Paper: Togpu: Automatic Source Transformation from C++ to CUDA using Clang/LLVM

Marangoni, Matthew; Wischgoll, Thomas; Wright State University; Dayton, Ohio

Abstract

Parallel processing using GPUs provides substantial increases in algorithm performance across many disciplines including image processing. Serial algorithms are commonly translated to parallel CUDA or OpenCL algorithms. To perform this translation a user must first overcome various GPU development entry barriers. These obstacles change depending on the user but in general may include learning to program using the chosen API, understanding the intricacies of parallel processing and optimization, and other issues such as the upkeep of two sets of code. Such barriers are experienced by experts and novices alike. Leveraging the unique source to source transformation tools provided by Clang/LLVM we have created a tool to generate CUDA from C++. Such transformations reduce obstacles experienced in developing GPU software and can increase efficiency and revision speed regardless of experience. Image processing algorithms specifically benefit greatly from a quick revision cycle which our tool facilitates. This manuscript details togpu, an open source, cross platform tool which performs C++ to CUDA source to source transformations. We present experimentation results using common image processing algorithms. The tool lowers entrance barriers while preserving a singular code base and readability. Providing core tools enhancing GPU developer facilitates further developments to improve high performance, parallel computing.

Introduction

Parallel computations have become affordable through the usage of consumer GPUs. The large performance increase offered by such devices has been utilized in many areas. The process of converting traditional CPU code to parallelized code is non-trivial. It often requires in depth knowledge of various areas related to GPU computing. This poses obstacles for those with relevant computational experience and more obstacles for those users lacking such experience. These are significant barriers to entry for GPU computing which restrict both the adoption of GPU computing and it's development. Various approaches exist to make GPU computing easier for the user. This manuscript presents the initial development cycle of a tool to transform C++ to CUDA in a configurable manner. The tool, *togpu*, improves upon existing approaches to lower GPU development entrance barriers, accelerate GPU developer workflow, and lay a foundation for further development. Minimizing the effort required to generate something functional and useful is a design goal in many areas — GPU development is no different.

The barriers of entry to effectively utilize GPUs for parallel processing are high in many areas. The large breadth and depth of knowledge a user must possess is one of the fundamental obstacles a user faces. It is common occurrence for users to lack the time to learn these areas sufficiently to ease development. Moreover, motivation is not necessarily present. For example, a physicist looking to accelerate molecular dynamic calculations may neither hold a passion for GPU development nor care what Single Instruction Multiple Data (SIMD) signifies for development. Similarly, a GPGPU expert with a deadline may be unable to dedicate the resources required to convert an image progressing algorithm into a GPU compatible form while ensuring high data throughput.

Strides in accessibility have been made through languages such as CUDA replacing previous techniques such as manipulating shader pipelines to perform computations. Despite these developments GPU accessibility remains limited. The potential benefits of parallel processing using consumer tier GPUs is high but the initial time investment by itself can be prohibitive. Including the time required to perform the operations of interest and workflow impacts such as maintaining separate GPU and CPU sources, the obstacles facing users grow further. These barriers are not limited to inexperienced users but instead also impact expert users.

An inexperienced user must know how to program, use a language that supports CUDA in some fashion and is applicable to the problem domain, and have knowledge of: parallel processing and programming, CUDA, any necessary APIs between CUDA and their language of choice, device/platform specifics, and GPU optimization and debugging if necessary. Similarly an experienced user must also have the aforementioned knowledge required of the inexperienced user but also may experience knowledge hurdles. The topics of optimization and debugging are such areas where intricate knowledge is often mandatory. The user may also be required to obtain at least functional domain specific knowledge if they are asked to implement algorithms versus optimizing existing code.

Existing solutions to lowering entrance barriers for parallel processing have shown great strides in accessibility while maintaining functionality. When using these solutions users continue to encounter various issues both during immediate development and in the overall development workflow. These approaches can be improved upon to minimize the remaining obstacles. It is common to require users to learn the equivalent of another API, while still requiring users to have in depth knowledge of GPUs and parallel computing. While in depth knowledge may become a necessity at stages where device/platform optimization and intricate performance monitoring take place, it should be minimized as much as possible to ease GPU development. Other occurrences such as mixing configuration and execution, forcing refactoring of code to adopt a new library, and closed source are just a few common issues which may accompany existing methods. By focusing on both development usage and workflow impacts we target mitigating common issues to lower GPU development barriers.

Related Work

Lowering barriers to entry for the usage of parallel processing on GPUs is not a new problem. Various approaches exist and are presented in this section. Exploration reveals existing solutions are largely spread throughout four categories: Domain Specific Language (DSL), Wrapper/Binding, Framework, and Source to Source Transformation.

Domain Specific Language

A Domain Specific Language (DSL) is a programming language that is suited to the needs of a particular domain. Constructs within the language facilitate common operations associated with the domain. For example, CUDA (or CUDA C as it is also referenced) is a DSL. CUDA provides access to functionality that is common within the GPU computing domain. Programming facilities to copy data to and from the GPU, launching kernels, declaration of device and host functions, and other related operations are bundled into this extended version of C. These operations are suited to the GPU computing domain. Other DSL examples include Microsoft Accelerator [12] and Copperhead [3].

While DSLs afford users working with GPU computing substantial convenience and ease of usage, they still stand as another learning obstacle — a time sink for the user. Some DSLs are a subset of a language, however they still require knowledge of data parallel operations, parallel computing knowledge, and additional language specific requirements and constructs. CUDA for example forces users to gain such knowledge. In addition to the other issues covered in the Workflow Complications section, this knowledge mandate is a shortcoming of DSLs. It is clear that a domain specific language, while useful, is not able to sufficiently mitigate knowledge requirements for inexperienced users performing feasibility exploration. These reasons lead us to investigate other approaches to lowering GPU development barriers. Wrappers or bindings are another approach to providing access to GPU capabilities.

Wrapper/Binding

A wrapper or binding is code that negotiates operations originating in one language with a functionally equivalent operation in the native library language. For example, JCUDA enables Java users to write calls to CUDA functions in Java while making the necessary calls to C functions through a Java Native Interface (JNI) layer [20]. Such libraries are extremely useful and enable the usage of CUDA in languages that are outside of C and C++. Using a wrapper, much of the knowledge required to use CUDA is required of the user. Additionally, users are responsible for any overhead required by the extra library layer (i.e., if the library handles automatic data management and you encounter an issue with it, users need to be aware of that to determine how to debug it). Wrappers may also be subject to the workflow issues identified in the Workflow Complications section. These properties clarify that while wrappers may be used, alone they are insufficient to achieve our target goals. A framework is another category of approach to easing GPU code development and adoption.

Framework

A framework provides CUDA implementations of libraries, functions, data structures, and operations to give users a small subset of GPU compatible algorithms. For example, the CUDA Basic Linear Algebra Subroutines (cuBLAS) provides a GPU version of the commonly used BLAS library [7]. Frameworks provide different facilities for users, some such as cuBLAS are focused on an area, others provide basic data structure translations. The library Thrust is an example [1].

Thrust is a parallel algorithm library that resembles the C++ Standard Template Library (STL) [1]. It provides access to extensive GPU-enabled functions in order to produce high-performance applications [1]. Thrust is similar in notion to libraries translated by NVIDIA, such as CUBLAS, but provides more generalized parallel structures [1]. Thrust requires that the user learn their API and still requires some knowledge of parallel computing. In conflict with the goal to provide a transparent as possible programmatic solution, these extra stipulations further differentiate Thrust. Another framework is available for Python titled Parakeet that provides Just In Time (JIT) compilation of Python code for a GPU [11].

With minimal annotations, Parakeet provides a strong framework for automatically generating and managing the execution of GPU code using JIT compilation [11].Parallelization is achieved by implementing NumPy operations in terms of four data parallel operations [11]. Parakeet utilizes heuristic based, array shape detection to determine algorithm execution location (CPU, GPU) [11]. Parakeet provides a functional demonstration of the usage of parallel operators to implement common functionality as a means of parallelization. A single annotation is required to identify a function for processing. Parakeet's minimal annotation usage lowers the amount of parallel domain knowledge required from users. However, some significant constraints preclude adoption outside of numerical computing. For instance, Parakeet parallelization only applies to Python NumPy (a Python numerical library) operations and developers are limited to scalar and array types [11]. Our application case is also targeting C++ as the source language. Nonetheless, Parakeet provides an excellent demonstration of feasibility and the substantial degree to which CUDA development barriers can be lowered.

Frameworks offer varying approaches but are susceptible to multiple issues. Often a framework provides only a fixed set of functionality, which may be too restrictive for many applications. A framework may also fail to support extensions, whether due to stagnation, licensing, software design, or other issues. Users are required to learn a framework, the provided functionality, limitations, requirements, and other attributes. Depending on the framework, a user may be forced to still learn various portions of parallel computing, device and platform specifics, and other applicable topics. Once familiarity with the framework has been established, it is then usually necessary to modify the source code to varying degrees in order to integrate the framework. In some instances this can be a monumental effort. Some of the issues present with frameworks are targets of this proposal.

Frameworks provide facilities for usage during GPU development. Similarly, source to source transformations may offer features for usage during development. In contrast, after initial development efforts additional features may be provided by source to source transformations such as optimizations for existing algorithms.

Source to Source Transformation

A source to source transformation is the act of taking one set of source code and generating another from it. Often the goal is optimization, some form of refactoring, or conversion from an input language to a different language. Automating this process is clearly desirable. For example, a user has a collection of Ruby code, performs a source to source transformation on it using some tool, and the resulting output is Ruby code that is now optimized to take advantage of a faster hashing approach. Another example, is an indentation tool that automatically indents code to conform to a specified format upon committing to a version control system. A source to source transformation lends itself to refactoring amongst other uses. With the support of a proper framework, analyzing and exploring source code can be done with various levels of abstraction and result in many interesting operations, from optimizations to automatic parallelization to updating deprecated code references.

Many source to source transformations exist, in various forms. Targeting specifically C++ to CUDA transformation, research revealed that existing options were available for automating conversions of C and C++ to CUDA code. Generally the conversion is aided through the use of annotations (usually in the form of pragma directives) and other annotations to outline areas to externalize to the GPU. In many cases these annotations require in depth knowledge of parallel processing for effective application. Additionally, the user must learn a new pseudo language composed of annotations which often map closely to CUDA operations or utilize underlying parallel concepts. Often such tools require preparation of a code base outside of special directives such as minimization of external library usage, restructuring code, changes to data storage, and other such modifications.

CUDA-Lite by Ueng et al. is an example of a tool that performs a source to source transformation focusing on easing the difficulty of working with GPU memory selection, transfer, and allocation when using CUDA [16]. CUDA-Lite uses source annotations provided by the user to help alleviate the burden of performing memory access coalescing. These annotations allow the user to denote portions of the code which CUDA-Lite utilizes to generate code that performs the necessary memory related operations. Such automation is valuable as memory utilization can have a substantial impact on performance — the authors noted a 2x-17x performance impact in preliminary results [16]. Ueng et al. also explained that CUDA-Lite performed on par with hand optimized CUDA code [16]. The value of automation is not limited to optimization of GPU code memory accesses but also extends to the core of GPU development: automatic parallelization of serial code.

GPSME is a toolkit to convert C and C++ code to OpenCL/CUDA while performing automatic parallelization [19]. GPSME focuses on the use of compiler directives: ”At its heart, the GPSME toolkit converts C and C++ code into OpenCL/CUDA by following compiler #pragmas” [18]. As previously discussed such application of annotations using #pragma directives is common. Both OpenMP (Open Multi-Processing) and the more recent standard OpenACC (Open Accelerators) utilize such annotations [6] [17]. The authors accomplish this by extending Mint, a C to CUDA translation tool leveraging the ROSE compiler [18]. By following GPSME specific, user provided annotations GPMSE outputs a single file containing both C++ and CUDA. Some guidelines are provided for what developers using GPSME should and should not include in their code. The guidelines contain suggestions such as including nested for loops, avoiding recursion/function calls/conditional logic/external library dependencies, and avoiding system operations [18]. The author also provides some suggestions on how to revise a code base to conform to these restrictions. Comparing against OpenMP and OpenACC using Polybench benchmarks, the authors determined GPSME performance exceeded the other test candidates [18].

To effectively gauge generated algorithm performance, Williams et al. compare OpenMP, OpenACC, Mint, and GPSME (Mint with modifications) using modified GPU versions of the PolyBench benchmarks [19]. To differentiate GPSME from other candidates some of the changes that GPSME contains versus the Mint toolkit and OpenACC are explored. The authors demonstrated that GPSME added performance increases substantially in some test cases. OpenACC and GPSME performed similarly. Interestingly, all of these candidates rely on extensive use of preprocessor directives. While such directives are useful, minimizing their usage is necessary to enhance user workflows. In most cases annotations reference parallel computing elements requiring some understanding of parallel computing. At the very least, a user must effectively learn the equivalent of a DSL composed of annotations. In many cases the annotations are very similar to that of CUDA (such as those for copying memory). Annotation usage also results in the combination of configuration and source code. This has direct impacts on workflow and usability. One example is that sharing configurations becomes more difficult due to version conflicts (whether inside or outside of a version control system). The approaches used in this manuscript aim to eliminate the usage of such annotations and directives as much as possible. While the authors acknowledge the place of annotations and large benefit they can provide, there are significant downsides to their usage which must be carefully considered. Instead of annotations, the user is provided behavior that is configurable outside of the source code and starts with opinionated defaults. This choice has workflow ramifications that extend outside of immediate usability and learning overhead.

At the time of writing, registering for GPSME's free trial fails. Also, the notion of a free trial, lack of source code availability, and inaccessible modifications to the ROSE engine in publications appear to suggest that the final product may be a commercial/proprietary entity. In contrast, the approach used for togpu focused at encouraging extension and providing a tool for the public. Hence the source code is provided under an open source software license and free of charge.

It is clear that source to source transformations provide a powerful approach to lowering GPU computing entrance barriers. When considering approaches, it is also necessary to look beyond the initial user experience to how the approach will fit into the users workflow. Further investigation revealed other potential complications with existing approaches.

Workflow Complications

Throughout all of the approaches specified in the related works section various workflow complications are noted. In general, these issues are the result of subtle features of the tools. While the attributes may be subtle, they can have large impacts on the workflow of users. The scope of a project may also determine the degree to which the impact is felt and thus it may not be detected during tool development until after an initial release. This section identifies various additional workflow complications that arise from both categories of approaches and specific implementations.

There are various maintenance issues that can come into play when transforming code for GPU compatibility. One such issue is the degree to which the approach being used hinders user adoption and modification. For example, if a DSL is used in a project it is likely that any user wishing to modify a significant portion of such a project would be required to learn that DSL. This initial cost applies to each approach available. In some cases this barrier is high and in others, such as with source to source transformation, the barrier can be lower.

A user may have to learn a framework to modify a single project. Instead, a project using source to source transformation tools may require a user to only know the input language — increasing accessibility to those without parallel computing knowledge. Moreover, such an approach encourages code reuse and reduces the amount of time an experienced developer may spend creating and modifying common code. An experienced user may wish to change the parallelization method used for a subset of 10 functions for example: instead of making the same changes 10 times, the user may change or configure the source to source transformation tool appropriately and run it again. Similar to a common clean and build process.

An additional issue is source divergence — a C++ code base now must somehow contain a CPU compatible version and a separate GPU compatible version. This often can result in the source code diverging either externally or internally. Two distinct code bases may emerge or alternatively, two instances of an algorithm exist within the same source. In either case, algorithm changes must propagate between algorithm instances. Every category of conversion is susceptible to this issue. Having to update two sets of code that do the same thing but have very different implementations (and subsequently different knowledge requirements) is clearly a suboptimal situation.

An initial attempt to mitigate this might be to add in some delegation code that determines which version of the algorithm to utilize and maintains functions for both CPU and GPU implementations. While there are advantages to this approach, it does not solve the issue but instead merely preserves the issue within a single code base. A more clear approach to this is to use something like GPSME to perform a source to source transformation, generating the GPU enabled code as required. However, as is common of source to source transformation tools for automating GPU parallelization, other issues must be considered.

As noted in other sections, GPSME and other similar tools (OpenMP, OpenACC, etc.) rely on the extensive use of source code annotations. Annotations, while powerful, may create an unanticipated workflow impediment which may not reveal itself until a more advanced stage in a project. Annotations are a way to provide hints to the engine performing the transformation. In some cases a user may wish to change these options, for their own experimentation, specific platform/device testing, or other reasons. As a result the user is forced to modify the source code. First, the breadth and depth of their changes may depend on what they wish to change and second, their source changes must now be reconciled against other developments when using revision control software. This project aims to minimize the use of annotations due to common recurrence, potential knowledge requirements, and the need for externalizing configuration parameters.

Configuration parameters are also relevant when dealing specifically with source to source transformation approaches and their output. Often the readability of generated code is suboptimal — it is very clear that a machine generated the code. One of the advantages of source to source transformation is that in the event it is not perfect, developers may still benefit due to the partial work completed in the transformation. Users who are attempting to reconcile bugs, identify functionality, or optimize code manually must cope with code readability post transformation. The availability to configure options for readability, even as simple as prefix and suffix names, would yield greater readability for users.

Another workflow obstruction is the license of the tools being used and the impact on the output software. Closed source libraries impede extension by users, even users who have paid for the software. It is desirable to maximize extensibility both from a software design standpoint and a code accessibility standpoint. Having high cost software would deter users, including students and researchers. As a result the proposed software will be available at no cost and under an open source license.

These complications, some of which may be subtle, can play a big role in the user software development cycle. As a result, the development of togpu considered them from the beginning. Focus was on the user from the start to build a useful, usable tool. The consideration of these complications and the state of current research yields various other potential avenues for extension. By building on the research of existing methods of GPU development, this project is implemented to minimize the impacts of the identified workflow complications and establishes an extensible, C++ to CUDA transformation tool foundation.

Implementation

Easing GPU development, specifically for image processing, by automating parallelization can be achieved using a source to source transformation from C++ to CUDA. The implementation of togpu is covered in this section. The chosen supporting libraries and their usage in the tool are discussed. Approaches to automatic parallelization are detailed and underlying elements of the transformation process are explored.

An iterative process was used throughout building togpu. Each iteration increased the feature set to handle more complex input. This allowed each iteration to focus on a specific area to improve while moving in the direction of the end goals. For example, one cycle may focus on supporting fixed size arrays and the next cycle may focus on adding optimizations to perform batch operations on such arrays. At the core of the investigation is the determination of which, if any, supporting libraries would be used to perform source analysis for the transformations.

Source Analysis

The requirements and desirable features for supporting libraries include: capability to perform source to source transformation, ability to parse at least C++11 (more languages/standards is desirable), provides access to parsed code, provides facilities to identify code sections of interest (loops, functions, locations of function implementations) with and without the use of annotations, supports extension (including a usable API), ability to output CUDA, appropriate licensing, supports multiple platforms, and has development activity and support such that the likelihood of library stagnation is a small as possible. Each candidate was evaluated against these requirements, the context of applicable goals for this project, usability, and against each other. One candidate that was reserved as a viable, backup option was the creation of such a library. It was preferable however to avoid this scenario if possible due to many reasons such as time constraints and community support.

Initial exploration revealed multiple candidates and after further research Clang/LLVM (Low Level Virtual Machine) emerged as the clear choice. Existing options were available for automating source to source transformations. After a cursory search for candidate frameworks, the ROSE compiler infrastructure and LLVM emerged as strong options. Their selection was due to their maturity and apparent potential to meet this project's requirements. In short, ROSE was the basis for GPSME, a C++ to GPGPU automatic parallelization toolkit, was open source, and appeared to have sufficient documentation. LLVM was chosen as the basis for nvcc, the compiler provided by NVIDIA for CUDA, was also open source, appeared to have sufficient documentation, and offered other useful tools — such as Clang — based on LLVM [8]. Each candidate, ROSE and LLVM, was investigated further and evaluated in the context of this project.

The ROSE compiler infrastructure was one candidate considered for the base of this project. An open source project, having substantial language support, and targeting application optimization and source-to-source transformations, ROSE appeared a more than suitable framework. A dearth of support, incorrect documentation, out of date libraries, missing Microsoft Windows support, and a lack of activity (including the more recent EDG4 branch) proved substantial detriments to ROSE usage and adoption [10]. In order to setup a working environment with ROSE it became necessary to construct a supplemental guide to the existing documentation which still resulted in compilation difficulties and failure. The API provided by ROSE shows the large influence of the SAGE III library. ROSE is based upon SAGE III thus this is not surprising. However, this conjunction introduced the necessity of mixing ROSE API members and SAGE API members when developing. That aside, the concepts of the API were sufficient and the API specifics were fairly straightforward [10]. However, when compared against Clang and LLVM with respect to the project's goals, ROSE met fewer of the requirements.

The LLVM (Low Level Virtual Machine) Project is an open source umbrella project consisting of a set of compilation and tool chain projects. The foundation of the project, LLVM IR (Internal Representation), enables tools such as a target-independent optimizer to work with various code bases [5]. Some projects include Clang, a C/C++/Objective-C compiler for LLVM which boasts substantial performance increases over GCC and offers enhanced usability [5]. Clang also provides libraries which allow users to parse and interact with C/C++/Objective-C code at various abstraction levels. Clang and LLVM are cross-platform [5]. LLVM also contains a native debugger as well as a native implementation of the C++ Standard Library. Many other projects are included in LLVM, including an official NVIDIA NVPTX port which is included in more recent versions of the CUDA toolchain [5].

Clang is a cross-platform C/C++/Objective-C compiler for the LLVM native IR [4]. Clang provides libraries enabling interaction with parsed code at various levels of abstraction. Clang generates an AST (Abstract Syntax Tree) which closely resembles the input code, facilitating quick user comprehension [4]. The provided libraries allow interaction with the generated AST for different purposes [15]. These libraries are libClang and libTooling [14].

It is possible to work with input code through Clang in multiple ways. The three available methods are: using libClang, using the Clang plugin system, and using libTooling. Choosing the appropriate approach requires evaluation of the available methods. The Clang documentation provides a set of guidelines which outline the capabilities of each method [14]. LibClang is insufficient as it may be desirable to access low-level abstractions inside Clang's AST. It is stated, "[Do not use LibClang when you] ... want full control over the Clang AST" [14]. Clang Plugins is not a viable approach as it sacrifices the ability to run as a standalone tool and the ability to run outside of the build process [14]. The third option, LibTooling, is an appropriate match. It provides low-level access to the Clang AST, supports executing independently of the build process, and is suggested for refactoring tools - an applicable relative of source-to-source translation [14].

As for technical requirements, a C++11 compatible compiler is required. Development, compilation, and testing was performed on Linux using Clang 3.5, CMake 3.3, and CUDA 6.5. Compilation and usage is possible on other operating systems as well as LLVM and Clang are also cross-platform.

The project remains tightly related to Clang and LLVM in styling and idioms to ease developer familiarization. Matchers and transforms are abstractions around the Clang ASTMatcher and MatchFinder::MatchCallback classes. These abstractions make it possible to quickly add new, reusable transforms that are configurable.

Automatic Parallelization

It is intended that this project facilitate and help advance the field of parallelization algorithm research in accordance with the goal of lowering GPU development barriers and workflow enhancement. Automating the task of parallelizing an algorithm is non-trivial. Extensive research has gone into this area both in terms of parallelization approaches and their automation. While automatic parallelization approaches and methods were core elements in this project developing a novel method of parallelization automation was not the focus. Novel parallelization algorithms may arise in the future through the usage or development of this project but to reiterate, it is not the focus.

The initial parallelization method integrated is the common vectorization approach. This approach splits operations on the input data into statements that can be executed in parallel. The canoncial example is vector addition — each addition operation may be executed independently using the same set of instructions functioning on different portions of the input data (SIMD execution). It is anticipated that more parallelization methods will be integrated in the future.

Several of the algorithms are implemented in a format that is naturally amenable to vectorization. Unlike the others, the histogram algorithm required some minor modifications of the parallelization transformation to detect dependencies between the input and output parameters. Using Clang/LLVM, these statement dependencies could be detected and acted upon during the transformation process. A process which is on the development roadmap to be replaced with more advanced dependency analysis. A large boon is the configurable nature of parallelization approach application. As more approaches are integrated, they may be applied conditionally to various algorithms. Advanced approaches such as performing analysis to determine which method of parallelization is most appropriate is another potential future enhancement. As the algorithm collection grows the potential value as an asset for both researchers and production developers also increases.

Matcher

Matchers are based on the Clang ASTMatcher but with a few extra elements, such as naming, registration, and configuration support to ease development and meet the needs of this project [13]. A matcher searches for specific elements in source code. Nested for loops, a while loop with a single condition, or the member named "flag" in the class "Country" are potential examples. See Figure 1. The units of code that are of interest may be bound to various parameters and the result is passed to the associated transforms.

Figure 1. *Example Matcher*

```
// Locate a function declaration that is also a function
      definition , defining a
// function named FunctionName that may or may not have
     a return statement.
// Bind the function declaration to the key:
     BindFunctionDeclName. If there is a
// return statement bind it to the
     BindFunctionDeclReturnStatementName key.
DeclarationMatcher FunctionMatcher = functionDecl(allOf(
     isDefinition (),
    hasName("FunctionName")),
    anyOf(hasDescendant(
    returnStmt ().bind(
        "BindFunctionDeclReturnStatementName")
        ),
    anything ()
    )
    ).bind("BindFunctionDeclName");
```

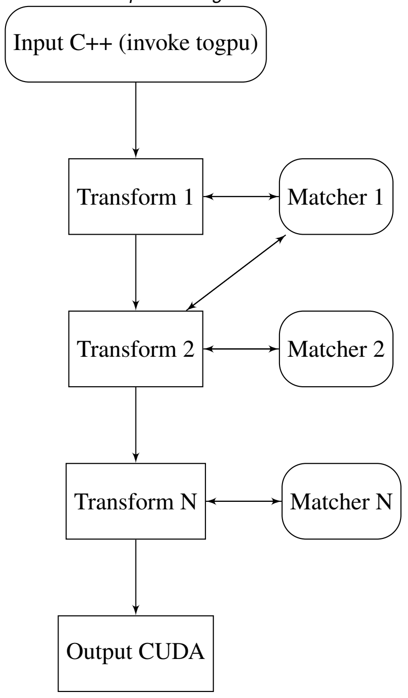
Transform

Each transform may perform a set of modifications to the input code. Once a matcher has located the desired area(s) of code various elements are bound to a key. The associated transforms receive results from the matcher and can access bound elements via lookup using the appropriate key. These keys can be exposed to the user for configuration. While the bound elements of the code from the associated matcher result are generally the transformation target, access to other portions of the source code is available.

Transform Pipeline

The transform pipeline currently follows the execution path inherent to Clang. Each transform is initialized, associated with the appropriate matchers,and added to the pipeline in the order requested for execution. Transform execution order is determined by the configuration file. The transforms execute in a serial fashion. As this project is in it's foundational stages, the transform pipeline is likely to expand. The transform pipeline and other process elements are also configurable via a configuration file.

Figure 2. *Transformation Pipeline Diagram*



Configuration

The tool, transformation process, and pipeline are configurable. Options may be set that influence the general tool operation (e.g.: location of resource files, etc.) via command line arguments or a configuration file. Individual transforms and matchers are configured using a configuration file. Each transform and matcher is represented by a sequence of mappings (keys and associated values) which is parsed by the tool. Matchers may be associated with transforms using the configuration file. Similarly the parameters assigned by matchers and utilized in transforms may be modified, allowing both matchers and transforms to be reused. The order of operations used in the transform pipeline corresponds to the order of the transforms in the configuration file. Thus if TransformA should be performed before or after TransformB, the configuration of TransformA can be moved to precede or succeed TransformB to achieve the desired ordering. The configuration file is constructed using Yet Another Markup Language (YAML) [2]. YAML is human readable, widely used, has many available parsers including one that ships with LLVM, and is documented sufficiently in the event creating a new parser becomes necessary [2]. An example configuration is shown in Figure 3.

Assumptions

Various assumptions are made for this project. The primary assumption is that the user has access to the C++ source code for the target project and preferably, it's dependencies. It is also

Figure 3. Sample Transformation Pipeline YAML Configuration

```
---
transforms:
  - name: AddCUDARuntimeIncludeTransform1
    type: AddCUDARuntimeIncludeTransform
    matcher_names:
      - FunctionDeclarationMatcher1
    options:
      - key: bound_function_decl_name
        value: 'KernelDecl'

  - name: CopyKernelParametersTransform1
    type: CopyKernelParametersTransform
    matcher_names:
      - FunctionCallMatcher1
    options:
      - key: bound_function_call_name
        value: 'KernelFunctionCall'
      - key: return_parameter_name
        value: 'return_parameter'
      - key: prefix_device_variable
        value: 'device_'

matchers:
  - name: FunctionDeclarationMatcher1
    type: FunctionDeclarationMatcher
    options:
      - key: function_name
        value: 'kernel'
      - key: bind_function_decl_name
        value: 'KernelDecl'
      - key:
            bind_function_decl_return_statement_name
        value: 'KernelReturn'

  - name: FunctionCallMatcher1
    type: FunctionCallMatcher
    options:
      - key: function_name
        value: 'kernel'
      - key: bind_function_call_name
        value: 'KernelFunctionCall'
...
```

assumed that the extent of language support is sufficient for the user's project. In cases where it is insufficient users may engage in further development efforts to increase the language coverage to their satisfaction. An assumption is made that the output source code desired is CUDA and correspondingly that the user wishes to utilize such code on a CUDA supporting platform. It is assumed that the ability to identify source code sections for transformation is doable with fewer or zero annotations than that of other approaches to C++ to CUDA transformation. It is assumed that the user is able to utilize the proposed software and it's dependencies with respect to licensing, operating system, device compatibility, platform, and/or other aspects.

Limitations

Limitations are present which are not unexpected of early stage projects. Any of these limitations may be addressed with later development efforts. One of the limitations of this project is that it is unlikely that the generated code will always outperform hand optimized code. There may be very fine-grained optimizations that are not applicable to include in the project yet may be manually inserted to yield performance gains. This system still makes the generation of the base code for an experienced user to optimize much easier, amongst other workflow benefits. Another limitation, especially in the initial stages, is the dependence upon extension.

Due to the project maturity level and very limited set of parallelization algorithm implementations, the applications of togpu are limited to specific input algorithms. Moreover, bugs not included, there are cases where the existing transformations are insufficient to convert specific code to CUDA. Similarly, support for specific device and/or platform optimizations, as well as other interesting additions to the tool must be deferred for future development. Each of these limitations may be mitigated by future extension efforts.

Another implementation inherent limitation is that by specifically targeting C++ to CUDA, other languages cannot be used with the same code base. Clang/LLVM offers support for C, C++, Objective-C, and Objective-C++ which may be leveraged in the future with little changes to support those input languages. That does not mean that the approaches and progress made in developing this tool cannot be applied elsewhere or that additional tools may be introduced to handle more input languages. Another instance of a similar tool for a different input language could be an interesting development.

Currently togpu can only work with a limited set of language elements in input algorithms. This is largely due to time constraints instead of some limitation of the underlying libraries. Fixed size arrays must be used for data collections. Looping operations must be performed using *for* constructs. While external libraries may be used both inside and outside of the function being parallelized, usage within the target function is discouraged as no actions are taken other than copying the calls verbatim. As development continues these restrictions should change and hopefully vanish.

Experimentation

Algorithms selected for experimentation represent of common classes of problems in image processing. Vector addition is the exception but is included as it is a common, introductory GPU development problem. Each algorithm was implemented in C++ by hand, using an initial, non-optimized (naive) approach. For example, a separable convolution algorithm was not used, neither was an accumulator but instead a sliding window. The use of naive approaches was to create algorithms representative of a user new to the topic area to improve transformation processes. Each algorithm was first constructed to run in serial on a CPU - multiple threads were not used. Second, the same algorithm was translated manually to CUDA (version 6.5) run on the GPU. Automatic transformation using togpu is then applied and the result are shown in Table 1.

Each algorithm is benchmarked and results are presented using executions or cycles per second. Only the function call or kernel call has been appropriately instrumented thus each cycle represents one execution of the algorithm. Each CUDA kernel launch was followed by a cudaDeviceSynchronize call to ensure proper benchmarking (kernel launches return before the operation has completed). Each algorithm was executed in a loop 1000 times on a Nvidia C1060. Three algorithms were constructed and benchmarked: CPU, GPU, and Generated GPU. The CPU algorithm is the original serial source code to be transformed. The GPU algorithm is the CPU algorithm manually translated to a CUDA version used for reference when constructing transforms. The GPU algorithm should have very similar performance to the Generated GPU algorithm. The Generated GPU algorithm is the result of using the automatic transformation tool on the CPU source.

Vector addition is performed using two 10000 component vectors. A general 3x3 matrix kernel convolution filter was implemented. The specific kernel variant run was a box blur (identity matrix). A color inversion filter was implemented which performs a bitwise inversion of each color channel on the source image. The histogram algorithm implemented utilizes 256 bins and represents a set of algorithms where output accesses must be synchronized in some manner (e.g., output[input[i]]++). Each image processing algorithm operated on a 256 x 256 (height x width) RGB image.

Table 1. Algorithm Execution (*executions/second — higher is better*)

Algorithm	CPU	GPU	Generated GPU
3x3 Convolution 256x256 RGB	4956	7172	7134
Color Inversion 256x256 RGB	19271	28662	28484
Histogram 256x256 RGB 256 bins	2074	1576	1479
Vector Addition 10000x1	16441	70430	70827

Effective and quantitative measurement of the differences in development time when comparing togpu usage versus traditional development is non-trivial. User experience levels must be taken into account and quantified, algorithm familiarity should be considered, and various other factors must be examined. While a significant, formal user study is possible, at this stage in development the tool is not mature enough to mandate such a study.

Anecdotal evidence suggests togpu significantly expedites GPU development both in source creation and workflow. Inexperienced users were able to quickly configure the transformation pipeline to explore new options and suit their needs. The ability to select a specific function for optimization provided usable scoping and eased usage in existing source code as not all the code was required to be modified. Configurations were easily shared between team members. Experienced users were able to eliminate boilerplate code using transformations and could edit the resulting code as needed. The generated code maintains high readability and is accessible while corrections can be applied either to the output code or integrated back into the producing transforms. New transforms can be added as required and other developers can easily use the additional transformations.

Discussion

The results shown in the results section and Table 1 point towards the benefit and effectiveness of automating C++ to CUDA transformation. The performance results combined with anecdotal user feedback clearly displays efficacy but more in depth and formal user workflow impact analysis is obviously desirable. Such a study is more appropriate for the future when the tool has matured further and is able to be used on higher complexity source code.

Many of the problems or algorithms in image processing are similar such that we anticipate the current state of the tool may be applicable to many more algorithms than expected. Supporting this hypothesis, out of the algorithms specifically investigated the results were achieved using only two methods of rudimentary parallelization.

The increase in performance shown when using the produced GPU color inversion algorithm suggests that per-pixel operations such as color inversion map well to SIMD design. This is expected and reinforced by the performance increase shown by the introductory vector addition algorithm as the problems are extremely similar.

The histogram shows over a quarter slow down when run on the GPU. Initial this might seem counterintuitive as a histogram operation is trivial but this slowdown is anticipated. Optimizing a histogram calculation for parallel execution is an explored problem and has been investigated in depth [9]. This difficulty arises due to the dependency on synchronized output — each bin in the histogram is dependent upon any number of input pixels. The general case, the class of algorithms that share common features with histograms, is the focal point of interest. Utilizing or developing intelligent parallelization techniques that construct efficient CUDA output for such scenarios becomes a clear need from these results. Intuitively this issue may persist within transformation software as recognition or detection of specific patterns may be incomplete.

Overall the results are suboptimal in terms of execution performance due to lack of optimizations and an absence of more advanced parallelization methods. Even with these issues, the results point out that automatic transformation retains value. An inexperienced developer may use the tool as a proof of feasibility or proof to warrant further investigation/optimization by a specialist the tool is useful and for an experienced developer, having an initial starting point from which to optimize or correct removes some time and burden. These types of boosts are but two of many potential areas of benefit. When used as a litmus test to determine if further resources should be dedicated to optimizing an algorithm on the GPU it is important to note that poor performance in a generated algorithm does not mean the algorithm cannot be mapped effectively for SIMD execution. The case of the histogram is one such example.

Only measurement of the execution time of each run of the selected algorithm is displayed. This means that other considerations such as bidirectional host and device memory transfers are not included. While this is necessary for gauging only algorithm execution, in terms of real world usage another exploration of the comparison of complete program execution times could prove useful. In the future more advanced parallelization algorithms, such as methods utilizing batch kernel execution and asynchronous data streaming, may have been introduced which would make such a comparison much more intriguing.

Future Work

There are many opportunities to extend togpu. A primary area of future work is to extend the capabilities supporting automatic parallelization. Adding additional methods of parallelization beyond simple methods such as vectorization would enable many interesting benchmarks as well as increase the utility of the application in production usage scenarios. Conveniently transforms and the overall system enables anyone to experiment with parallelization approaches. Expanding beyond applications within image processing is another area where development would bring significant value to the project. Some other intriguing avenues are exploring the benefits of integrating device specific optimizations and investigating an interactive transformation mode.

Conclusion

We presented a new tool, *togpu*, to perform CUDA to C++ transformations using Clang/LLVM to aid users in harnessing parallel computing power by lowering GPU development barriers. Collected results from application of automatic transformations to common image processing algorithms displayed effective generated algorithm performance. Implementation details were presented and future potential development areas were outlined. Further development to improve the tool may facilitate adoption in production and research environments, easing GPU development for many users.

Acknowledgments

The authors acknowledge the support of Wright State University.

References

[1] N. Bell and J. Hoberock. Thrust: A Productivity-Oriented Library for CUDA. In *GPU Comput. Gems Jade Ed.*, pages 359–371. 2012.

[2] O. Ben-Kiki, C. Evans, and B. Ingerson. YAML Aint Markup Language (YAML) Version 1.2, 2009.

[3] B. Catanzaro, M. Garland, and K. Keutzer. Copperhead: compiling an embedded data parallel language. *Symp. Princ. Pract. parallel Program.*, pages 47–56, 2011.

[4] C. Lattner. LLVM and Clang: Advancing compiler technology. *Keynote Talk, Free Open Source Dev. ...*, page 28, 2011.

[5] C. Lattner and V. Adve. The LLVM Compiler Framework and Infrastructure Tutorial. *Lang. Compil. High Perform. Comput.*, pages 15–16, 2005.

[6] S. Lee, S. J. Min, and R. Eigenmann. OpenMP to GPGPU: A Compiler Framework for Automatic Translation and Optimization. In *PPoPP'09*, pages 101–110, 2009.

[7] NVIDIA. cuBLAS.

[8] NVIDIA. CUDA LLVM Compiler.

[9] V. Podlozhnyuk. *Histogram calculation in CUDA*. 2007.

[10] D. Quinlan, M. Schordan, R. Vuduc, Q. Yi, T. Panas, C. Liao, and J. J. Willcock. ROSE Tutorial : A Tool for Building Source-to-Source Translators Draft Tutorial (version 0 . 9 . 5a), 2013.

[11] A. Rubinsteyn and E. Hielscher. Parakeet: a just-in-time parallel accelerator for python. *Proc. 4th ...*, 2012.

[12] D. Tarditi, S. Puri, and J. Oglesby. Accelerator: using data parallelism to program GPUs for general-purpose uses. In *Proc. 12th Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, volume 34, pages 325–335, 2006.

[13] The Clang Team. AST Matcher Reference.

[14] The Clang Team. Choosing the Right Interface for Your Application - Clang 3.6 documentation.

[15] The Clang Team. Introduction to the Clang AST - Clang 3.6 documentation.

[16] S. Ueng and M. Lathara. CUDA-lite: Reducing GPU programming complexity. ... *Compil. ...*, pages 1–15, 2008.

[17] W. Wang, L. Xu, J. Cavazos, H. H. Huang, and M. Kay. Fast acceleration of 2D wave propagation simulations using modern computational accelerators. *PLoS One*, 9, 2014.

[18] D. Williams. Automatically converting C / C ++ to OpenCL / CUDA.

[19] D. Williams, V. Codreanu, P. Yang, and B. Liu. Evaluation of autoparallelization toolkits for commodity graphics hardware. *10th Int. Conf. Parallel Process. Appl. Math.*, 2013.

[20] Y. Yan, M. Grossman, and V. Sarkar. JCUDA: A programmer-friendly interface for accelerating java programs with CUDA. In *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, volume 5704 LNCS, pages 887–899, 2009.

Author Biography

Matthew Marangoni received a BS in Computer Engineering and Computer Science (2011) and MS in Computer Engineering (2013) from Wright State University and is currently pursuing his Computer Science and Engineering PhD advised by Dr. Thomas Wischgoll. Matthew's work has involved HCI, VR, UX/UI, GPUs, data visualization, ML, web development and many additional areas. He has previously presented at VDA and is an ACM and TBP member.