Supporting CUDA for an extended RISC-V GPU architecture

Ruobing Han
hanruobing@gatech.edu
Georgia Institute of Technology

Blaise Tine
blaisetine@gatech.edu
Georgia Institute of Technology

Jaewon Lee
jaewon.lee@gatech.edu
Georgia Institute of Technology

Jaewoong Sim
jaewoong@snu.ac.kr
Seoul National University

Hyesoon Kim
hyesoon@cc.gatech.edu
Georgia Institute of Technology

ABSTRACT

With the rapid development of scientific computation, more and more researchers and developers are committed to implementing various workloads/operations on different devices. Among all these devices, NVIDIA GPU is the most popular choice due to its comprehensive documentation and excellent development tools. As a result, there are abundant resources for hand-writing high-performance CUDA codes. However, CUDA is mainly supported by only commercial products and there has been no support for open-source H/W platforms. RISC-V is the most popular choice for hardware ISA, thanks to its elegant design and open-source license. In this project, we aim to utilize these existing CUDA codes with RISC-V devices. More specifically, we design and implement a pipeline that can execute CUDA source code on an RISC-V GPU architecture. We have succeeded in executing CUDA kernels with several important features, like multi-thread and atomic instructions, on an RISC-V GPU architecture.

KEYWORDS

CUDA, RISC-V, Code Migration

ACM Reference Format:
Ruobing Han, Blaise Tine, Jaewon Lee, Jaewoong Sim, and Hyesoon Kim. 2021. Supporting CUDA for an extended RISC-V GPU architecture . In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

RISC-V is the most popular choice for researchers in the academic community and engineers in hardware companies. The most important reason is its open-source spirit. These open-source licenses encourage many researchers to devote themselves to the development of a mature ecology for RISC-V, and thus, in turn, more and more people are willing to join the community, as there are existing fancy codes, hardware designs, and so on.

In the RISC-V ecology, the software support is the bottleneck for the blooming of the RISC-V community. Although OpenCL is an open platform for heterogeneous computing, due to the stability and software tool chain support, CUDA has been used widely. Unfortunately, CUDA source code can only be compiled and then executed on NVIDIA's devices, which is a major obstacle to using RISC-V for a wide range of applications, especially high-performance computing and machine learning workloads.

One way to solve this dilemma is to use code migration[8, 10, 13]. Instead of using the default method to compile CUDA source code with NVIDIA's compiler, some researchers try to parse and modify the source code to other high-level languages; more detail is shown in Sec. 2.1. However, because these methods highly rely on the high similarity between CUDA and the target high-level languages, they are not general solutions. Another solution is to build a compiler that directly compiles high-level CUDA language into a low-level RISC-V binary file. To the best of our knowledge, although there are translators that support generating RISC-V, none of them can handle CUDA source code.

Thus, in this project we propose and build a pipeline to support an end-to-end CUDA migration: the pipeline accepts CUDA source codes as input and executes them on an extended RISC-V GPU architecture. Our pipeline consists of several steps: translates CUDA source code into NVVM IR[4], converts NVVM IR into SPIR-V IR [7], forwards SPIR-V IR into POCL[5] to get RISC-V binary file, and finally executes the binary file on an extended RISC-V GPU architecture. We choose to use an intermediate representation (SPIR-V) for two reasons 1) RISC-V is still in development and has a lot of extensions, so we should not directly convert CUDA into RISC-V, as it will make supporting new features in the future difficult for our pipeline; 2) we want to make our pipeline more general so that we can support CUDA as front-end and RISC-V as back-end. Our pipeline is represented by Fig. 1.

In conclusion, the main contributions of our paper include the following:

- propose and implement a pipeline for executing CUDA source code on RISC-V GPU;
- build a translator support translating from NVVM to SPIR-V[1];
- pipeline that is easy to maintain and further support other front-end languages and back-end devices.
- pipeline that is lightweight, which can be executed without NVIDIA GPUs
- extend existing POCL to support RISC-V back-end

The rest of this paper is organized as follows. Section 2 provides a survey that describes various attempts to migrate CUDA

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

[1] https://github.com/gthparch/NVPTX-SPIRV-Translator

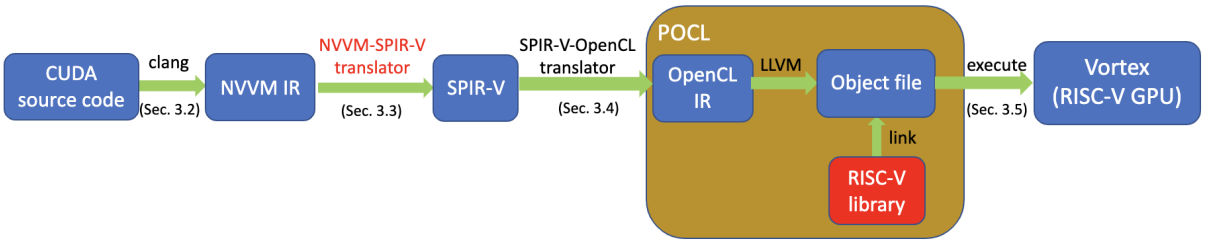

Figure 1: Overview of pipeline. The red part (NVVM-SPIR-V translator and RISC-V library) is developed in this work.

source code and introduces the current Available frames used in our pipeline. Section 3 uses a simple example to go through the entire pipeline to give a brief introduction of each phase in the pipeline. Section 4 records how to support several important features in CUDA. Finally, our concluding thoughts are in Section 5.

2 BACKGROUND AND RELATED WORK

2.1 Program migration

Many existing works[1, 14] aim to migrate from CUDA to other representations for different back-end devices.

HIPIFY[2] is a tool to translate CUDA source code into portable HIP C++, another high-level language proposed for AMD devices. HIPIFY has two methods to translate CUDA to HIP C++: hipify-perl and hipify-clang. Hipify-perl is quite straightforward; it heavily uses regular expressions to translate CUDA keywords to corresponding HIP C++ keywords. Instead of word-to-word replacement, Hipify-clang uses a more complex method: it adds an adaptor in Clang. It uses Clang's syntax analysis, converts CUDA source code into an abstract syntax tree, uses transformation matchers to convert this syntax tree, and finally generates HIP C++. Both methods can only be used for HIP C++ or other source languages whose grammars are highly closed to CUDA. HIPIFY is a translator from a high-level language to another high-level language, as most of its workload is done at the lexical level. To support executing CUDA on AMD devices, users need to first use HIPIFY to convert CUDA into HIP C++ and then compile the generated code with the HIP C++ compiler. We cannot use the same method to migrate CUDA on RISC-V devices, as there is not a corresponding high-level language for RISC-V.

SYCL[6, 15], proposed by the Khronos group, is another project that focuses on deploying source kernels on different devices. It is a high-level programming model that aims to improve programming productivity on various hardware accelerators. It can be regarded as a series of libraries of C++. These libraries provide APIs needed to write programs that can be deployed and executed in various back-end devices without modifying the code. This is much like OpenCL, which also uses C/C++ for high-level users, and generates host/device programs and executes them automatically for different back-end devices. Compared with OpenCL, SYCL is at a higher level; it provides primitives to support users to implement programs for devices directly, instead of regarding the code as a string and directly forwarding it to back-end devices' drivers. Thus, SYCL is highly portable thanks to its high-level abstraction. However, SYCL cannot solve our dilemma, as it does not support CUDA. SYCL can not migrate the existing CUDA source code to execute it with RISC-V. Instead, it requires users to re-implement it with APIs provided by SYCL.

[2]https://github.com/ROCm-Developer-Tools/HIPIFY

2.2 Intermediate Representation

In modern compiler design, a compiler can always generate object files for different back-end devices with various input languages. To support multiple source languages with multiple back-end devices, compilers first compiles different source languages to a standard intermediate representation (IR) and then emits this IR into different binary files according to our target back-end devices. Our pipeline has three involved IRs: NVVM IR, SPIR-V, and OpenCL IR. In Fig. 2, we show how these three IRs represent a vecadd example. IRs in Fig. 2(a) and (c) have similar format, most of instructions are same except some call instructions. These is due to the difference between NVVM and OpenCL IR is only for built-in functions: they have different built-in function($llvm.nvvm.read.ptx.srge.tid.x$ and get_local_id) for a same primitive (get the x-dim thread index). SPIR-V IR is total different with NVVM and OpenCL IR, not only for different built-in functions, but also different instructions for load/-store/binary op etc.

2.2.1 NVVM. NVVM IR[4], proposed by NVIDIA, is a compiler IR used to represent GPU compute kernels. In the general case, to execute CUDA source code on NVIDIA GPUs, we need to call nvcc to compile CUDA into NVVM IR, and the GPU driver will further translate the NVVM IR into a binary code that can be run on the processing cores of NVIDIA GPUs. NVVM IR is highly compatible with LLVM[11]; it supports a subset of LLVM IR along with a defined set of the build-in functions used to represent GPU programming concepts.

2.2.2 SPIR-V. SPIR-V [7] is the fifth version of SPIR. SPIR (Standard Portable Intermediate Representation) is an intermediate representation (IR) used for expressing parallel computation and GPU-based graphics. SPIR is a general IR that can allow high-level users to use various front-end's programming languages. SPIR-V can be used as an isolation layer that departs the front-ends programming language (e.g. MLIR, OpenCL, OpenGL) and low-level compute architecture. With SPIR, we first compile the source code into SPIR IRs and forward these IRs to devices. Thus, we can eliminate the need for high-level language front-end compilers in device drivers. This will also relieve the kernel launch time, as we do not deal with complex and abstract high-level languages, but rather the hardware-friendly SPIV IR. Besides, with SPIR as an isolation layer,

```
define dso_local void @_Z6vecaddPiS_S_(i32* %a, i32* %b, i32* %c) {
entry:
  %0 = tail call i32 @llvm.nvvm.read.ptx.sreg.tid.x() #2, !range !10
  %idxprom8 = zext i32 %0 to i64
  %arrayidx = getelementptr inbounds i32, i32* %a, i64 %idxprom8
  %1 = load i32, i32* %arrayidx, align 4, !tbaa !11
  %arrayidx2 = getelementptr inbounds i32, i32* %b, i64 %idxprom8
  %2 = load i32, i32* %arrayidx2, align 4, !tbaa !11
  %add = add nsw i32 %2, %1
  %arrayidx4 = getelementptr inbounds i32, i32* %c, i64 %idxprom8
  store i32 %add, i32* %arrayidx4, align 4, !tbaa !11
  ret void
}

declare i32 @llvm.nvvm.read.ptx.sreg.tid.x() #1
```

(a) NVVM

```
%9 = OpTypeFunction %void
                    %_ptr_Function_uint
                    %_ptr_Function_uint
                    %_ptr_Function_uint
%__spirv_BuiltInLocalInvocationId = OpVariable %_ptr_Input_v3ulong Input
%10 = OpFunction %void None %9
...
%15 = OpLoad %v3ulong %__spirv_BuiltInLocalInvocationId
%16 = OpCompositeExtract %ulong %15 0
%17 = OpUConvert %uint %16
 %idxprom8 = OpUConvert %ulong %17
 %arrayidx = OpInBoundsPtrAccessChain %_ptr_Function_uint %a %idxprom8
%20 = OpLoad %uint %arrayidx Aligned 4
%arrayidx2 = OpInBoundsPtrAccessChain %_ptr_Function_uint %b %idxprom8
%22 = OpLoad %uint %arrayidx2 Aligned 4
%add = OpIAdd %uint %22 %20
%arrayidx4 = OpInBoundsPtrAccessChain %_ptr_Function_uint %c %idxprom8
OpStore %arrayidx4 %add Aligned 4
OpReturn
OpFunctionEnd
```

(b) SPIR-V

```
define spir_kernel void @_Z6vecaddPiS_S_(i32* %a, i32* %b, i32* %c) {
entry:
  %0 = call spir_func i64 @_Z12get_local_idj(i32 0) #1
  %1 = trunc i64 %0 to i32
  %idxprom8 = zext i32 %1 to i64
  %arrayidx = getelementptr inbounds i32, i32* %a, i64 %idxprom8
  %2 = load i32, i32* %arrayidx, align 4
  %arrayidx2 = getelementptr inbounds i32, i32* %b, i64 %idxprom8
  %3 = load i32, i32* %arrayidx2, align 4
  %add = add i32 %3, %2
  %arrayidx4 = getelementptr inbounds i32, i32* %c, i64 %idxprom8
  store i32 %add, i32* %arrayidx4, align 4
  ret void
}

declare spir_func i64 @_Z12get_local_idj(i32) #1
```

(c) OpenCL IR

Figure 2: Implementation of vecadd in NVVM/SPIR-V/OpenCL IR

high-level users do not care about low-level hardware; they can choose to use any programming language they like without worry about the reliability and portability of their programs. For hardware researchers, they can only focus on the optimization for the SPIR without worry about front-end language.

2.2.3 OpenCL IR. OpenCL (Open Computing Language) [12] is a framework for writing programs that execute across heterogeneous platforms. It supports several back-end devices: CPUs, GPUs, FPGAs, and so on. Although OpenCL can support using the native language (CUDA, HIP C++ etc.) to implement the kernel for each back-end device, it also provides OpenCL C programming language, a series of high-level APIs used for high-performance computing. This language provides a rich set of built-in functions for scalar and vector operations. These functions support scalar and vector argument types and can be used to implement high-performance programs for different back-end devices. When we refer to LLVM-IR, we generally refer to LLVM bc format of LLVM-IR.

2.3 Current Available Frames

2.3.1 OpenCL-SPIR-V translator. The Khronos Group developed a Bi-Directional translator[3] that supports converting between OpenCL IR and SPIR-V files. Although it generates SPIR-V, which can further be deployed and executed with RISC-V back-end devices, it only accepts OpenCL IR as input. In our project, we add some extensions based on the original translator to support its handling NVVM. As described in Sec. 2, NVVM is a subset of LLVM IR along with a defined set of the built-in functions. The translator cannot handle these NVVM-specific built-in functions. For example, in Fig. 2, (c) is an OpenCL IR and can be translated by the translator. However, (a) is an NVVM IR and has an NVVM-specific built-in function, $llvm.nvvm.read.ptx.srge.tid.x()$. When we pass this NVVM to the translator, it will raise an error, as it does not recognize this built-in function, but only recognize *get_local_id* in OpenCL IR.

2.3.2 POCL. POCL (Portable Computing Language) [5] is being developed as an efficient implementation of the OpenCL standard. POCL is a framework that accepts SPIR-V binary files for input. POCL will first convert SPIR-V into OpenCL IR, using the translator mentioned above, links the converted OpenCL IR with the runtime library, and finally emits the object file. POCL can support several back-end devices, like x86, CUDA, MIPS, etc. POCL is highly extensible; to support a new back-end device, researchers only need to provide a runtime API library for OpenCL IR, and this library will be used to link when emitting object files. Although original POCL does not support RISC-V, other researchers [3] have supported RISC-V on POCL and executed these file on several back-end devices. In our experiment, we use this version of POCL.[4]

2.3.3 Vortex. Vortex[5] [3] is an open-source RISC-V-based GPGPU processor. Vortex implements a SIMT architecture with a minimal ISA extension to RISC-V (*tmc*: activate threads, *wspan*: spawn a waive-front (or warp), *split/join*: divergent branch handling instructions, *bar* : stall waive-front). Currently, Vortex can execute OpenCL IR through POCL runtime systems. Thus, we can directly execute our generated object file on Vortex. We also use Vortx's RISC-V library in the linkage phase in POCL.

3 OVERVIEW OF THE PIPELINE

In this section, we show how to execute a simple vector add CUDA source code (Code. 1) on an RISC-V GPU.

3.1 Input CUDA source code

CUDA can be regarded as an extension of standard C++. The only difference is that CUDA has some other extra features, like definitions of memory hierarchy and some unique built-in functions. As it's quite easy to execute C++ on RISC-V back-end devices, the only part we need to handle is the extra part specific for CUDA. For example, in our example, only two CUDA specific features do not belong to standard C++: *__global__* and *threadIdx.x*. *__global__* is a CUDA C keyword that says the function should be called from the host. *threadIdx.x* can be regarded as a built-in function that records the work-item's local index in x-dim.

Code 1: VectorAdd CUDA source code

```
1  __global__ void vecadd (int *a, int *b, int *c) {
2    int gid = threadIdx.x;
3    c[gid] = a[gid] + b[gid];
4  }
```

[3]https://github.com/KhronosGroup/SPIRV-LLVM-Translator

[4]https://github.com/vortexgpgpu/pocl

[5]https://vortex.cc.gatech.edu/

function name	detail
llvm.nvvm.read.ptx.sreg.ctaid	get the block index
llvm.nvvm.read.ptx.sreg.ntid	get the block dimension
llvm.nvvm.read.ptx.sreg.tid	get the thread index
llvm.nvvm.barrie	synchronize threads within a block
llvm.sqrt	calculte the square root
llvm.fabs	calculate the absoulte value
llvm.nvvm.d2i	narrowing convertions
llvm.fma	fused multiply–add

Table 1: Built-in functions that are widely used in high-performance computing programs.

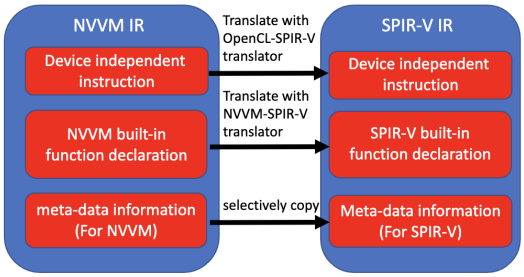

Figure 3: An overview of our translator.

3.2 Compile to NVVM

Our pipeline will directly use the CUDA toolkit to compile CUDA source code into NVVM IRs. Thus, we can get the following NVVM IR (Code. 2)(For clarity, we remove some unimportant content):

Code 2: NVVM VectorAdd IR

```
1  target triple = "nvptx64-nvidia-cuda"
2
3  define dso_local void @vecadd(
4      i32* nocapture readonly %a,
5      i32* nocapture readonly %b,
6      i32* nocapture %c) {
7  entry:
8    ; see Fig. 2(a) for deatil code
9  }
10 declare i32 @llvm.nvvm.read.ptx.sreg.tid.x()
11
12 !nvvm.annotations = !{!3}
13 !3 = !{void (i32*, i32*, i32*)* @vecadd,
14         !"kernel", i32 1}
```

At the beginning, the IR has a meta-data variable *target triple*. This meta-data records the target back-end device, which is used for further code generation to emit object files. For NVVM, it will always have value "$nvptx64 - nvidia - cuda$".

nvvm.annotations is another important meta-data. It records the kernel function of this program. As mentioned in Sec. 3.1, CUDA has a special keyword, *__global__*, to mark a function as a kernel function. As we cannot use this keyword in NVVM, we have to use an extra meta-data to record this information.

This NVVM IR has a function *llvm.nvvm.read.ptx.srge.tid.x*, which is declared but not defined. This is a built-in function that will be linked with CUDA libraries. As these libraries can only be used for NVIDIA GPUs, we have to replace these built-in functions in the next phase. In Table. 1, we show several built-in functions that are widely used in high-performance computing programs.

3.3 Translate NVVM to SPIR-V

In Fig. 3, we show an overview of our translator. The input NVVM IR comprises three components: metadata (e.g. device version, function property), NVVM built-in function declaration (e.g. functions for getting thread index), and device-independent instruction.

3.3.1 Meta-data. Meta-data is used to record information like function name, data layout, and address length. Most meta-data are independent with back-end devices; thus we can directly copy them. Thus, in our vecadd example, we remove all meta-data except these back-end related meta-data. For a target triple, we have to modify it to the corresponding SPIR-V triple. As for some CUDA-specific meta-data, we have to either directly drop it or translate it to corresponding SPIR-V meta-data. In NVVM, *nvvm.annotations* is used to denote the kernel function for the whole program, which is marked with *__global__* in CUDA source code. However, for SPIR-V, it uses a different method to record the kernel functions: these functions that have meta-data *kernel_arg_type*. Thus, for each given NVVM, we have to record the functions recorded by *nvvm.annotations*, remove this meta-data, and add a new meta-data *kernel_arg_type* for these functions. Finally, some meta-data are not needed in NVVM due to some CUDA's hypotheses. However, these hypotheses do not exist in SPIR-V. Thus, we have to add these meta-data explicitly. For example, in CUDA, when the input variables are pointer, they should always point to the global memory. So there is no meta-data to represent the memory hierarchy for each input pointer. While SPIR-V does not have this limitation, we have to add an extra meta-data *kernel_arg_addr_space* to record this information, to tell the compile and back-end devices that these pointers are all point to the address in global memory.

3.3.2 Built-in function. Built-in functions are functions that are special for the given back-end devices or framework. They are the key for each device, as they can be regarded as primitives of frameworks. Although for the Single-Instruction-Multiple-Data protocol, all devices have the same primitives: for getting threads' indexes, getting blocks' indexes, getting the length of each dimension, and so on. They have different built-in functions to implement these primitives.

For example, in NVVM, programs directly call a built-in function *llvm.nvvm.read.ptx.srge.tid.x* to get the global thread index in x-dimension for a thread. SPIR-V uses a different method; in SPIR-V, a variable *GlobalInvocationId* records a thread's index in different dimensions. To get the index, we have to first load this variable from memory and then extract the index for a special dimension.

3.3.3 Device independent instruction. In the above two sections, we modify contents that are different between NVVM and OpenCL IR, which cannot be translated by the existing OpenCL-SPIR-V translator. However, usually only a small part of a program belongs to these two classes, and most instructions are device independent. For these instructions, we can directly use the existing OpenCL-SPIR-V translator. For example, in Fig. 4, we have four instructions in NVVM IR. Only the first instruction (shown with red text) is

NVVM-specific and has to be translated with the NVVM-SPIR-V translator. The rest instructions (shown with green text) are device independent can be translated with the existing OpenCL-SPIR-V translator.

Separately handling device-dependent/independent instructions can avoid duplication of workload, as we do not implement the workload already existing in the OpenCL-SPIR-V translator when we develop the NVVM-SPIR-V translator. We show a diagram of the handling instructions in NVVM IR in Fig. 5. After translating, we can get SPIR-V IR shown in Fig. 2(b).

3.4 Translate SPIR-V to OpenCL IR

We need this phase for two reasons: 1) we want to execute SPIR-V on Vortex, and Vortex only accept the OpenCL IR as input; 2) SPIR-V is not human-readable; in other words, it's hard to debug. Thus, we add this phase to translate SPIR-V to OpenCL IR. In this phase, we directly invoke the LLVM-SPIRV translator, described in Sec. 2.3.1. After this step, we will get the OpenCL IR shown in Fig. 2(c).

3.5 Execute OpenCL IR with Vortex

The final step is to execute OpenCL IR with Vortex. For this phase, we have a customized host code (host code is running on x86 while Kernel programs are running on RISC-V or extended RISC-V) to prepare for input data, set the arguments' types for the kernel function, allocate the memory buffer for the input and output, invoke Vortex's corresponding kernel launch function, and finally get the results and verify them.

4 DETAILED DESCRIPTION AND EXPERIMENTS

In this section, we describe the detailed information about what we need to do to support executing several CUDA features.

4.1 Support built-in functions

4.1.1 Grid/Block information. One of the most important features for CUDA is that it is SIMD.The key to implementing SIMD is to assign a different index for multiple threads, using a unique API. Although we have described how to support getting *threadIdx.x* in CUDA on RISC-V, some detailed information still needs to explicitly be discussed. NVVM will invoke different functions (Code. 3) to get the index for different dimensions.

Code 3: NVVM built-in function for getting index

```
; get block index, from x-z
@llvm.nvvm.read.ptx.sreg.ctaid.x()
@llvm.nvvm.read.ptx.sreg.ctaid.y()
@llvm.nvvm.read.ptx.sreg.ctaid.z()
; get thread index, from x-z
@llvm.nvvm.read.ptx.sreg.tid.x()
@llvm.nvvm.read.ptx.sreg.tid.y()
@llvm.nvvm.read.ptx.sreg.tid.z()
```

While SPIR-V will first load a global variable from memory and then extract different position for different dimensions, as shown in Code. 4.

Code 4: OpenCL built-in function for getting index

```
; get block index, from x-z
variable = load BuiltInWorkgroupId;
extract global_variable, 0;
extract global_variable, 1;
extract global_variable, 2;
; get thread index, from x-z
variable = load BuiltInLocalInvocationId;
extract global_variable, 0;
extract global_variable, 1;
extract global_variable, 2;
```

To translate from an NVVM built-in function to SPIR-V, we have to replace the *@llvm.nvvm.read.ptx.sreg.ctaid.x*()'s call instruction with two consecutive instructions: load instruction and extract instruction. Besides, we have to analyze the function name of the call instruction to get the dimension the instruction needs and forward this dimension as an argument for the extract instruction. If this function name ends with *x*, it is for dimension 0. While function name ends with *y* or *z* for dimension 1 or 2.

4.1.2 barrier. NVVM has a simple API (Code. 5) to implement synchronization among all threads within a block.

Code 5: NVVM built-in function for synchronization

```
call void @llvm.nvvm.barrier0()
```

OpenCL's synchronization function, *barrier*, has a different prototype; it requests an argument for the memory address space needed to synchronize: *CLK_LOCAL_MEM_FENCE* for local memory and *CLK_GLOBAL_MEM_FENCE* for global memory. In fact, OpenCL's *barrier* is a function to ensure the correct ordering of memory operations to global/local memory, not an actual synchronization barrier as *barrier*0 in NVVM. For NVVM, it *barrier*0 is used to synchronize all threads within a block to a same line. As the threads within a block can visit both global memory and local memory (share memory), we have to make sure they have the same order for these two memories. Thus, we translate the NVVM barrier to OpenCL's *barrier* with a parameter to ensure the ordering for both local and global memory, as shown in Code. 6.

Code 6: OpenCL built-in function for synchronization

```
call void @barrier(i32
    CLK_LOCAL_MEM_FENCE|CLK_GLOBAL_MEM_FENCE)
```

4.1.3 Atomic. Several atomic operations are provided by CUDA, suck as *atomicAdd*, *atomicSub*, *atomicExch*, and so on. In NVVM, all these operations will be presented by the *atomicrmw* instruction with different operations for add, sub, exchange, etc. (Code. 7).

Code 7: NVVM atomic operations

```
;atomicAdd(&data[0], 1);
%0 = atomicrmw add i32* %data, i32 1 seq_cst
;atomicSub(&data[0], -1);
%1 = atomicrmw add i32* %data, i32 -1 seq_cst
;atomicExch(&data[0], 1);
%2 = atomicrmw xchg i32* %data, i32 1 seq_cst
```

In OpenCL, these atomic operations are regarded as normal function calls (Code. 8). To translate from NVVM to OpenCL IR, our pipeline needs to extract the operation in each *atomicrmw* instruction, and use this operation to choose the corresponding

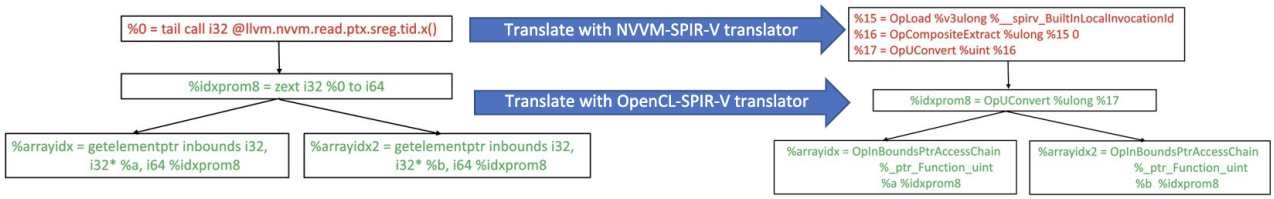

Figure 4: For NVVM specific built-in function, we have to handle the corresponding instructions (red text) with NVVM-SPIR-V translator. While for device independent instructions (green text), we can directly translate them with existing OpenCL-SPIR-V translator

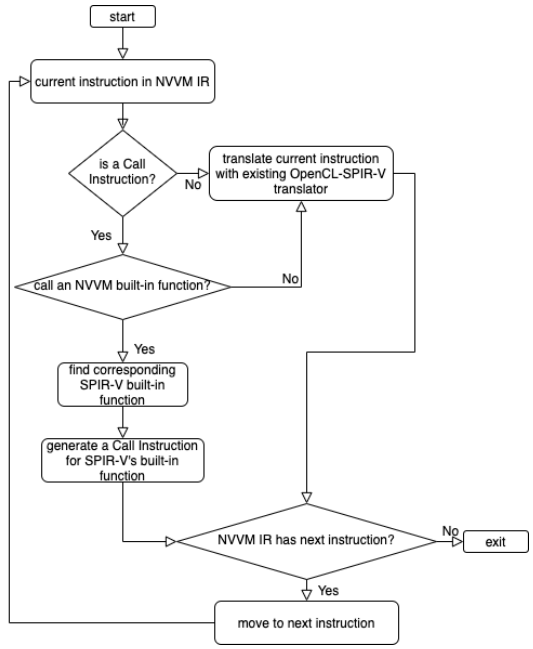

Figure 5: Our translators will reuse the OpenCL-SPIR-V translator, except when a instruction calls NVVM built-in function.

OpenCL's function name. After that, it directly copies the parameters in NVVM's instruction to OpenCL's instruction, as they have the same argument prototype (first argument: pointer, second argument: int32).

Code 8: OpenCL atomic functions

```
%0 = call i32 @atomic_add(i32* %data, i32 1)
%1 = call i32 @atomic_add(i32* %data, i32 -1)
%2 = call i32 @atomic_xchg(i32* %data, i32 1)
```

4.2 Benchmark Experiments

We also try to translate benchmarks. In Table. 2, we record the translating results for applications in Rodinia[2]. After we support features for grid/block information, barrier, and atomic instructions, we can succeed in translating most applications. However, there are still some applications we have not yet supported. These applications use either texture or some mathematical functions.

application	feature	support?
b+tree	-	yes
bfs	-	yes
cfd	double3 type	yes
huffman	atomic	yes
pathfinder	memory hierachy	yes
gaussian	-	yes
hotspot	-	yes
hotspot3D	-	yes
lud	memory hierachy	yes
nw	-	yes
streamcluster	-	yes
particlefilter	d2i	on going
backprop	__log2f	on going
lavaMD	d2i	on going
kmeans	texture	no
hybrid sort	texture	no
leukocyte	texture	no

Table 2: Translating applications in Rodinia benchmark

5 CONCLUSION

We have demonstrated a way to execute CUDA source code on an RISC-V back-end devices. To validate the feasibility, we build a pipeline that can succeed in executing multiple CUDA source codes with multiple features, including multi-thread,multi-block, atomic, and synchronization. Our pipeline comprises four steps: compiles CUDA source code into NVVM, translates NVVM and SPIR-V, uses modified POCL to emit object file, and finally, executes the generated object file on an open-source RISC-V GPU architecture. Except for the CUDA toolkit, which is required to compile NVVM, all other components are open-source and can be easily found in Github.

We also build a translator that supports translating NVVM into SPIR-V. This translator is lightweight and only relies on LLVM. It can be executed without the CUDA toolkit and GPUs. Our experiment results show that our translator can support most applications in Rodinia. In the future, we will try to support the remaining applications. In detail, we will support texture memory and mathematical functions, not only to convert from NVVM to SPIR-V, but also to support these corresponding libraries which will be needed when executing our generated SPIR-V on RISC-V GPUs.

REFERENCES

[1] Michal Babej and Pekka Jääskeläinen. 2020. HIPCL: Tool for Porting CUDA Applications to Advanced OpenCL Platforms Through HIP. In *Proceedings of the International Workshop on OpenCL*. 1–3.

[2] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE international symposium on workload characterization (IISWC)*. Ieee, 44–54.

[3] Fares Elsabbagh, Blaise Tine, Priyadarshini Roshan, Ethan Lyons, Euna Kim, Da Eun Shim, Lingjun Zhu, Sung Kyu Lim, et al. 2020. Vortex: OpenCL Compatible RISC-V GPGPU. *arXiv preprint arXiv:2002.12151* (2020).

[4] Vinod Grover and Yuan Lin. 2012. Compiling CUDA and other languages for GPUs. In *GPU Technology Conference (GTC)*.

[5] Pekka Jääskeläinen, Carlos Sánchez de La Lama, Erik Schnetter, Kalle Raiskila, Jarmo Takala, and Heikki Berg. 2015. pocl: A performance-portable OpenCL implementation. *International Journal of Parallel Programming* 43, 5 (2015), 752–785.

[6] Ronan Keryell, Ruyman Reyes, and Lee Howes. 2015. Khronos SYCL for OpenCL: a tutorial. In *Proceedings of the 3rd International Workshop on OpenCL*. 1–1.

[7] John Kessenich, Boaz Ouriel, and Raun Krisch. 2018. SPIR-V Specification. *Khronos Group* 3 (2018).

[8] Kostas Kontogiannis, Johannes Martin, Kenny Wong, Richard Gregory, Hausi Müller, and John Mylopoulos. 2010. Code migration through transformations: An experience report. In *CASCON First Decade High Impact Papers*. 201–213.

[9] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems* 25 (2012), 1097–1105.

[10] Evgeny Kuznetsov and Vladimir Stegailov. 2019. Porting CUDA-Based Molecular Dynamics Algorithms to AMD ROCm Platform Using HIP Framework: Performance Analysis. In *Russian Supercomputing Days*. Springer, 121–130.

[11] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 75–86.

[12] Aaftab Munshi. 2009. The opencl specification. In *2009 IEEE Hot Chips 21 Symposium (HCS)*. IEEE, 1–314.

[13] Trong Duc Nguyen, Anh Tuan Nguyen, and Tien N Nguyen. 2016. Mapping API elements for code migration with vector representations. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 756–758.

[14] Hugh Perkins. 2017. CUDA-on-CL: a compiler and runtime for running NVIDIA® CUDA™ C++ 11 applications on OpenCL™ 1.2 Devices. In *Proceedings of the 5th International Workshop on OpenCL*. 1–4.

[15] André Silveira, Rafael Bohrer Avila, Marcos E Barreto, and Philippe Olivier Alexandre Navaux. 2000. DPC++: Object-Oriented Programming Applied to Cluster Computing.. In *PDPTA*.