A Case Study: Exploiting Neural Machine Translation to Translate CUDA to OpenCL

Yonghae Kim
School of Computer Science
Georgia Institute of Technology
Atlanta, USA
yonghae@gatech.edu

Hyesoon Kim
School of Computer Science
Georgia Institute of Technology
Atlanta, USA
hyesoon@cc.gatech.edu

***Abstract*—As hardware complexity increases, the need of hardware generation from high-level languages has arisen. To describe the behavior of hardware using high-level languages, high-level synthesis (HLS) and domain-specific languages (DSLs) have been developed. However, due to the nature of domain specific languages, there are many number of programming languages and tool chains, which all require a big learning curve in learning them and writing code.**

To tackle the challenge, we propose source-to-source translation using neural machine translation (NMT) techniques to allow software/hardware engineers to easily interchange different frames. The main benefit of NMT is that if it can successfully translate one program language (PL) to another program language, it can do the same for other program languages. To test out whether NMT can learn source-to-source PL translation, we evaluate it with CUDA to OpenCL, which both have very similar programming styles. Our work shows (i) a training input set generation method, (ii) pre/post processing, and (iii) a case study using Polybench-gpu-1.0, NVIDIA SDK, and Rodinia benchmarks.

***Index Terms*—CUDA, OpenCL, Program Translator, Neural Machine Translation**

I. Introduction

For decades, software and hardware engineers have desired hardware generation using high-level languages. This is because hardware-descriptive languages (HDLs) require a big learning curve for engineers who are not familiar with those languages. Considering increasing design complexity and the high productivity required in the rapidly changing market, enabling the development of hardware design using high-level languages is essential since it enables reduced engineering effort and higher productivity.

To reduce the development effort, several high-level synthesis (HLS) and domain-specific languages (DSLs) such as Chisel [2], Bluespec [11], and SystemC [13] have been developed. However, they tend to target some specific applications, and a number of languages and tool chains still require a huge amount effort to learn and write code. In addition, developing and maintaining those infrastructures demands lots of manpower.

To address the challenge, we propose source-to-source translation using neural machine translation (NMT). With the recent development, NMT becomes an attractive option for program language translation [3], [5], [6], [16]. Especially, the language agnostic neural network design in sequence-to-sequence (seq2seq) models [9] is a promising method, as it is currently used in Google. The seq2Seq model can be trained using only a pair of in/output without considering the language grammar difference, which makes the same network applicable for different natural language translation.

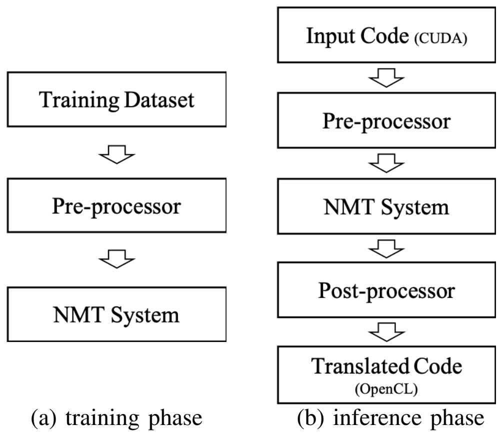


Fig. 1: Overview of workflow [8].

In this work, we present a case study of source-to-source translation using NMT techniques by translating CUDA to OpenCL. CUDA and OpenCL are both parallel computing programming languages for accelerators. We chose these languages for two reasons; (1) CUDA and OpenCL share many similarities, so they provide a good platform to develop the techniques of PLNMT. Based on the knowledge/techniques from this translation, we will expand our work to other program language translation. (2) OpenCL is one of the HLS frameworks.

Please note that our contribution is not proposing CUDA to OpenCL translation. There have already been several tools that do that such as SnuCL [7], CU2CL [10], and HIP in AMD [1]. Instead, our contribution is to explore the possibility of whether NMT can be used for program source-to-source translation by just learning from end-to-end training examples. Especially, we are focusing on PL that can be used for hardware design.

TABLE I: Comparison between natural languages and programming languages.

	Natural Language	Programming Language
Token	Punctuation marks	Punctuation marks, operators, variables
Scope in sentence	Vague	Explicit
Operator precedence	None	Exist
Ambiguous	Allowed	Not allowed
Naming Scope	weak	Explicit
Rigid syntax	Exist but flexible in context	Exist
Number of names	Huge but finite	Arbitrary

The summary of our contribution is as follows. We develop a dataset generation flow for translating CUDA to OpenCL using NMT. To do so, we first write a pair of API usages for CUDA and OpenCL. With the API usages written, we construct usage symbol trees to extract sentences from the CUDA samples and find uncovered API usages. Using the dataset we generated, we train the NMT system, which learns the structural similarity between CUDA and OpenCL, and translate CUDA code to OpenCL code. Finally, we present translation examples and discuss the limitations and feasibility of our current approach.

II. Background

A. Neural Machine Translation (NMT)

Neural networks have demonstrated outstanding performance in natural language processing (NLP). For example, the sequence-to-sequence (seq2seq) model [15] presents a large, deep Long Short-Term Memory (LSTM), outperforming a mature SMT system by a sizable margin. It also shows the capability of translating very long sentences.

Motivated by the meaningful success and the similarities between natural and programming languages, we exploit the seq2seq model to translate programming languages. We take a statement (or statements) as a sentence and translate it to another statement written in the target language. Since we translate CUDA to OpenCL, in a training dataset, CUDA code becomes source sentences, and OpenCL code becomes target sentences. During an inference phase, we take as input CUDA code and infer OpenCL code.

B. CUDA vs. OpenCL

Both CUDA and OpenCL have host and kernel code. In the host code case, most host API functions have one-to-one correspondence between CUDA and OpenCL. Consider an example of `cudaMalloc` and `clCreateBuffer` that have the same meaning between CUDA and OpenCL. Since a function call of `cudaMalloc` contains all the necessary information to be translated, such as arguments, we can write a function call of `clCreateBuffer` with a given source code.

Kernel code also has similarities between CUDA and OpenCL. Kernel qualifiers and built-in functions have equivalents, and therefore we replace one program language's keyword with an equivalent one in the other language. Program translation rules between CUDA and OpenCL have already been studied [10], [7], and we use these rules to train our NMT system.

TABLE II: Mapping tokens to abstract symbols.

Symbol	Token
_id	identifiers, string literals, numeric constants
_op	operators
_tp	data types

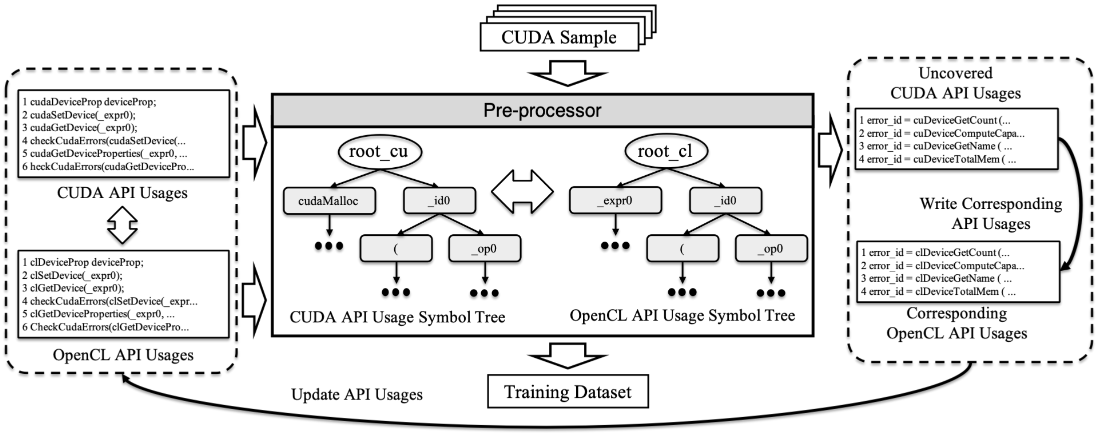
III. Overview of Workflow

In this section, we describe the overview of the NMT workflow. As can be seen in Fig. 1, in addition to an NMT system, we use a pre-processor during the training phase and a pre-/post-processor during the inference phase. Compared to natural languages in which a large, but finite, set of vocabulary exists, programmers use numerous arbitrary variable names such as alphabet characters or abbreviations of variables/functions. By having a pre-/post-processor, we enable arbitrary variable names in programming languages to be translated. This also contributes to minimizing the size of vocabulary. Both pre-/post-processors are developed as Python scripts.

A. Pre-processor

A pre-processor performs lexical analyses and variable renaming. First, it reads and tokenizes a given program code. While tokenizing the code, it merges tokens that comprise one statement as a sentence—i.e., even when a statement is written in multiple lines, it puts the tokens together as a sentence. In this way, the pre-processor generates a set of sentences, each of which consists of tokens. Next, we map tokens (from tokenized sentences) to three types of symbols depending on their variable type. This is because we try to reduce the cases in which different sentences are renamed to one sentence. As can be seen in Table II, we map identifier, string literals, and numeric constants to *_id*, and each symbol is tagged as a number in ascending order starting from zero—i.e., the first token mapped to *_id* becomes *_id0*, and the next one becomes *_id1*. And, we map operators to *_op* and data types to *_tp* with a tagged number in ascending order starting from zero. The numbering for each type has its own order.

Moreover, pre-processing sentences creates a mapping table that contains mapping information between variable names and abstract symbols. This is used later by a post-processor when it replaces the renamed tokens with their original names. Note that we do not rename CUDA/OpenCL APIs since they decide the context of a sentence and which rule to be used to translate it. Finally, the pre-processor is also capable of generating a



```
1 cudaDeviceProp deviceProp;
2 cudaSetDevice(_expr0);
3 cudaGetDevice(_expr0);
4 checkCudaErrors(cudaSetDevice(...
5 cudaGetDeviceProperties(_expr0, ...
6 heckCudaErrors(cudaGetDevicePro...
```

```
1 clDeviceProp deviceProp;
2 clSetDevice(_expr0);
3 clGetDevice(_expr0);
4 checkCudaErrors(clSetDevice(_expr...
5 clGetDeviceProperties(_expr0, ...
6 CheckCudaErrors(clGetDevicePro...
```

```
1 error_id = cuDeviceGetCount (...
2 error_id = cuDeviceComputeCapa...
3 error_id = cuDeviceGetName ( ...
4 error_id = cuDeviceTotalMem ( ...
```

```
1 error_id = clDeviceGetCount (...
2 error_id = clDeviceComputeCapa...
3 error_id = clDeviceGetName ( ...
4 error_id = clDeviceTotalMem ( ...
```

Fig. 2: Overview of the proposed dataset generation flow.

training dataset. The details of how we generate a dataset are covered in Section IV.

B. NMT System

In an NMT system, we exploit a seq2seq model as it has demonstrated outstanding performance in translating long sentences. The NMT system is a component where actual translation occurs. During a training phase, a training dataset, which consists of source sentences for CUDA and target sentences for OpenCL, is pre-processed and fed into the NMT system. During an inference phase, input CUDA code is pre-processed and fed into the NMT system. Then, the NMT system outputs renamed OpenCL code.

C. Post-processor

A post-processor performs initial name replacement and code restructure. As we train the NMT system with renamed sentences, the output sentences generated from the NMT system also have abstract symbols. Based on the mapping table created by a pre-processor, a post-processor replaces them with their original names. Since the initial variable names from the input code remain in the output code, the translation provides a high-quality code. Finally, based on syntactic rules, it puts appropriate indents between tokens for better readability and generates the final outcome.

IV. Dataset Generation

Since there is no publicly available dataset for translating CUDA to OpenCL using NMT, we develop a dataset generation flow and generate a dataset from CUDA samples. Fig. 2 shows the overview of our proposed dataset generation flow, and we explain the steps of dataset generation as below.

A. Steps of Dataset Generation

We first summarize how we generate a dataset and describe additional details in the following subsections.

1) Write API usages

- Write a pair of API usages for CUDA and OpenCL.

2) Build usage symbol trees

- Read API usages written.
- Tokenize each sentence and rename tokens as abstract symbols.
- Build usage symbol trees that consist of renamed tokens.

3) Gather expressions and find uncovered API usages from CUDA samples

- Tokenize each sentence in CUDA samples.
- If a sentence includes CUDA APIs, see if the sentence is found in the CUDA usage symbol tree.
- If found, and the API usage has expression nodes, add each expression to a corresponding expression node.
- If not found, write the sentence to a separate file that contains uncovered API usages.

4) Generate a dataset

- Read again API usages written.
- If an API usage has expression keywords, permutate the expression nodes and add multiple sentences to a dataset.
- If not, add sentences to a dataset without permutation.

B. API Usage Generation

Our method requires users to manually write a pair of API usages for CUDA and OpenCL. A CUDA API usage has a sentence pattern to translate, and any variable names can be used in the sentence pattern since they will be renamed as abstract symbols by a pre-processor. An OpenCL API usage has a sentence pattern that the corresponding CUDA API usage should be translated to. Each line of API usages has a one-to-one correspondence with each other—i.e. the i^{th} line in the set of CUDA API usages corresponds to the i^{th} line in the set of OpenCL API usages.

C. Building a Usage Symbol Tree

A pre-processor tokenizes API usages that are manually written and renames tokens as abstract symbols. Then, it builds usage symbol trees that consist of renamed tokens for each CUDA and OpenCL. By traversing the usage symbol trees with tokens of a sentence, we can easily determine whether a given sentence is covered by our API usages, and based on the outcome, we can generate a corresponding OpenCL sentence.

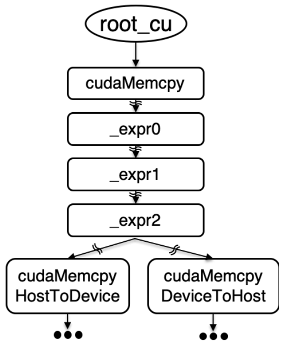


Fig. 3: Node representation of `cudaMemcpy` usage.

```
cudaMemcpy(A_gpu, A, sizeof(double)*NI, cudaMemcpy-HostToDevice);
cudaMemcpy(B_gpu, B, sizeof(double)*NI*NL, cudaMem-cpyHostToDevice);
cudaMemcpy(C_gpu, C, sizeof(double)*NI*NJ*NK, cud-aMemcpyHostToDevice);
```

(a) Three different `cudaMemcpy` sentences

```
cudaMemcpy(_expr0, _expr1, _expr2, cudaMemcpyHost-ToDevice);
```

(b) `cudaMemcpy` usage with expression keywords

Fig. 4: Example of using expression keywords.

If a sentence includes CUDA APIs but is not found in the usage symbol tree, it is considered an uncovered API usage and written to a separate file. The usage symbol trees enable us to maintain sentence patterns to translate and easily generate a new larger dataset when we get new CUDA samples.

D. Using an Expression Keyword

When we write a pair of API usages, we use an expression keyword, *_expr*, to represent function parameters. Each one is tagged as a number in ascending order starting from zero. Each expression keyword becomes an expression node in a usage symbol tree. Fig. 3 shows the example of node expression using expression keywords. This reduces manual efforts to write API usages. Consider the example of `cudaMemcpy`. It takes four parameters in its function call, of which each can have various shapes. Fig 4 (a) shows three different sentences. For these sentences, instead of writing three CUDA API usages for each sentence, we write one CUDA API usage with expression keywords, as shown in Fig 4 (b).

E. Expression Node Permutation

Expression node permutation is used to increase the size of a dataset. As explained in Section IV, when we define a pair of API usages, we use an expression keyword. Each expression keyword becomes an expression node in a usage symbol tree. When we generate sentences from CUDA samples, instead of simply finding sentences covered by API usages and adding them to a dataset, we collect expressions for each corresponding expression node. After looking through all samples, we permutate each expression node and generate the increased number of sentences. Consider `cudaMemcpy` in Fig. 3 and assume that we collect n expressions for *_expr0*, m expressions for *_expr1*, and k expressions for *_expr2*. By permutating each expression node, we produce $n \times m \times k$ sentences. Note that *_expr* symbols are not present in sentences fed into the NMT model. It is only used to generate a dataset and is not used as a vocabulary.

TABLE III: The number of sentences generated from CUDA benchmarks.

Benchmark	# application	# sentences found	# sentences generated
Polybench-gpu	15	169	221
NVIDIA SDK	25	265	583
Rodinia	13	286	538
Total	53	715	1874

TABLE IV: Hyper-parameters of the NMT system [9].

	Seq2Seq model
Batch size	128
Number of RNN layers	43
RNN cell	LSTM
Initial learning rate	0.005
Dropout rate	0.2
Attention model	scaled luong

V. Results

As discussed in Section IV, we extract sentences from CUDA samples and generate a dataset. Our current dataset has 126 lines of API usages, and we use Polybench-gpu-1.0 [14], NVIDIA SDK [12], and Rodinia [4] benchmarks as target translation. We currently support a subset of API usages that exist in the evaluated benchmarks. Table III presents the number of sentences generated. In the table, Column 2 indicates the number of sentences that include CUDA APIs and are found in our current usage symbol trees, and Column 3 indicates the number of sentences that we generate using the expression node permutation method. With only 126 lines of API usages, we generate 1874 sentences from CUDA samples. We can see that the permutation method increases the number of sentences by 2.6x.

We use the seq2seq model as our NMT model. Table IV shows the hyper-parameters of the NMT system used. We use the same dataset as training, development, and test dataset. This is because we intend to have the NMT system to produce a correct sentence and understand the limitation of this NMT-based translation approach. In contrast to natural languages, programming languages have a rigid syntax, and therefore we need to generate correct sentences to make the translated code executable. Therefore, we intentionally cause overfitting by using a shared dataset. While we achieve a 99.1 BLEU score, this does not imply that the NMT system has a better capability to infer and translate unseen sentences. However, it can correctly translate most of the sentences covered by the API usages written by users. For the polybench-gpu-1.0 benchmark, we manually changed about 10 lines out of 200-

350 lines for each application and were able to make it run. To fully utilize the potential of using machine learning, we would need a larger dataset and need to split the dataset so that there is no overlap among the training, development, and test dataset. We leave this as future work.

Fig 5 shows a translation example of a 2-D matrix multiplication. Fig. 5 (a) and (b) show the CUDA host and kernel code, respectively. Before being fed into the NMT system, they are pre-processed by a pre-processor; Fig. 5 (c) and (d) show the pre-processed code. If a sentence does not have CUDA APIs, it becomes `_line_not_to_translate` symbol and is replaced with the original sentences later by a post-processor. After translation using the NMT system, the pre-processed code is translated to OpenCL code that retains the renamed tokens, which can be seen in Fig. 5 (e) and (f). Finally, a post-processor replaces the renamed tokens with their initial names and provides the final OpenCL code. The final output code is shown in Fig. 5 (g) and (h).

VI. Limitation

In this section, we discuss the limitation of our current approach. Fig 6 presents an example of sentences incorrectly translated.

Long sentence translation Despite the LSTM's ability to learn long-range temporal dependencies, we observe several sentences in which some last words are truncated. In our translation, a sentence fed into the NMT system might be very long as we tokenize sentences in CUDA samples and each token becomes a word. Since many of the tokens are from function parameters, we believe a new embedding layer structure needs to be developed to encode parameter parts in a different way. We leave this as future work.

Unseen sentence translation Since we manually write API usages, the dataset generated inevitably has limited function coverage and does not guarantee the successful translation of random programs. Although variable renaming and the use of an expression keyword contribute to increasing its coverage with the limited number of API usages, it is still non-trivial to correctly translate a sentence not covered by API usages.

Manual steps for API function mapping We require manual efforts to write a pair of API usages. Considering that most host API functions have one-to-one correspondence between CUDA and OpenCL, most parts of the job in writing API usages would be changing the function name and the positions of parameters. However, since there are various sentence patterns, we still require efforts to write API usages for those sentences.

VII. Related Work

Using NMT for program translation Chen et al. [5] propose a novel tree-to-tree neural network and demonstrates higher accuracy for program translation, but it has a limited set of variables and restricts the vocabulary size. Zuo et al. [16] utilize NMT techniques to deal with a cross-architecture code similarity comparison. However, it does not handle high-level language translation.

```
void mm2Cuda(float* A, float* B, float* C) {
float *A_gpu;
...
cudaMalloc((void **)&A_gpu, sizeof(float) * NI * NK);
cudaMemcpy(A_gpu, A, sizeof(float) * NI * NK, cudaMem-
cpyHostToDevice);
...
```

(a) CUDA host code

```
__global__ void mm2_kernel1(float *A, float *B, float *C) {
int j = blockIdx.x * blockDim.x + threadIdx.x;
...
for (k = 0; k ¡ NK; k++) {
C[i * NJ + j] += A[i * NK + k] * B[k * NJ + j];
...
```

(b) CUDA kernel code

```
_line_not_to_translate
_line_not_to_translate
_tp0 _op0 _id0 ;
...
cudaMalloc ( ( _tp0 _op0 ) _op1 _id0 , sizeof ( _tp1 ) _op2
_id1 _op2 _id2 ) ;
cudaMemcpy ( _id0 , _id1 , sizeof ( _tp0 ) _op0 _id2 _op0
_id3 , cudaMemcpyHostToDevice ) ;
...
```

(c) Pre-processed CUDA host code

```
__global__ _tp0 _id0 ( _tp1 _op0 _id1 , _tp1 _op0 _id2 , _tp1
_op0 _id3 )
_line_not_to_translate
_tp0 _id0 _op0 blockIdx.x _op1 blockDim.x _op2 threadIdx.x
;
...
_line_not_to_translate
_line_not_to_translate
_line_not_to_translate
...
```

(d) Pre-processed CUDA kernel code

```
_line_not_to_translate
_line_not_to_translate
cl_mem _id0 ;
...
_id0 = clCreateBuffer ( context , CL_MEM_READ_WRITE ,
sizeof ( _tp1 ) _op2 _id1 _op2 _id2 , NULL , NULL ) ;
clEnqueueWriteBuffer ( command_queue , _id0 , CL_TRUE
, 0 , sizeof ( _tp0 ) _op0 _id2 _op0 _id3 , _id1 , 0 , NULL ,
NULL ) ;
...
```

(e) Host code translated by NMT system

```
__kernel _tp0 _id0 ( __global _tp1 _op0 _id1 , __global _tp1
_op0 _id2 , __global _tp1 _op0 _id3 )
_line_not_to_translate
_tp0 _id0 _op0 get_group_id ( 0 ) _op1 get_group_id ( 0 )
...
_line_not_to_translate
_line_not_to_translate
_line_not_to_translate
...
```

(f) Kernel code translated by NMT system

```
void mm2Cuda ( float * A , float * B , float * C ) {
cl_mem A_gpu ;
...
A_gpu=clCreateBuffer(context,   CL_MEM_READ_WRITE,
sizeof(float)*NI*NK, NULL, NULL);
clEnqueueWriteBuffer(command_queue,  A_gpu,  CL_TRUE,
0, sizeof(float)*NI*NK, A, 0, NULL, NULL);
...
```

(g) Post-processed OpenCL host code

```
__kernel void mm2_kernel1(__global float*A, __global float*
B, __global float* C) {
int j = get_group_id(0)*get_group_id(0)+get_local_id(0);
...
for ( k = 0 ; k ¡ NK ; k ++ ) {
C[i * NJ + j] += A[i * NK + k] * B[k * NJ + j] ;
...
```

(h) Post-processed OpenCL kernel code

Fig. 5: Translation example of 2-D matrix multiplication code [8].

Source	float* A_gpu; _br cudaMalloc((void **) & A_gpu, _expr0);
Translated	_expr0 = clCreateBuffer(context, CL_MEM_READ_WRITE, sizeof (DATA _TYPE)*_id8*_id6, NULL, NULL);
Expected	cl_mem A_gpu;
Source	cudaMalloc((void **) &data_gpu, sizeof(DATA_TYPE)*(M+1)*(N+1));
Translated	data_gpu=clCreateBuffer(context, CL_MEM _READ_WRITE, sizeof(DATA _TYPE)*_id7*_id6*_id0, NULL, NULL);
Expected	data_gpu=clCreateBuffer(context, CL_MEM_READ_WRITE, sizeof (DATA _TYPE)*(M+1)*(N+1), NULL, NULL);
Source	Convolution2D_kernel <<<grid, block >>>(A_gpu, B_gpu);
Translated	_clSetKernelArg(”Convolution2D_kernel” , 0, A_gpu); _clSetKernelArg (”Convolution2D_kernel”, 1, B_gpu); _clEnqueueNDRangeKernel(grid,
Expected	_clSetKernelArg(”Convolution2D_kernel”, 0, A_gpu); _clSetKernelArg(”Convolution2D _kernel”, 1, B_gpu); _clEnqueueNDRange Kernel(grid, block, ”Convolution2D_kernel”);
Source	cudaFree(A_gpu);
Translated	clGetDevice(A_gpu);
Expected	clReleaseMemObject(A_gpu);

Fig. 6: Example of sentences incorrectly translated.

CUDA to OpenCL translation Martinez et al. [10] translate CUDA to OpenCL at an AST level, and Kim et al. [7] use wrapper functions to translate between CUDA and OpenCL. Compared to those works, we use source-to-source translation and exploit NMT techniques to translate programming langauges.

VIII. Conclusion

In this work, we exploited NMT techniques to translate CUDA to OpenCL. To do so, we developed a dataset generation flow and generated a dataset from CUDA benchmarks. Moreover, for training and inference phases, the pre-/post-processor were developed to enable arbitrary variable names to be translated. While our current approach correctly translates most of the sentences covered by the API usages that are manually written, we discover several challenges of using NMT for program translation, especially for unseen or long sentences. We also note that the efforts of developing a training input set generator is not necessarily less than developing a source to source translator. In this work, the NMT itself has not been changed at all. It is our future work to improve the NMT to make it more program language translation friendly.

References

[1] AMD. Hip : Convert cuda to portable c++ code. https://github.com/ROCm-Developer-Tools/HIP, 2013.

[2] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Aviienis, J. Wawrzynek, and K. Asanovi. Chisel: Constructing hardware in a scala embedded language. In *DAC Design Automation Conference 2012*, pages 1212–1221, June 2012.

[3] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *CoRR*, abs/1409.0473, 2015.

[4] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54, Oct 2009.

[5] Xinyun Chen, Chang Liu, and Dawn Xiaodong Song. Tree-to-tree neural networks for program translation. In *NeurIPS*, 2018.

[6] Kyunghyun Cho, Bart van Merrienboer, aglar Gülehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. In *EMNLP*, 2014.

[7] J. Kim, T. T. Dao, J. Jung, J. Joo, and J. Lee. Bridging opencl and cuda: a comparative analysis and translation. In *SC ’15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, Nov 2015.

[8] Yonghae Kim and Hyesoon Kim. Translating cuda to opencl for hardware generation using neural machine translation. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO 2019, pages 285–286, Piscataway, NJ, USA, 2019. IEEE Press.

[9] Minh-Thang Luong, Eugene Brevdo, and Rui Zhao. Neural machine translation (seq2seq) tutorial. *https://github.com/tensorflow/nmt*, 2017.

[10] G. Martinez, M. Gardner, and W. Feng. Cu2cl: A cuda-to-opencl translator for multi- and many-core architectures. In *2011 IEEE 17th International Conference on Parallel and Distributed Systems*, pages 300–307, Dec 2011.

[11] R. Nikhil. Bluespec system verilog: efficient, correct rtl from high level specifications. In *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE ’04.*, pages 69–70, June 2004.

[12] NVIDIA. Nvidia. In *CUDA Samples Reference Manual*, Sept 2017.

[13] Preeti Ranjan Panda. Systemc: A modeling platform supporting multiple design abstractions. In *Proceedings of the 14th International Symposium on Systems Synthesis*, ISSS ’01, pages 75–80, New York, NY, USA, 2001. ACM.

[14] Robert Searles Sudhee Ayalasomayajula Scott Grauer-Gray, Lifan Xu and John Cavazos. Auto-tuning a high-level language targeted to gpu codes. In *Proceedings of Innovative Parallel Computing (InPar ’12)*, Mar 2012.

[15] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, NIPS’14, pages 3104–3112, Cambridge, MA, USA, 2014. MIT Press.

[16] Fei Zuo, Xiaopeng Li, Zhexin Zhang, Patrick Young, Lannan Luo, and Qiang Zeng. Neural machine translation inspired binary code similarity comparison beyond function pairs. *CoRR*, abs/1808.04706, 2018.