Ignoring the returned value—When ignoring the returned value of cudaMalloc, the AST section for this call is comprised of just the call expression. **Figure 3** shows a portion of the AST for this call.

```
cudaMalloc((void **) &ptr, size);
```

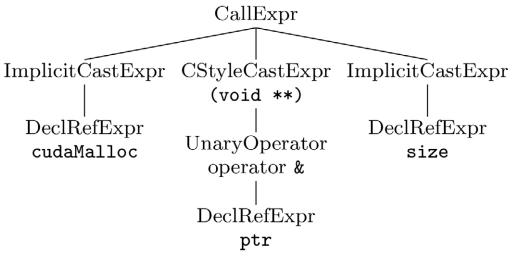

Figure 3. CudaMalloc call with return value ignored.

```
callExpr(isExpansionInMainFile(),
    callee(functionDecl(hasName("cudaMalloc"))),
    optionally(
        anyOf(
            hasParent(callExpr().bind("parent_call")),
            hasParent(parenExpr(hasParent(callExpr().bind("
    parent_macrocall")))),
            hasAncestor(varDecl().bind("lhs_vardecl")),
            hasAncestor(binaryOperator(hasOperatorName("="),
    hasLHS(declRefExpr().bind("lhs_declref"))))
        )
    ),
    hasArgument(0, cStyleCastExpr(hasDescendant(declRefExpr().bind
    ("ptr")))),
    hasArgument(1, declRefExpr().bind("size"))
).bind("cudaMalloc_call");
```

Listing 5.1. AST matcher for calls to cudaMalloc function.

```
callExpr(isExpansionInMainFile(),
    callee(functionDecl(hasName("cudaMemcpy"))),
    optionally(
        anyOf(
            hasParent(callExpr().bind("parent_call")),
            hasParent(parenExpr(hasParent(callExpr().bind("
    parent_macrocall")))),
            hasAncestor(varDecl().bind("lhs_vardecl")),
            hasAncestor(binaryOperator(hasOperatorName("="),
      hasLHS(declRefExpr().bind("lhs_declref"))))
        )
    ),
    hasArgument(0, declRefExpr().bind("dest_ptr")),
    hasArgument(1, declRefExpr().bind("src_ptr")),
    hasArgument(2, declRefExpr().bind("size")),
    hasArgument(3, declRefExpr().bind("kind"))
).bind("cudaMemcpy_call");
```

Listing 5.2. AST matcher for calls to cudaMemcpy function.

2) Using variable declaration to capture the returned value—In this case, the returned value of cudaMalloc is assigned to a variable that is declared in-place. Figure 4 shows a portion of the AST for this call.

3) Using variable reference to capture the returned value—In this case, the returned value of cudaMalloc is assigned to a variable that has previously been declared. Figure 5 shows a portion of the AST for this call.

4) Using an error-checking function—In this case the returned value of cudaMalloc is directly passed to a function call that checks the value and handles it appropriately. Figure 6 shows a portion of the AST for this call.

The matchers start with a base node to match, callExpr in the above two cases, as well as any number of narrowing matchers such as hasName which help narrow

```
cudaError_t err = cudaMalloc((void **) &ptr, size);
```

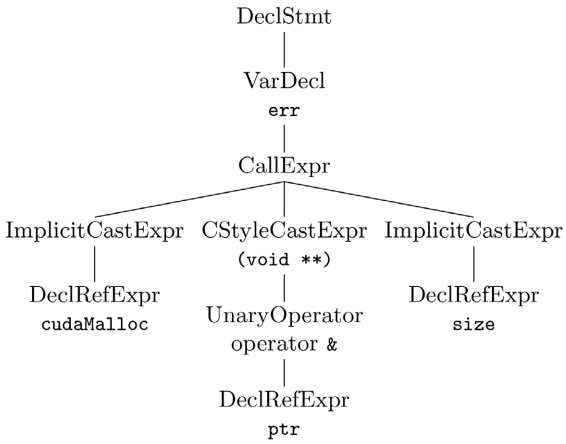

Figure 4. CudaMalloc call with return value assigned to a variable declared in-place.

```
err = cudaMalloc((void **) &ptr, size);
```

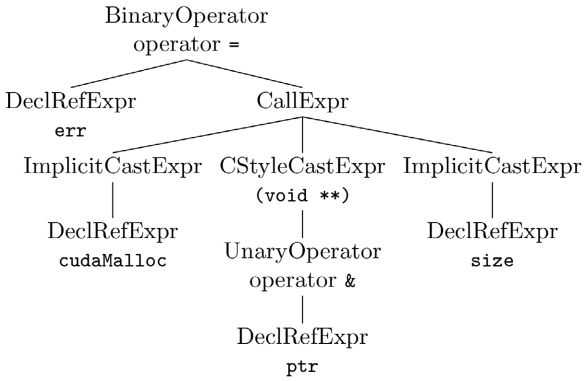

Figure 5. CudaMalloc call with return value assigned to an existing variable.

```
checkError(cudaMalloc((void **) &ptr, size));
```

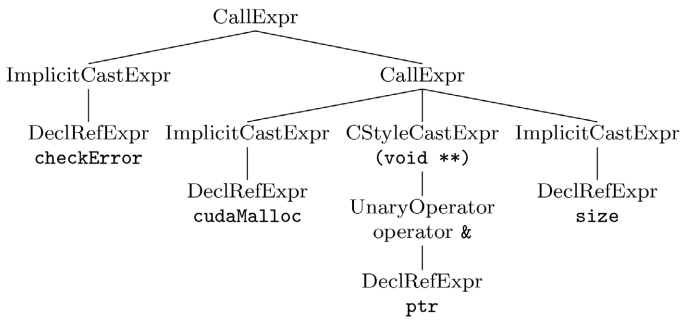

Figure 6. CudaMalloc call with error-checking function.

the selection down to more specific matches. Several other nodes in relation to the selected node can also be matched. In the case of cudaMalloc and cudaMemcpy, a potential parent function call for error checking or a variable decla-

ration or reference expression to the left-hand side of the function call can be matched as well. Using bind points, at the time of performing refactoring actions, the tool is able to get a handle to these specific points of the AST in order to create replacement objects.

The AST matcher for calls to cudaFree is much simpler, as there are less relative nodes that need to be matched (Listing 5.3).

And last but not least, the AST matcher for calls to malloc (Listing 5.4).

Inside each callback function (the counterpart to registered matchers), AST nodes are retrieved using the binding references, and cast to the correct pointer type. It is very important to use correct binding names and node types in the callback functions (Listing 5.5).

The AST matcher callback functions add the collected node pointers to an injected instance of RefactorContext object, which is responsible for maintaining the collected data for the refactoring tool for analysis. Once all the information has been collected by the tool, execution stage begins. The following pseudo-code, based on the transformation framework that has been identified in section 4, describes the high-level operations of the execution stage (Listing 5.6).

The time complexity of the operations done in the execution phase is $O(n \times m)$ where n is the number of cudaMalloc calls and m is the number of cudaMemcpy calls. While this time complexity is not in the overall efficient category, we believe it not to be critical source of concern for a refactoring tool. Especially considering the fact that in each translation unit fed into the refactoring tool, the total number of such calls is not too high.

```
callExpr(isExpansionInMainFile(),
    callee(functionDecl(hasName("cudaFree"))),
    hasArgument(0, declRefExpr().bind("ptr")))
.bind("cudaFree_call");
```

Listing 5.3. AST matcher for calls to cudaFree.

```
callExpr(isExpansionInMainFile(),
    callee(functionDecl(hasName("malloc"))),
    optionally(
        anyOf(
            hasAncestor(varDecl().bind("lhs_vardecl")),
            hasAncestor(binaryOperator(hasOperatorName("="),
    hasLHS(declRefExpr().bind("lhs_declref"))))
        )
    ),
    hasArgument(0, declRefExpr().bind("size"))
).bind("malloc_call");
```

Listing 5.4. AST matcher for calls to malloc.

```
const clang::CallExpr *call = match_result.Nodes.getNodeAs<clang::
    CallExpr>("cudaMemcpy_call");
const clang::DeclRefExpr *dest_ptr = match_result.Nodes.getNodeAs<
    clang::DeclRefExpr>("dest_ptr");
const clang::DeclRefExpr *src_ptr = match_result.Nodes.getNodeAs<
    clang::DeclRefExpr>("src_ptr");
const clang::DeclRefExpr *size = match_result.Nodes.getNodeAs<
    clang::DeclRefExpr>("size");
const clang::DeclRefExpr *kind = match_result.Nodes.getNodeAs<
    clang::DeclRefExpr>("kind");
```

Listing 5.5. Part of the callback function for cudaMemcpy.

```
for each cudaMalloc call {
    for each cudaMemcpy call {
        if (cudaMemcpy call is device-to-device) {
            skip to next iteration
        }

        capture source and destination pointers of cudaMemcpy
        if (source or destination are not present in the
    cudaMalloc call) {
            skip to next iteration
        }

        remove cudaMalloc call
        remove cudaMemcpy call
        add cudaDeviceSynchronize call if needed
        refactor malloc into cudaMallocManaged and remove free
    call
        refactor device memory pointer references
    }
}
```

Listing 5.6. Pseudo-code for refactoring actions.

5.2. Empirical Results

The refactoring tool was run on 161 CUDA sample programs provided in [18]. In total 297 source files were processed by the tool, out of which 207 ran through refactoring with normal exit code while 90 indicated an abnormal exit code. The following root causes have been found for the identified failures:

- For 30 files out of 90 that indicated an abnormal exit code when running the refactoring tool on them, the root cause is due to Clang not being able to locate a header file that is included in the source file being processed. Given that the refactoring tool needs to run the pre-processor on source files, this is a fatal issue and therefore, the refactoring operation fails. While compiler flags and arguments can be passed to the refactoring tool for such cases, the testing script being used for this test run is not capable of accommodating this customization per test program.
- For 45 files out of 90 (there is a slight overlap of less than 5 with the 30 files mentioned in previous item), there was at least one error generated due to calling an undeclared function. Reasons for this category of failure include:

- Inappropriate compiler flags passed to the refactoring tool (missing specific include folders).
- A missing functionality [21] in Clang with CUDA texture functions [22] such as tex2D and tex3D. The course of action for resolving this issue is to re-compile the refactoring tool using newer versions of LLVM and Clang libraries whenever the fix is available.
- "*No matching function for call to 'min'*." There is a difference in the implementations of math.h and cmath headers between NVIDIA's nvcc compiler and Clang [23]. During tests, cases have been found where nvcc can compile a certain code that uses min and max functions where Clang generates the previously mentioned error. The fix is rather simple: with Clang, min and max should be changed to std::min and std::max. This works when using either of math.h or cmath.

- For 6 files out of 90, there was an error during refactoring operations: "*failure in removing/replacing call to cudaMemcpy*." The tool prints this error mes-

sage when it identifies a scenario where there are overlapping/conflicting replacements for a specific range. All of the cases of this specific error that were observed during this test occur due to mistakenly adding a variable replacement for a cudaMemcpy function. The issue has been fixed during the development process and the test suite has been updated to include a test covering this issue.

- In 2 cases, the following error was observed: "*error: host pointer's declaration is a function argument. This feature is not implemented yet.*" This is an edge case that has been identified during development, which the refactoring tool identifies and reports as an error. Implementing this feature requires adding new AST matchers to find calls to the function that holds the declaration of the identified variable.
- In 2 cases, features of C++11 were used in the original source code. This can be easily resolved by adding the compiler flag-std=c++11 to the refactoring tool to ensure the appropriate C++ language standard level is used when pre-processing and parsing the source file.
- In at least 1 case, refactoring is unsuccessful due to a failed assertion check for malloc calls with a left-hand side declaration reference that is a struct member, rather than a regular variable:

```
result.values = (float *) malloc(result.mat_size_f);
```

The AST for such call is presented in Figure 7.

After looking into a randomly selected number of test programs that had normal exit code when being refactored, the following issues have been identified:

- The refactoring tool fails to match calls to cudaMemcpy if the pointer expression is not a plain *declaration reference expression*. Figure 8 shows the generated AST for a case where the pointer argument of cudaMemcpy is an element of a std::vector obtained by calling its operator [] member function. As an example:

```
std::vector<unsigned int> results(init_size);
cudaMemcpy(&results[0], d_results, size,
           cudaMemcpyDeviceToHost);
```

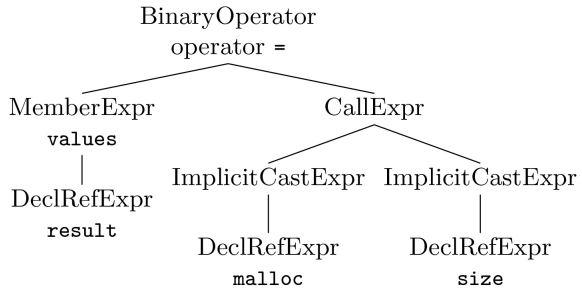

Figure 7. Malloc call with member variable reference on left-hand side.

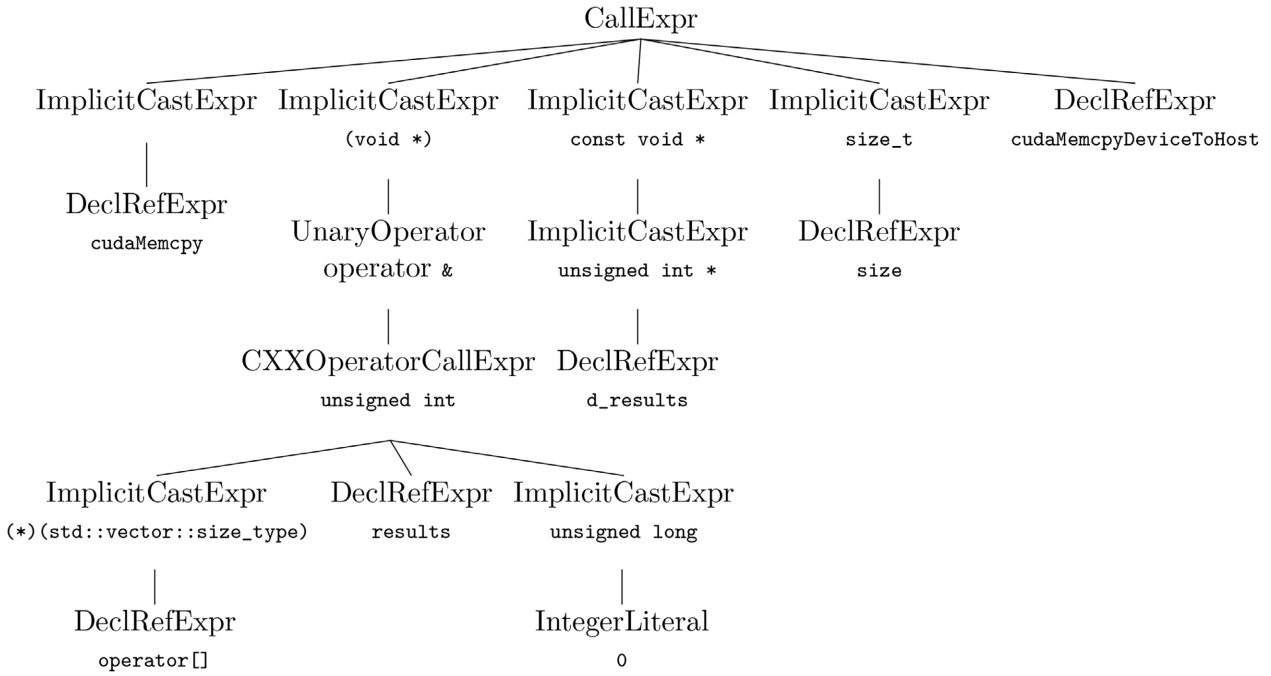

Figure 8. CudaMemcpy call with pointer argument that points to a std::vector element.

- In a few test programs, a host memory space gets allocated and initialized to a default value. Then, this same host memory gets copied to two separate device memory spaces that have been allocated via cudaMalloc. This pattern creates an ambiguity for the refactoring tool when associating the host and device memory pointers together. As a result, the refactoring tool tries to replace both of the device memory pointers with the same managed memory pointer, which is wrong. In order to modify this test program to resolve this ambiguity for the refactoring tool, the following modification was done to the source file manually, before running the refactoring tool:

```
checkCudaErrors(cudaMemcpy(d_idata, h_idata, bytes,
    cudaMemcpyHostToDevice));
checkCudaErrors(cudaMemcpy(d_odata, h_idata, maxNumBlocks *
    sizeof(T), cudaMemcpyHostToDevice));

// The above two lines were changed to:

checkCudaErrors(cudaMemcpy(d_idata, h_idata, bytes,
    cudaMemcpyHostToDevice));
cudaDeviceSynchronize();
checkCudaErrors(cudaMemcpy(d_odata, d_idata, maxNumBlocks *
    sizeof(T), cudaMemcpyDeviceToDevice));
```

- In at least 4 programs, compilation fails due to an improper order of declaration, allowing the existence of declaration references to variables that are not in scope yet. This is due to the way the refactoring tool has been designed to replace cudaMalloc and malloc with cudaMallocManaged. Currently, it removes cudaMalloc and converts malloc o cudaMallocManaged, assuming certain checks and balances are met. The following source code demonstrates a case where the current algorithm results in a bad variable reference:

```
cudaMalloc((void **) &d_data, size);
initValue<<<grid_size, block_size>>>(d_data, 0.0f);
// other code potentially using d_data
float *odata = (float *) malloc(size);
cudaMemcpy(odata, d_data, size, cudaMemcpyDeviceToHost));
```

To correctly refactor the above code, the refactoring tool must recognize that in this case, the device memory is declared earlier than host memory. Therefore, when replacing this device/host pair, it must declare the managed memory pointer at the line which the original device memory has been declared, in order to avoid any reference issues.

It is important to note that a normal exit code by the tool does not necessarily mean refactoring has been done correctly. To verify that, the full source code of the corresponding program must be re-compiled and tests must be executed to ensure the program's correctness is intact.

Results for 13 selected programs from the test set executed on a high-end workstation (Xeon Silver 4116 CPU, 3.0 GHz, 32 GB) with an NVIDIA Titan RTX GPU (4608 cores, 24 GB GDDR6 memory, CUDA runtime version 10.0) are presented in Table 2. The "*Original* (*bytes*)" and the "*Refactored* (*bytes*)" columns refer to the size of source file(s) before and after refactoring, respectively. If multiple files have had changes during refactoring, the sum of their size is displayed. *"Refactored lines"* is the number of changed lines during refactoring. The last column shows the conversion time to run the refactoring tool on each source program. It can be observed that using our tool, the conversion can be done less than one second, much less than the manual conversion which usually takes several minutes.

Table 2. Empirical results for selected CUDA programs from the test set.

Program Name	Original (bytes)	Refactored (bytes)	Refactored Lines	Conversion Time
BlackScholes	8382	7568	28	0.56 s
convolutionFFT2D	19,987	18,178	65	0.64 s
histogram	7091	6644	14	0.63 s
mergeSort	4065	3437	23	0.58 s
convolutionTexture	5830	5660	8	1.71 s
quasirandomGenerator	5404	5141	9	0.59 s
convolutionSeparable	5431	5052	13	2.56 s
transpose	19,877	19,599	14	0.96 s
reduction	18,204	17,644	18	0.66 s
scan	5982	5636	11	0.61 s
scalarProd	5364	4980	16	0.58 s
simpleHyperQ	7650	7514	6	0.56 s
sortingNetworks	5356	4553	23	6.79 s

6. Concluding Remarks and Future Work

The modern CUDA programming environment supports the unified memory and hence no longer needs to have explicit data movement between the host and the device as in the traditional CUDA model. Instead of manually translating the old CUDA code to the modern style, we have designed and implemented an automatic source-to-source conversion tool. Our implementation which uses the Clang LibTooling has been proved to be a successful approach. The tool is able to identify various scenarios and perform the desired action based on program's context. The tests did not show any false-positive or false-negative operations done by the tool. Furthermore, the empirical results show that the automatic conversion takes much less time than the manual transformation.

Future works include adding support for *prefetching* data from host to device to improve kernel execution performance in certain scenarios by adding a call to cudaMemPrefetchAsync. Moreover, additional cleanup of source code can be added to the tool to cover cases such as a cudaMemcpy call with error-checking using an if-statement. When the refactoring tool removes such calls, it leaves behind the if-statement and issues a warning to suggest manual cleanup.

As a safety check, the tool can be enhanced by running in-memory syntax check after creating refactoring changes and prior to writing these changes back to the original source file. This can help with identifying issues earlier in the process, creating diagnostic information regarding the syntax issues found, and exiting the tool with an abnormal exit code.

And last but not least, the use of clang::Rewriter in addition to clang::Replacements should be evaluated. Particularly, the Rewriter class has much better capability for inserting new lines, while the Replacements class is better suited for replacing a specific part of source code with another. The refactoring tool could benefit from both of these operations for a cleaner job.

Conflicts of Interest

The authors declare no conflicts of interest regarding the publication of this paper.

References

[1] Murphy-Hill, E. and Black, A.P. (2008) Refactoring Tools: Fitness for Purpose. *IEEE Software*, **25**, 38-44. https://doi.org/10.1109/MS.2008.123

[2] Kaur, A. and Kaur, M. (2016) Analysis of Code Refactoring Impact on Software Quality. *MATEC Web of Conferences*, **57**, 6. https://doi.org/10.1051/matecconf/20165702012

[3] (2021) 2to3—Automated Python 2 to 3 Code Translation. https://docs.python.org/3/library/2to3.html

[4] Hylton, A., Henselman-Petrusek, G., Sang, J. and Short, R. (2019) Tuning the Performance of a Computational Persistent Homology Package. *Software: Practice and Experience*, **49**, 885-905. https://doi.org/10.1002/spe.2678

[5] Hojnacki, M., Leeseberg, A., O'Shaughnessy, J., Dauchy, M., Hylton, A., Gold, L.

and Sang, J. (2020) Parallel Computation of Grobner Bases on a Graphics Processing Unit. *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, 417-432.

[6] Steinkraus, D., Buck, I. and Simard, P.Y. (2005) Using GPUs for Machine Learning Algorithms. *Eighth International Conference on Document Analysis and Recognition*, **2**, 1115-1120. https://doi.org/10.1109/ICDAR.2005.251

[7] Sang, J., Lee, C., Rego, V. and King, C. (2019) Experiences with Implementing Parallel Discrete-Event Simulation on GPU. *Journal of Supercomputing*, **75**, 4132-4149. https://doi.org/10.1007/s11227-018-2254-4

[8] Kirk, D.B. and Hwu, W.M.W. (2016) Programming Massively Parallel Processors: A Hands-on Approach. 3rd ed., Morgan Kaufmann Publishers Inc., Burlington.

[9] Harris, M. (2013) Unified Memory in CUDA 6. https://developer.nvidia.com/blog/unified-memory-in-cuda-6/

[10] (2020) The LLVM Compiler Infrastructure. http://llvm.org/

[11] Clang Tooling (2020) Choosing the Right Interface for Your Application. https://clang.llvm.org/docs/Tooling.html

[12] Damevski, K. and Muralimanohar, M. (2011) A Refactoring Tool to Extract GPU Kernels. *Proceedings of the 4th Workshop on Refactoring Tools*, May 2011, 29-32. https://doi.org/10.1145/1984732.1984739

[13] Wu, J., Belevich, A., Bendersky, E., Heffernan, M., Leary, C., Pienaar, J., Roune, B., Springer, R., Weng, X. and Hundt, R. (2016) GPUCC—An Open-Source GPGPU Compiler. *Proceedings of the* 2016 *International Symposium on Code Generation and Optimization*, February 2016, 105-116. https://doi.org/10.1145/2854038.2854041

[14] (2020) Clang: A C Language Family Frontend for LLVM. https://clang.llvm.org/

[15] Bertolli, C., Betts, A., Mudalige, G., Giles, M. and Kelly, P. (2011) Design and Performance of the OP2 Library for Unstructured Mesh Applications. In: Alexander, M., et al., Eds., *Euro-Par* 2011: *Parallel Processing Workshops*. Euro-Par 2011. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-29737-3_22

[16] Balogh, G.D., Mudalige, G.R., Reguly, I.Z., Antao, S.F. and Bertolli, C. (2018) OP2-Clang: A Source-to-Source Translator Using Clang/LLVM LibTooling. 2018 *IEEE/ACM 5th Workshop on the LLVM Compiler Infrastructure in HPC* (*LLVM-HPC*), Dallas, 12 November 2018, 59-70. https://doi.org/10.1109/LLVM-HPC.2018.8639205

[17] Barlas, G. (2015) Chapter 7—The Thrust Template Library. In: Barlas, G., Ed., *Multicore and GPU Programming*, Morgan Kaufmann, Boston, 527-573. https://doi.org/10.1016/B978-0-12-417137-4.00007-1

[18] (2020) CUDA Sample Code. https://github.com/NVIDIA/cuda-samples

[19] Bendersky, E. (2020) AST Matchers and Clang Refactoring Tools. https://eli.thegreenplace.net/2014/07/29/ast-matchers-and-clang-refactoring-tools

[20] (2021) Clang: RecursiveASTVisitor. https://clang.llvm.org/doxygen/classclang_1_1RecursiveASTVisitor.html

[21] (2020) Clang Bug: Add CUDA Builtin Surface/Texture Reference Support. https://reviews.llvm.org/D76365

[22] (2021) CUDA Toolkit Documentation: Texture Functions, https://docs.nvidia.com/cuda/cuda-c-programming-guide/#texture-functions

[23] (2021) Compiling CUDA with Clang: Standard Library Support. https://llvm.org/docs/CompileCudaWithLLVM.html#standard-library-support