BabelTower: Learning to Auto-parallelized Program Translation

Yuanbo Wen [1 2 3] **Qi Guo** [2 4] **Qiang Fu** [1 2 3] **Xiaqing Li** [2] **Jianxing Xu** [1 2 3] **Yanlin Tang** [2 4] **Yongwei Zhao** [2 3] **Xing Hu** [2] **Zidong Du** [2 3] **Ling Li** [5] **Chao Wang** [1] **Xuehai Zhou** [1] **Yunji Chen** [2 4]

Abstract

GPUs have become the dominant computing platforms for many applications, while programming GPUs with the widely-used CUDA parallel programming model is difficult. As sequential C code is relatively easy to obtain either from legacy repositories or by manual implementation, automatically translating C to its parallel CUDA counterpart is promising to relieve the burden of GPU programming. However, because of huge differences between the sequential C and the parallel CUDA programming model, existing approaches fail to conduct the challenging *auto-parallelized program translation*. In this paper, we propose a learning-based framework, i.e., BabelTower, to address this problem. We first create a large-scale dataset consisting of compute-intensive function-level monolingual corpora. We further propose using back-translation with a discriminative reranker to cope with unpaired corpora and parallel semantic conversion. Experimental results show that BabelTower outperforms state-of-the-art by 1.79, 6.09, and 9.39 in terms of BLEU, CodeBLEU, and specifically designed ParaBLEU, respectively. The CUDA code generated by BabelTower attains a speedup of up to 347× over the sequential C code, and the developer productivity is improved by at most 3.8×.

1. Introduction

Massively parallel architectures such as GPUs have become the dominant computing platforms for a wide range of applications, such as graphical processing, high-performance computing (HPC), and machine learning. To exploit their highly parallel computation ability, practitioners typically leverage the CUDA parallel programming model (Sanders & Kandrot, 2010). However, writing parallel CUDA code is quite hard since the programmers should be aware of complicated architectural characteristics of GPUs, e.g., thread/block data organization, on-chip shared memory, and warp synchronization.

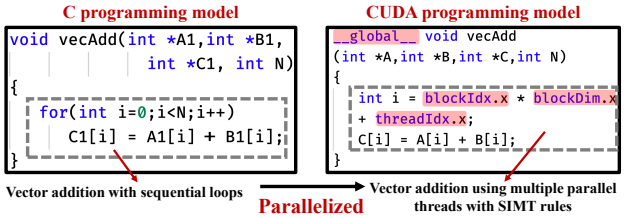

Figure 1. Both syntax and semantics are inherently different between sequential C and parallel CUDA programming model.

As sequential C code is relatively easy to obtain, it is very promising to alleviate the burden of GPU programming by automatically translating from C to CUDA. Concretely, there already exist vast repositories of legacy C code, which can be directly utilized or slightly modified for generating their parallel CUDA counterpart. Even if appropriate C code cannot be obtained from existing repositories, it is still much easier to write the sequential C code than the parallel CUDA code given the same functional specification.

However, such program translation is greatly challenging in practice. The main reason is that the CUDA programming model, which follows the SIMT (Single Instruction, Multiple Threads) model to partition the data into different parts with the same code, is quite different from conventional C programming in syntax and particularly semantics. Figure 1 shows an example of the key differences between the C and CUDA programming model, where the vector addition with sequential loops of C is parallelized using multiple indexed threads with religious SIMT rules of CUDA. Apparently, automating such process requires not only syntax-level translation (e.g., generating the CUDA-specific keywords such as threadIdx.x), but also loop detection, parallel semantic analysis, and sequential-to-parallel conversion. Essentially, this problem can be modeled as *auto-parallelized program translation*, which is harder than either traditional automatic

[1]University of Science and Technology of China [2]State Key Lab of Processors, Institute of Computing Technology, Chinese Academy of Sciences [3]Cambricon Technologies, Beijing, China [4]University of Chinese Academy of Sciences [5]Institute of Software, Chinese Academy of Sciences. Correspondence to: Yunji Chen <cyj@ict.ac.cn>.

Proceedings of the 39th International Conference on Machine Learning, Baltimore, Maryland, USA, PMLR 162, 2022.

parallelization or program translation tasks.

The difficulty of this problem makes existing approaches including *auto-parallelization approaches* (Verdoolaege et al., 2013; Nugteren & Corporaal, 2014; Mendonça et al., 2017) and *statistical program translation approaches* (Nguyen et al., 2015; Chen et al., 2018; Roziere et al., 2020; 2021a) unsurprisingly ineffective. The state-of-the-art auto-parallelization approaches employ code templates or polyhedral models for program transformation. However, these approaches either require considerable manual efforts for code annotation or confront scalability and generality problems. The statistical program translation approaches train either probabilistic models or neural networks, which are inspired by recent advances on statistical/neural machine translation, for end-to-end automatic code translation with large-scale datasets. However, such approaches fail to address our problem because of not addressing two challenges: 1) *scarcity of effective dataset*. There lacks a large-scale dataset due to huge efforts on data collection, cleaning, and labeling. Moreover, the amount of CUDA code is much less that that of C code, e.g., around four orders of magnitude in open-source repositories such as GitHub, and thus it is intractable to build a large number of well-paired corpora for training. 2) *lack of parallel semantics.* Existing statistical models fall short of taking the parallel semantics into consideration, which are decisive to detect loops from the sequential C code and then convert them to the parallel CUDA code, since it is awkward to add semantic features into such generative models (Shen et al., 2004).

Regarding the above challenges, in this paper, we propose a novel learning-based framework, i.e., BabelTower, for auto-parallelized program translation specifically designed to translate from sequential C to parallel CUDA. As the basis of training, we create a large-scale dataset consisting of $501,732$ C functions, $129,497$ CUDA functions, as well as C-CUDA function pairs for validation and test, all of which are compute-intensive to evaluate the effectiveness of parallel semantic conversion, mined from open-source repositories. To cope with unpaired corpora and parallel semantic conversion, we propose *using back-translation with a discriminative reranker*. Concretely, we first leverage the widely used data augmentation technique, i.e., back-translation, to enable unsupervised translation from C to CUDA based on large-scale unpaired monolingual corpora. Then, the parallel semantics are embedded into a discriminative model for selecting the best hypothesis within the n-best beam search candidates. Experimental results show that BabelTower outperforms state-of-the-art by 1.79, 6.09, and 9.39 in terms of BLEU (Papineni et al., 2002), CodeBLEU (Ren et al., 2020), and specifically designed ParaBLEU, respectively, and thus 92.8% generated CUDA code can be correctly compiled. We also demonstrate that the generated CUDA code of BabelTower attains a speedup of up to $347\times$ over the original sequential C code. Furthermore, BabelTower improves developer productivity of real-life CUDA programs by at most $3.8\times$ in our empirical study.

Our contributions are:

- We are the first to provide a publicly-available large-scale C-CUDA dataset, enabling advanced research on the important domain of auto-parallelized program translation.
- We are the first to introduce a learning method for translating from C to CUDA, which addresses the key challenges of unpaired corpora and parallel semantic conversion.
- We conduct thorough evaluation in terms of accuracy, functionality, performance, and productivity, which well demonstrates the benefits and potential of BabelTower.

2. Problem Statement

We consider the problem of translating the serial program (specifically, in the C language) into a parallel one (in the CUDA C++ language) given only monolingual corpora. One approach is to model the problem as a machine translation problem between programming languages, and thus existing unsupervised methods can be applied. However, the inherent differences between the serial and the parallel programming languages require the translating method to capture the *parallel semantics*, i.e., to auto-parallelize the semantics behind the serial program, rather than only performing translation between syntaxes. Thus, we model the problem as *Auto-Parallelized Program Translation* instead.

Definition 2.1 (Auto-Parallelized Program Translation)**.** There is a serial programming language $\mathbb{L}_{\mathrm{S}}$ and a parallel programming language $\mathbb{L}_{\mathrm{P}}$, each is an infinite set of valid program strings. There exists a binary relation $\leftrightharpoons$ over $\mathbb{L}_{\mathrm{S}}$ and $\mathbb{L}_{\mathrm{P}}$ that relates the semantically-equivalent serial and parallel program pairs. Given two monolingual datasets $L_{\mathrm{S}} \subset \mathbb{L}_{\mathrm{S}}$ and $L_{\mathrm{P}} \subset \mathbb{L}_{\mathrm{P}}$, the problem is to learn a translator F such that $\forall x \in \mathbb{L}_{\mathrm{S}}, (\exists u \in \mathbb{L}_{\mathrm{P}}, x \leftrightharpoons u) \rightarrow (x \leftrightharpoons F(x))$.

The main challenge of the problem is that the alignment between dataset L_{S} and L_{P} is lacking, thus the model must be trained under an unsupervised approach. The knowledge about the semantic alignment have to be firstly induced *a priori* because of the absence of an aligned dataset, and to be then learned *a posteriori* because that the auto-parallelization task itself is non-trivial. According to Rice's Theorem of computability theory (Rice, 1953), there is no set of rules that can accurately model the relation $\leftrightharpoons$, because it is *undecidable* whether two programs are semantically-equivalent.

Table 1. **Statistics of the built dataset.** The Monolingual Corpora serve as the training dataset for unsupervised training, while the Paired Corpora is for validation and test. We show the total sizes, the number of functions, and detailed statistics of tokens.

	MONOLINGUAL CORPORA		PAIRED CORPORA
	C	CUDA	C-CUDA
SIZES	977 MB	165 MB	305 KB
FUNCTIONS	501,732	129,497	364
TOKENS	134,961,512	30,818,416	80,788
- AVG	268	237	111
- MAX	1,000	1,000	470
- MIN	9	9	27

3. Dataset

In this section, we first present the design requirements and then show details of the constructed dataset.

3.1. Requirements of the Dataset

There are three key requirements in the dataset design for studying the C-to-CUDA translation.

- **Large-scale.** The learned model is expected to generalize well to a wide variety of workloads, and thus the dataset should contain abundant workloads from various fields such as graphical processing, HPC, and machine learning.
- **Function-level.** As the glue-code specified by CUDA programming model (e.g., kernel launch function) is relatively easy to generate, the dataset focuses on the challenging fine-grained translation within each function.
- **Compute-intensive.** The key challenge of the C-to-CUDA translation is to convert sequential C loops to parallel CUDA code, and thus most functions in the dataset should be compute-intensive with multiple nested loops.

3.2. Dataset Construction

Data collection. It is non-trivial to collect large-scale monolingual corpora, especially for the scarce CUDA code. Existing program translation approaches such as TransCoder (Roziere et al., 2020) usually collect the source code with a simple SQL query from the GitHub Public Dataset on Google Big Query[1], which contains only a limited number of CUDA files (i.e., 97,330 files). Instead, we crawl all the CUDA codes available on GitHub (i.e., 617,048 files) by using GitHub Search API and collect the C codes in the same directory or repositories as well, so as to mine potential correlation between them.

Data cleaning. After collecting the source code files, we perform data cleaning as follows. First, we extract all functions from the C and CUDA files. Then, we tokenize the functions and remove all the comments by using regular expression. Finally and most importantly, we filter out unqualified C and CUDA code based on their distinct characteristics. Regarding the C code, the key principle is to filter out a great amount of functional codes (e.g., control and communication, system calls, front-end codes), and retain compute-intensive kernels for graphical processing, HPC, and machine learning. Therefore, we detect the nested loops by leveraging the tree-sitter syntax parser[2] and inspect the loop to check whether it contains intensive computation on vector or matrix data structures, which are suitable for parallelization. Regarding the CUDA code, the key principle is to filter out duplicated codes, as most of the CUDA kernels are frequently reused because of cumbersome programming burden. Thus, we filter out 95.53% of the original CUDA code by performing a text similarity check.

Data labeling. For validation and test, we label the paired corpora[3] of C-CUDA by carefully inspecting the C and CUDA code in the same file. We also conduct an extra compilation check to further improve the data quality.

3.3. Dataset Statistics

Table 1 summarizes statistics of the built dataset, which contains large-scale monolingual corpora for unsupervised training and light-weighted paired corpora for validation and test. By following aforementioned data cleaning principles, in the monolingual corpora, we retain 25.08% (i.e., $501,732$) of the total C functions, and 4.47% (i.e., $129,497$) of the total CUDA functions. Also, we can group the functions into different fields based on their GitHub topics, including Deep Learning, Machine Learning, Physics Simulation, HPC, Image Processing, Graph Processing, etc. To demonstrate that the functions are compute-intensive, we further show the distribution of loops nesting depth in Figure 6. In short, the built dataset well satisfies the stated design requirements, that is, large-scale, function-level, and compute-intensive.

4. Learning Model

In this section, we introduce the learning framework of BabelTower and then detail the process of pretraining, back-translation, and an unsupervised training of discriminative ranking model.

4.1. Learning Framework

BabelTower is a translation model T coupled with a discriminative ranking model D. The translation model T is a Transformer architecture (Vaswani et al., 2017),

[1] https://cloud.google.com/bigquery

[2] https://tree-sitter.github.io

[3] We use paired corpora rather than parallel corpora to distinguish it from the parallel semantics.

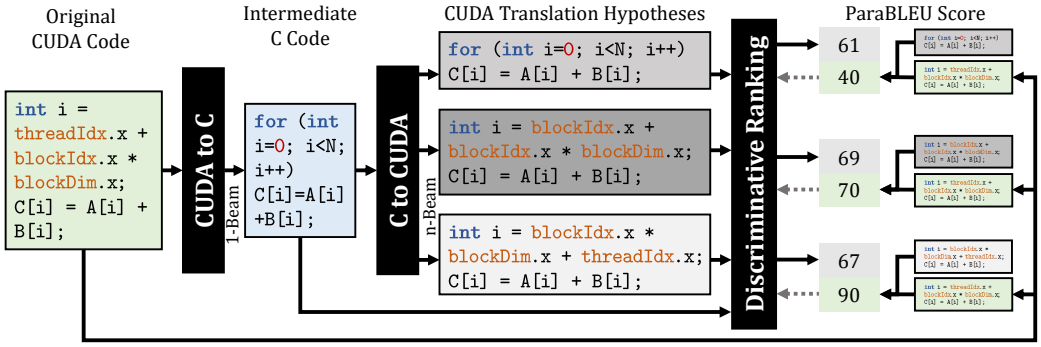

Figure 2. **Overview of BabelTower learning framework.** We train the discriminative ranking model in the back-translation step, i.e., CUDA-C-CUDA, to synthesize paired data. Further, we specially designed the metrics ParaBLEU for CUDA, and learn to predict the ParaBLEU score by minimizing the KL-divergence between the output distribution of the model and target distribution.

where the encoder e and the decoder d are shared for both C and CUDA. After training with back-translation, $T = d\left(e\left(\cdot;\theta_e, \mathrm{S}\right);\theta_d, \mathrm{P}\right)$ is able to generate CUDA code hypotheses from C input by beam searches, and $T^{-1} = d\left(e\left(\cdot;\theta_e, \mathrm{P}\right);\theta_d, \mathrm{S}\right)$ can generate C code hypotheses from CUDA input. The discriminative ranking model D is for selecting the best hypothesis within. D is also a Transformer architecture, given both the original C code x and the translated hypothesis u_i as the input sequence, output a score $o_i = D(u_i|x)$ to judge the quality of the translation. The best hypothesis is chosen as the one with the highest output score, thus the framework can be described as:

$$F(x) = \arg \max_{u_i \in T(x;\theta_T)} D(u_i|x;\theta_D) \tag{1}$$

4.2. Pretraining and Back-Translation

First, we train BabelTower with pretrained XLM model and back-translation following (Roziere et al., 2020). The translation model T is initialized by a pretrained XLM model following (CONNEAU & Lample, 2019), which is pretrained on the monolingual corpora with the Masked Language Modeling (MLM) task (Kenton & Toutanova, 2019).

After pretraining, the translation model is trained with the Denoising Auto-Encoding (DAE) task (Vincent et al., 2008) and the Back-Translation (BT) task alternatively. The source-to-target model T and the target-to-source model T^{-1} is trained in parallel until convergence on the monolingual corpora. Specifically, in BT iterations we minimize the loss function:

$$\begin{aligned}\mathcal{L}_{\mathrm{DAE}}(\theta_T) =& \mathbb{E}_{x\sim L_{\mathrm{S}}}\Delta(x, d\left(e\left(C(x);\theta_e, \mathrm{S}\right);\theta_d, \mathrm{S}\right)) \\ &+\mathbb{E}_{x\sim L_{\mathrm{P}}}\Delta(x, d\left(e\left(C(x);\theta_e, \mathrm{P}\right);\theta_d, \mathrm{P}\right))\end{aligned} \tag{2}$$

$$\begin{aligned}\mathcal{L}_{\mathrm{BT}}(\theta_T) =& \mathbb{E}_{x\sim L_{\mathrm{S}}}\Delta(x, T^{-1}\left(T_{t-1}\left(x\right);\theta_T\right)) \\ &+\mathbb{E}_{x\sim L_{\mathrm{P}}}\Delta(x, T\left(T_{t-1}^{-1}\left(x\right);\theta_T\right))\end{aligned} \tag{3}$$

where Δ denotes the sum of token-level cross-entropy losses, and $C(\cdot)$ denotes the stochastic corruption applied on the code; where T_{t-1} denotes the translation model from the previous training iteration.

The training methods described above is on par with (Roziere et al., 2020). Therefore the model is expected to generate high-quality translations between similar programming languages (such as between C++, Python, and Java). However, for language pairs such as C and CUDA, which is inherently different in programming models (sequential model versus Single Instruction, Multiple Threads, SIMT), the generated translations tend to be low in quality. This is because that the model is never trained to convert between *sequential loop structure* and *parallel semantics*, i.e., to learn the loop expansion on parallel threads.

4.3. Discriminative ranking

To address the issue stated, we adopt a discriminative ranking model following (Lee et al., 2021) to enable the model to capture the parallel semantics. However, (Lee et al., 2021) is not directly applicable on BabelTower. The reason is twofold: 1) traditional metrics such as BLEU and CodeBLEU cannot measure the similarity of parallel semantics, and 2) (Lee et al., 2021) requires aligned dataset to be trained in a supervised manner. We discuss the solutions in this subsection.

4.3.1. Capturing the Parallel Semantics

To capture the similarity of parallel semantics, we specially designed the metrics ParaBLEU for CUDA. By introducing *a prior* rules the semantically-equivalent CUDA code pairs tend to have high scores. Concretely, we introduce ParaBLEU with three factors to evaluate the CUDA codes, i.e., CUDA keywords match, loop nested similarity, and parallel semantics similarity. The CUDA keywords similarity

simply distinct CUDA code from C code in syntax-level. The loop nested similarity and parallel semantics similarity measure the loop structures and conversion from sequential C loops to parallel threads, respectively. Note that we consider leveraging similarity distance rather than using match score to evaluate the loop nested structure and parallel semantics, in case the duplicated parallel threads which gains a high match score but leads to a false parallelization. The ParaBLEU is formulated in Equation 4.

We introduce ParaBLEU with three factors to evaluate the CUDA codes, i.e., CUDA keywords match, loop nested similarity, and parallel semantics similarity. The CUDA keywords similarity simply distinct CUDA code from C code in syntax-level. The loop nested similarity and parallel semantics similarity measure the loop structures and conversion from sequential C loops to parallel threads, respectively. Note that we consider leveraging similarity distance (Levenshtein, 1966) rather than using match score to evaluate the loop nested structure and parallel semantics, in case the duplicated parallel threads which gains a high match score but leads to a false parallelization. The ParaBLEU is formulated as follows:.

$$\begin{aligned}\text{ParaBLEU} =&\alpha \cdot \text{BLEU} + \beta \cdot \text{BLEU}_{\text{weight}} \\ &+(\gamma \cdot \text{Match}_{\text{ast}} + \delta \cdot \text{Match}_{\text{df}}) \\ &\times \text{SIM}_{\text{CUDAkeywords}} \times \text{SIM}_{\text{loops}} \times \text{SIM}_{\text{parallel}}\end{aligned} \tag{4}$$

where we borrow the majority definitions of CodeBLEU and apply the parallel semantics penalty to the program-level parts.

Hypothesis

```
__global__ void gemm(
    int **A, int **B, int **C){
  for(int i = 0; i < M; i++){
    for(int j = 0; j < N; j++){
      C[i][j] = 0;
      for(int k = 0; k < K; k++) {
        C[i][j] += A[i][k] * B[k][j];
      }
    }
  }}}
```

Reference

```
__global__ void gemm(
    int **A, int **B, int **C){
  int i = blockIdx.x * blockDim.x
      + threadIdx.x;
  int j = blockIdx.y * blockDim.y
      + threadIdx.y;
  if(i < M && j < N){
    C[i][j] = 0;
    for(int k = 0; k < K; k++){
      C[i][j] += A[i][k] * B[k][j]; }}}
```

Figure 3. **Comparisons of different evaluation metrics. BLEU: 48.22, CodeBLEU: 61.40, ParaBLEU: 26.34.** The proposed ParaBLEU takes into accounts the correctness of the parallel semantics conversion, and gives a more fair evaluation for this task.

Figure 3 illustrates an example of the difference between ParaBLEU and other metrics. This function performs a simple two-dimensional matrix multiplication with three loop axes: *i*, *j*, and *k*. The hypothesis and reference almost share the same translation result, yields inflated text-level BLEU (48.22), syntax- and data-flow semantics-level CodeBLEU (61.40), and they all pass the CUDA compilation. However, the hypothesis fails to detect the potential paralilization on the outer loops and maintain the sequential loop structure, which makes an incorrect translation on the parallel semantics, resulting a low score for ParaBLEU (26.34). Overall, the ParaBLEU is a more fair evaluation metric for this task.

4.3.2. Unsupervised Training

Figure 2 shows the training process of the discriminative ranking model. After the previous training steps described in Section 4.2, the translation model in BabelTower is able to generate translations between C and CUDA but in low quality. To address the lack of aligned corpora, we train the discriminative ranking model in the back-translation setting CUDA-C-CUDA:

$$w_1, w_2, \ldots, w_N = T(u;\theta_T) = T\left(T^{-1}\left(x;\theta_T\right);\theta_T\right) \tag{5}$$

In the CUDA to C step, given the input CUDA code x, the 1-beam intermediate C code u is generated by beam search. In the C to CUDA step, we performs beam search to generate N-beam hypotheses $w_i, i \in [1, N]$ ($N \simeq 50$).

The discriminative ranking model D is then applied to the N hypotheses individually, each time takes the intermediate C code u as the original input. D learns to predict the ParaBLEU score of each hypothesis w_i, where the original CUDA code x is taken as the reference. We minimize the KL-divergence between the output distribution of the discriminative ranking model p_D and the target distribution p, both normalized to $[0, 1]$, the formulation is detailed in Equation 6.

5. Performance Evaluation

In this section, we first present the experimental methodology and results. Then, we conduct case studies to show that BabelTower can improve the performance and developer productivity of CUDA programs.

5.1. Experimental Methodology

Evaluation metrics. We use four key metrics to evaluate BabelTower. The first metric is **BLEU** (Papineni et al., 2002), which is widely used in both natural language translation and programming language translation. We further adopt a metric designed for programming languages, i.e., **CodeBLEU** (Ren et al., 2020), which not only considers the text-level similarity of weighed n-gram BLEU but also injects code syntax of abstract syntax trees and code semantics of data-flow graph. However, the above metrics lack careful consideration of parallel semantics, which is key to C-to-CUDA translation. Thus, we introduce a new metric, dubbed **ParaBLEU**, which fully takes parallel semantics into consideration. In addition, to evaluate the functionality of translated CUDA code, we also measure the **compilation accuracy**, which is the ratio of correctly compiled programs.

Table 2. **Experimental results on the C-to-CUDA dataset.** We evaluate the results of auto-parallelization approaches (i.e., Bones and PPCG) and statistical program translation (i.e., Transcoder, BabelTower) and in different metrics: BLEU, CodeBLEU, ParaBLEU, and compilation accuracy. We perform beam decoding with beam size 50 for Statistical program translation approaches.

		BLEU		CodeBLEU		ParaBLEU		compilation accuracy (%)	
		valid	test	valid	test	valid	test	valid	test
Auto parallelization	Bones	-	-	-	-	-	-	-	-
	PPCG	0	0	6.33	6.44	5.31	5.39	41.3	37.8
Statistical program translation	Transcoder	75.61	72.21	72.58	71.03	45.94	44.63	90.2	83.8
	BabelTower	76.85	74.00	78.92	77.12	55.85	54.02	93.3	92.8
	Upper Bound	80.51	74.94	79.20	77.19	57.24	56.07	96.2	98.3

Benchmarks. The benchmarks we used come from the paired corpora of the built **C-to-CUDA dataset**, where half of them are for validation and the other half are for test, with 364 C-CUDA function pairs in total.

Comparison baselines. The comparison baselines we used include the auto-parallelization approaches, i.e., Bones (Nugteren & Corporaal, 2014) and PPCG (Verdoolaege et al., 2013), and the statistical program translation approach, i.e., TransCoder (Roziere et al., 2020). Specifically, Bones is a template-based source-to-source compiler, which takes the C code with intricate annotations as the input and generates the CUDA code with built-in skeletons. PPCG is the state-of-the-art auto-parallelization approach that uses polyhedral models to exploit the parallel semantic in C code and convert it to CUDA code. TransCoder is the state-of-the-art statistical program translation approach that translates between similar programming models such as C and Python without considering auto-parallelization. Here, we extend it to support the translation from C to CUDA.

Training parameters. We build all models based on the Transformer architecture with 6 layers, 8 attention heads, and 1024 embedding size. For the pretrain model and back-translation model, we follow the same setting as TransCoder. For the discriminative reranking model, we use a separate classifier decoder of two MLP layers with *tanh* activation function, and training the model by minimizing the KL-divergence between the model distribution and target distribution with the ParaBLEU metric. We optimize BabelTower with Adam optimizer with learning rate 0.0001, and apply a learning rate decay schedule with $10,000$ warm up steps and 0.01 decay factor. We use 32 V100 GPUs for training the pretrain model and back-translation model and a RTX 8000 for training the discriminative reranking model.

5.2. Results On C-to-CUDA Dataset

Table 2 lists the experimental results on the C-to-CUDA dataset. Generally, the statistical program translation approaches perform significantly better than auto-parallelization approaches for C-to-CUDA translation, since the auto-parallelization approaches have several inherent limitations. For example, Bones completely fails to conduct end-to-end C-to-CUDA translation without manual code annotation, and thus all the evaluated metrics are not available. PPCG achieves extremely low BLEU-related scores (e.g., BLEU and ParaBLEU are 0 and 5.39, respectively) and compilation accuracy (e.g., 37.8 on the test set) since the employed polyhedral model is limited by conservative static analysis and thus not applicable to a broad range of programs.

Compared to the original TransCoder which performs beam search with machine translation score to select the 1-best hypothsis, BabelTower improves the BLEU, CodeBLEU, and ParaBLEU by 1.79, 6.09, and 9.39 on C-to-CUDA test set, respectively. Also, the compilation accuracy is significantly improved from 83.8 to 92.8. Although BabelTower achieves the best compilation accuracy among all the evaluated approaches, the absolute value of ParaBLEU is generally smaller than that of the other two metrics. The reason is that we take parallel semantics into consideration, and apply a penalty factor for the flawed conversion from the sequential C code to the parallel CUDA code. In other words, ParaBLEU provides a more accurate evaluation of the C-to-CUDA translation task. Also, we further compare the functional correctness of BabelTower against both auto-parallelization and statistical program translation approaches by randomly sampling 100 functions in the test dataset due to huge engineering efforts. Results show that Bones, PPCG, Transcoder, and BabelTower achieve 0%, 39%, 70%, and 73% in terms of functional correctness score, respectively.

We also present the theoretical upper bound of BabelTower, which is obtained by selecting the best hypothsis of each evaluation metric from all candidates (e.g., 50) generated by beam search. Except for the compilation accuracy, all

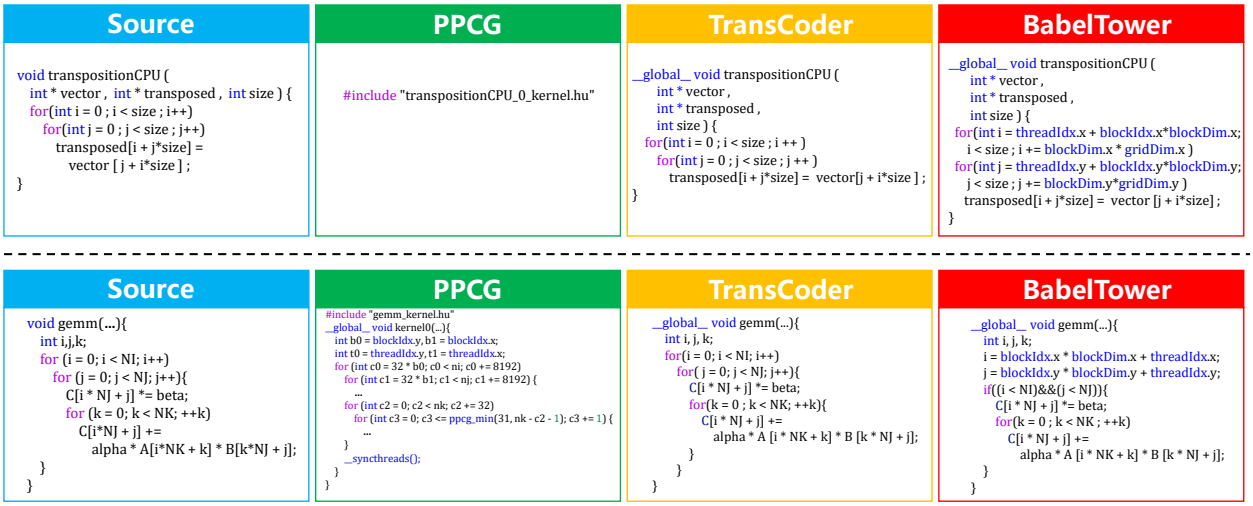

Figure 4. **Case studies on the conversion of parallel semantics.** BabelTower generates correct CUDA codes on both cases. In contrast, PPCG fails to analyze whether or not a loop can be parallelized, while Transcoder only performs code translation but fails to conduct automatic parallelization.

the BLEU-related metrics of BabelTower are close to that of the upper bound. The reason that the upper bound of compilation accuracy is hard to reach is because existing BLEU-related metrics do not directly correspond to the compilation. It is expected that the compilation accuracy might be improved by introducing the compilation-accuracy-guided metrics to the reranking model, which is left as our future work.

5.3. Case Studies

We conduct case studies to demonstrate that BabelTower performs better in identifying potentially parallel semantics from the C code and converting to well-parallelized CUDA code following SIMT rules, compared against the state-of-the-art auto-parallelization approach (i.e., PPCG) and program translation approach (i.e., TransCoder). Concretely, we consider 2 well-known computation programs including transpose and gemm, which are shown in Figure 4.

Regarding the transpose computation, only TransCoder and BabelTower can generate functionally correct CUDA code while PPCG fails due to its conservative static analysis on determine whether or not a loop can be parallelized. Moreover, the generated CUDA code of TransCoder is almost the same as the original sequential C code, in other words, TransCoder fails to conduct automatic parallelization. On contrary, BabelTower is able to generate well-parallelized CUDA code even with non-intuitive block-level parallelism, which indicates that each threads performs a dozen of computations to gain high utilization.

Regarding the gemm computation, although all evaluated approaches manage to generate correct CUDA code, the benefit of BabelTower is two-fold. The first is that the CUDA code of BabelTower is much more human-readable than that of PPCG, as shown in the appendix (see Figure 7), since the PPCG employs formal mathematical formulation for code generation. The second is that BabelTower generate high-performance well-parallelized code compared to TransCoder, which still generates almost the same code as the original sequential C code. Moreover, the code of Transcoder is functionally correct only when running with a single thread, otherwise different parallel threads will concurrently write in the same location of the global memory and lead to incorrect results.

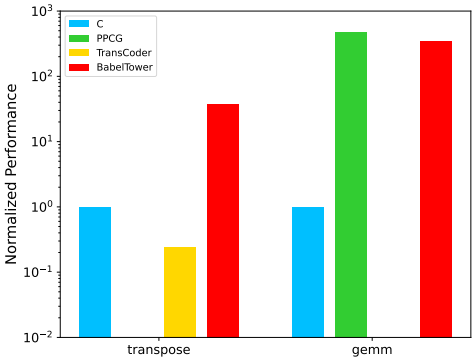

Figure 5. **Comparisons on the performance of the generated code in Figure 4.** BabelTower generates correct CUDA codes on both cases, and the speedup over the sequential C code reaches at most $347\times$.

In addition to evaluating the functional correctness, we also compare the performance of the generated code, as well as the original sequential C code. Figure 5 shows that the

Table 3. **Productivity improvement by BabelTower.** We evaluate the manual efforts of developing CUDA program with- and without- the guidance of BabelTower for CUDA experts and novices on two cases. On average, BabelTower reduces the development efforts by 2.4× and 3.8× at most for CUDA experts and novices, respectively.

	TIME(S)	KERNELXOR	GESUMMV
CUDA EXPERTS	MANUAL	109	334
	W/ BABELTOWER	76	138
	TIME SAVING	1.4×	2.4×
CUDA NOVICES	MANUAL	503	1,190
	W/ BABELTOWER	150	310
	TIME SAVING	3.3×	3.8×

CUDA code generated by BabelTower outperforms that of TransCode and the sequential C code by at most 157× and 347×, respectively. Although the performance of generated CUDA code of BabelTower is slightly worse than that of PPCG, PPCG only works on the studied gemm computation.

In summary, the above case studies well demonstrate that BabelTower greatly outperforms the state-of-the-art auto-parallelization approach, which fail to address a broad range of compute-intensive C programs, and statistical program translation approach, which fails to conduct automatic parallelization during translation.

5.4. Improving Productivity of CUDA

In addition to validating BabelTower on the paired corpora with labeled ground truth, we further use BabelTower to improve the productivity of real-life CUDA programs. Concretely, given a specific C code either collected from legacy repositories or manually programmed, since BabelTower cannot guarantee semantically-equivalent translation, we measure the additional manual efforts invested to modify the translated code to the functionally correct CUDA programs. The measured efforts are compared to that of directly developing the CUDA programs without the guidance of BabelTower.

In this study, we invite 8 computer science students as participants to write a CUDA program based on the given C program. Those participants are divided into two groups (i.e., CUDA experts and CUDA novices) by their familiarity with CUDA programming models. Table 3 reports the manual efforts of developing the CUDA programs with- and without- the guidance of BabelTower. With the help of BabelTower, the development efforts are reduced by 2.4× and 3.8× at most for CUDA experts and novices, respectively. We illustrate the two examples in Figure 8.

6. Related Work

Auto-parallelization. Many approaches focus on auto-parallelization using polyhedral model (Benabderrahmane et al., 2010; lim; Liu et al., 2017; Pouchet et al., 2013; Verdoolaege, 2010; Li et al., 2013; Baskaran et al., 2008). For instance, Pluto (Bondhugula et al., 2008a;b) enables end-to-end auto-parallelization and locality optimization of affine programs for multi-core processors. PPCG (Verdoolaege et al., 2013) uses polyhedral model to exploit the parallel semantic in C code and convert it to CUDA automatically. DawnCC (Mendonça et al., 2017) can automatically detect potential parallel code in C/C++ programs and then insert OpenMP/OpenACC directives where appropriate. Unfortunately, most of these approaches limited in generality (i.e., only supports a single statement in perfectly nested loops) and scalability (i.e., unscalable in complex designs). Auto-parallelization based on code template is also an important research field. For example, Bones (Nugteren & Corporaal, 2014) transforms annotated C programs to parallel CUDA or OpenCL with the built-in skeletons, where the skeleton sets are based on well defined grammar and vocabulary. However, it requires considerable manual efforts for code annotation. As a result, despite of all these advances for decades, the improved performance of auto-parallelization is still limited, and thus cannot meet the requirement of the community.

Statistical program translation. The statistical program translation is inspired by recent advances on machine translation including statistical machine translation (SMT) and neural machine translation (NMT). In the line of SMT, (Karaivanov et al., 2014; Nguyen et al., 2013; 2016; Oda et al., 2015; Nguyen et al., 2015) are proposed to use SMT for code migration, but they cannot be extended well for API usages. Therefore for better API usages, (Nguyen et al., 2016; 2017) facilitate the translation from Java to C# by using word embeddings. An encoder/decoder is also leveraged in (Gu et al., 2016) to learn the semantics of queries and the corresponding API sequences. A number of works are also proposed in NMT line. Transcoder (Roziere et al., 2020) trains a neural networks to translate functions and methods between programming languages with denoising auto-encoder and back-translation. TransCoder-ST (Roziere et al., 2021b) increases a parallel corpus translation by leveraging an automated unit-testing system. CodeXGLUE (Lu et al., 2021) aggregates a number of programming benchmarks based on CodeBLEU. In spite of yielding comparable performance, these approaches still cannot be applied in C-to-CUDA translation due to the scarcity of effective dataset and parallel semantics. Differently, we create C-to-CUDA dataset for BabelTower and then cope with unpaired corpora and parallel semantic conversion by using back-translation with a discriminative reranker. As such, BabelTower can effectively translate sequential C to prallel CUDA.

Reranking approaches. (Shen et al., 2004; Och et al., 2004) are the seminal work of using discriminative reranking for SMT. (Liu et al., 2018; Imamura & Sumita, 2017; Wang et al., 2017; Yu et al., 2016; Yee et al., 2019) have focused on generative reranking for NMT, which optimize the parameters of the reranker with a criterion. In addition, several work (Auli & Gao, 2014; Ehara, 2017) combines the advantage of SMT and NMT by reranking one with the other. Recently, (Lee et al., 2021) can achieve better performance by training large transformer models with reranking objective. We extend this work to BabelTower with two key differences. First, instead of supervised training on aligned dataset, unsupervised translation based on unpaired mono-lingual corpora is used in BabelTower. Second, since the traditional metrics such as BLEU and CodeBLEU fail to capture the similarity of parallel semantics for CUDA, specially designed ParaBLEU is used in BabelTower.

7. Conclusion and Future Work

In this paper, we propose a novel learning framework, i.e., BabelTower, to translate from sequential C to Parallel CUDA, which can significantly relieve the burden of GPU programming. In addition to building the first large-scale dataset, we also introduce a novel learning framework to cope with unpaired corpora and parallel semantic conversion. Experimental results show that BabelTower outperforms state-of-the-art by 1.79, 6.09 and 9.39 in terms of BLEU, CodeBLEU, and ParaBLEU, respectively. Moreover, the CUDA code generated by BabelTower attains a speedup of up to $347\times$ over the sequential C code, and the developer productivity is also significantly improved.

As the first attempt to tackle the challenging auto-parallelized program translation problem, the functional correctness of C-to-CUDA translation will be the main focus of our future work. In addition to further improving the accuracy of learning models, there are several potential techniques to guarantee the functional correctness, e.g., introducing formal methods such as Satisfiability Modulo Theories during or after training. It is expected that this work will inspire more advanced research on this field.

Acknowledgements

This work is partially supported by the National Key Research and Development Program of China (under Grant 2020AAA0103802), the NSF of China (under Grants 61925208, 62102398, 62002338, 61732020, U19B2019), Strategic Priority Research Program of Chinese Academy of Science (XDB32050200), Beijing Academy of Artificial Intelligence (BAAI) and Beijing Nova Program of Science and Technology (Z191100001119093), CAS Project for Young Scientists in Basic Research (YSBR-029), Youth Innovation Promotion Association CAS and Xplore Prize.

References

Auli, M. and Gao, J. Decoder integration and expected bleu training for recurrent neural network language models. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (ACL)*, pp. 136–142, 2014.

Baskaran, M. M., Bondhugula, U., Krishnamoorthy, S., Ramanujam, J., Rountev, A., and Sadayappan, P. A compiler framework for optimization of affine loop nests for gpgpus. In *Proceedings of the 22nd Annual International Conference on Supercomputing (ISC)*, pp. 225–234, 2008.

Benabderrahmane, M.-W., Pouchet, L.-N., Cohen, A., and Bastoul, C. The polyhedral model is more widely applicable than you think. In *Proceedings of International Conference on Compiler Construction (CC)*, pp. 283–303, 2010.

Bondhugula, U., Baskaran, M., Krishnamoorthy, S., Ramanujam, J., Rountev, A., and Sadayappan, P. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In *Proceedings of International Conference on Compiler Construction (CC)*, pp. 132–146, 2008a.

Bondhugula, U., Hartono, A., Ramanujam, J., and Sadayappan, P. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 101–113, 2008b.

Chen, X., Liu, C., and Song, D. Tree-to-tree neural networks for program translation. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems (NeurIPS)*, pp. 2552–2562, 2018.

CONNEAU, A. and Lample, G. Cross-lingual language model pretraining. In Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems (NeurIPS)*, volume 32, 2019.

Ehara, T. Smt reranked nmt. In *Proceedings of the 4th Workshop on Asian Translation (WAT2017)*, pp. 119–126, 2017.

Gu, X., Zhang, H., Zhang, D., and Kim, S. Deep api learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering(FSE)*, pp. 631–642, 2016.

Imamura, K. and Sumita, E. Ensemble and reranking: Using multiple models in the nict-2 neural machine translation system at wat2017. In *Proceedings of the 4th Workshop on Asian Translation (WAT2017)*, pp. 127–134, 2017.

Karaivanov, S., Raychev, V., and Vechev, M. Phrase-based statistical translation of programming languages. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming Software*, pp. 173–184, 2014.

Kenton, J. D. M.-W. C. and Toutanova, L. K. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of NAACL-HLT*, pp. 4171–4186, 2019.

Lee, A., Auli, M., and Ranzato, M. Discriminative reranking for neural machine translation. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics(ACL) and the 11th International Joint Conference on Natural Language Processing(IJCNLP) (Volume 1: Long Papers)*, pp. 7250–7264, 2021.

Levenshtein, V. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, 10:707, 1966.

Li, Z., Jannesari, A., and Wolf, F. Discovery of potential parallelism in sequential programs. In *2013 42nd International Conference on Parallel Processing(ICPP)*, pp. 1004–1013. IEEE, 2013.

Liu, J., Wickerson, J., Bayliss, S., and Constantinides, G. A. Polyhedral-based dynamic loop pipelining for high-level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(9):1802–1815, 2017.

Liu, Y., Zhou, L., Wang, Y., Zhao, Y., Zhang, J., and Zong, C. A comparable study on model averaging, ensembling and reranking in nmt. In *CCF International Conference on Natural Language Processing and Chinese Computing(NLPCC)*, pp. 299–308. Springer, 2018.

Lu, S., Guo, D., Ren, S., Huang, J., Svyatkovskiy, A., Blanco, A., Clement, C., Drain, D., Jiang, D., Tang, D., et al. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*, 2021.

Mendonça, G., Guimarães, B., Alves, P., Pereira, M., Araújo, G., and Pereira, F. M. Q. Dawncc: automatic annotation for data parallelism and offloading. *ACM Transactions on Architecture and Code Optimization (TACO)*, 14(2):1–25, 2017.

Mendonça, G., Guimarães, B., Alves, P., Pereira, M., Araújo, G., and Pereira, F. M. Q. a. Dawncc: Automatic annotation for data parallelism and offloading. *ACM Trans. Archit. Code Optim.*, 14(2), may 2017.

Nguyen, A. T., Nguyen, T. T., and Nguyen, T. N. Lexical statistical machine translation for language migration. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering(FSE)*, pp. 651–654, 2013.

Nguyen, A. T., Nguyen, T. T., and Nguyen, T. N. Divide-and-conquer approach for multi-phase statistical migration for source code. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 585–596, 2015.

Nguyen, T. D., Nguyen, A. T., and Nguyen, T. N. Mapping api elements for code migration with vector representations. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, pp. 756–758. IEEE, 2016.

Nguyen, T. D., Nguyen, A. T., Phan, H. D., and Nguyen, T. N. Exploring api embedding for api usages and applications. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pp. 438–449. IEEE, 2017.

Nugteren, C. and Corporaal, H. Bones: An automatic skeleton-based c-to-cuda compiler for gpus. *ACM Transactions on Architecture and Code Optimization (TACO)*, 11(4):1–25, 2014.

Och, F. J., Gildea, D., Khudanpur, S., Sarkar, A., Yamada, K., Fraser, A., Kumar, S., Shen, L., Smith, D. A., Eng, K., et al. A smorgasbord of features for statistical machine translation. In *Proceedings of the Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics(HLT-NAACL)*, pp. 161–168, 2004.

Oda, Y., Fudaba, H., Neubig, G., Hata, H., Sakti, S., Toda, T., and Nakamura, S. Learning to generate pseudo-code from source code using statistical machine translation. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 574–584. IEEE, 2015.

Papineni, K., Roukos, S., Ward, T., and Zhu, W.-J. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics(ACL)*, pp. 311–318, 2002.

Pouchet, L.-N., Zhang, P., Sadayappan, P., and Cong, J. Polyhedral-based data reuse optimization for configurable computing. In *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*, pp. 29–38, 2013.

Ren, S., Guo, D., Lu, S., Zhou, L., Liu, S., Tang, D., Sundaresan, N., Zhou, M., Blanco, A., and Ma, S. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*, 2020.

Rice, H. G. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.

Roziere, B., Lachaux, M.-A., Chanussot, L., and Lample, G. Unsupervised translation of programming languages. In Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M. F., and Lin, H. (eds.), *Advances in Neural Information Processing Systems(NeurIPS)*, volume 33, pp. 20601–20611. Curran Associates, Inc., 2020.

Roziere, B., Lachaux, M.-A., Szafraniec, M., and Lample, G. Dobf: A deobfuscation pre-training objective for programming languages. In *Proceedings of Advancesin Neural Information Processing Systems 34 (NeurIPS)*, 2021a.

Roziere, B., Zhang, J. M., Charton, F., Harman, M., Synnaeve, G., and Lample, G. Leveraging automated unit tests for unsupervised code translation. *arXiv preprint arXiv:2110.06773*, 2021b.

Sanders, J. and Kandrot, E. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.

Shen, L., Sarkar, A., and Och, F. J. Discriminative reranking for machine translation. In *Proceedings of the Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics(HLT-NAACL)*, pp. 177–184, 2004.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. Attention is all you need. In *Advances in Neural Information Processing Systems(NeurIPS)*, pp. 5998–6008, 2017.

Verdoolaege, S. isl: An integer set library for the polyhedral model. In *International Congress on Mathematical Software(ICMS)*, pp. 299–302. Springer, 2010.

Verdoolaege, S., Carlos Juega, J., Cohen, A., Ignacio Gomez, J., Tenllado, C., and Catthoor, F. Polyhedral parallel code generation for cuda. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):1–23, 2013.

Vincent, P., Larochelle, H., Bengio, Y., and Manzagol, P.-A. Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th International Conference on Machine Learning(ICML)*, pp. 1096–1103, 2008.

Wang, Y., Cheng, S., Jiang, L., Yang, J., Chen, W., Li, M., Shi, L., Wang, Y., and Yang, H. Sogou neural machine translation systems for wmt17. In *Proceedings of the Second Conference on Machine Translation*, pp. 410–415, 2017.

Yee, K., Ng, N., Dauphin, Y. N., and Auli, M. Simple and effective noisy channel modeling for neural machine translation. *arXiv preprint arXiv:1908.05731*, 2019.

Yu, L., Blunsom, P., Dyer, C., Grefenstette, E., and Kocisky, T. The neural noisy channel. *arXiv preprint arXiv:1611.02554*, 2016.

A. Ablation Study

We conduct detailed ablation study to demonstrate the effectiveness of BabelTower, including using different metrics for building reranker, different beam sizes, and different data set for training reranker.

Different metrics. Table 4 lists the effects of using different metrics (i.e., BLEU, CodeBLEU, and ParaBLEU) for building the reranker. Experimental results show that ParaBLEU is the best among all metric candidates in terms of improving the BLEU, CodeBLEU, ParaBLEU, and compilation accuracy by at most 0.18, 2.73, 1.64, and 13.7, respectively. Note that the reranker-BLEU only captures the text-level features, which leads to a poor performance in compilation accuracy.

Table 4. **Ablation study on the choices of metrics for BabelTower.** All results are evaluated on the C-to-CUDA test with beam size 50.

	reranker (BLEU)	reranker (CodeBLEU)	reranker (ParaBLEU)
BLEU	73.82	73.90	74.00
CodeBLEU	74.39	77.12	77.12
ParaBLEU	52.38	53.35	54.02
compilation accuracy (%)	79.1	89.4	92.8

Different beam sizes. By using ParaBLEU for reranking, Table 5 further lists the effects of different beam sizes, i.e., 1, 5, 10, 25, 50. Experimental results show that generally larger beam size can obtain better models in terms of evaluated metrics. Concretely, beam size of 50 improves the BLEU, CodeBLEU, and ParaBLEU, and compilation accuracy by 1.79, 6.09, 7.60, and 4.50, respectively, than that beam size of 1. Note that the most significant ParaBLEU is improved mostly, which also demonstrates the effectiveness of using ParaBLEU for reranking.

Table 5. **Ablation study on the effects of beam sizes for BabelTower.** All results are evaluated on the C-to-CUDA test by optimizing the model with ParaBLEU.

beam size	1	5	10	25	50
BLEU	72.21	69.96	73.98	73.59	74.00
CodeBLEU	71.03	71.41	73.83	75.38	77.12
ParaBLEU	46.42	45.19	47.89	51.86	54.02
compilation accuracy (%)	88.3	91.7	89.4	89.9	92.8

Different training sets. Table 6 lists the effects of different training sets for building the reranker. In addition to the adopted policy for filtering data pairs generated by back-translation (i.e., Filter-BT Data), we also consider to use all generated data pairs (i.e., Full-BT Data) and randomly select half of the generated data pairs (i.e., Half-BT Data) for training the reranker. It can be observed that the proposed filtering policy performs significantly better than using data directly generated by back-translation (e.g., ParaBLEU is improved by at most 18.11), even the data size of latter is much larger for training.

Table 6. **Ablation study on the effects of training data for BabelTower's raranking model.** All results are evaluated on the C-to-CUDA test by optimizing the model with ParaBLEU.

	Full-BT data	Half-BT data	Filtered-BT data
data pairs	$108,735$	$54,825$	$12,571$
BLEU	54.68	52.60	74.00
CodeBLEU	66.20	65.23	77.12
ParaBLEU	36.91	35.91	54.02
compilation accuracy (%)	77.1	81.5	92.8

B. Formulation of the Training for Discriminative Ranking

In the CUDA to C step, given the input CUDA code x, the 1-beam intermediate C code u is generated by beam search. In the C to CUDA step, we performs beam search to generate N-beam hypotheses $w_i, i \in [1, N]$ ($N \simeq 50$).

The discriminative ranking model D is then applied to the N hypotheses individually, each time takes the intermediate C code u as the original input. D learns to predict the ParaBLEU score of each hypothesis w_i, where the original CUDA code x is taken as the reference. We minimize the KL-divergence between the output distribution of the discriminative ranking model p_D and the target distribution p, both normalized to $[0, 1]$, the formulation is detailed in Equation 6.

$$
\begin{aligned}
p_D(w_i|u;\theta_D) &= \frac{\exp\left(D\left(w_i|u;\theta_D\right)\right)}{\sum_{j=1}^{N}\exp\left(D\left(w_j|u;\theta_D\right)\right)} \\
score_i &= \frac{\text{ParaBLEU}\left(w_i|x\right) - \min_j \text{ParaBLEU}\left(w_j|x\right)}{\max_j \text{ParaBLEU}\left(w_j|x\right) - \min_j \text{ParaBLEU}\left(w_j|x\right)} \\
p(w_i) &= \frac{\exp\left(score_i/\tau\right)}{\sum_{j=1}^{N}\exp\left(score_j/\tau\right)} \\
\mathcal{L}(\theta_D) &= -\sum_{j=1}^{N} p(w_i)\log\frac{p_D(w_i|u;\theta_D)}{p(w_i)},
\end{aligned}
\tag{6}
$$

where τ is the softmax temperature set to 0.5 in practice.

C. Distribution of the Loop Nesting Depth in the C-to-CUDA Dataset

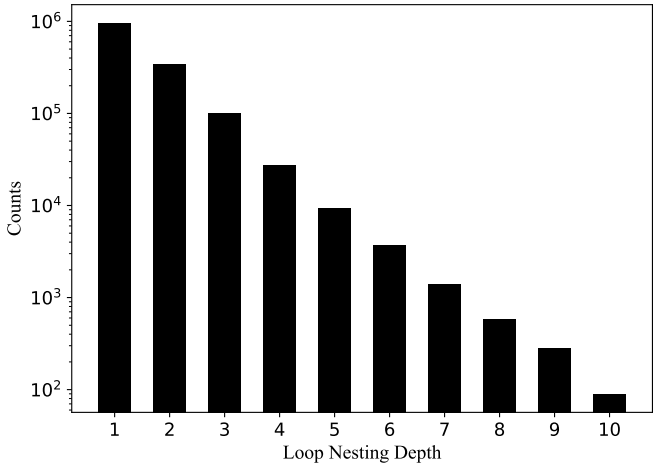

Figure 6. **The distribution of loop nesting depth in the dataset.**

D. Generated Code of PPCG and BabelTower

PPCG

```
#include "gemm_kernel.hu"
__global__ void kernel0(
    double *A, double *B,
    double *C, double alpha, double beta,
    int ni, int nj, int nk)
{
    int b0 = blockIdx.y, b1 = blockIdx.x;
    int t0 = threadIdx.y, t1 = threadIdx.x;
    double private_C[1][2];

    #define ppcg_min(x,y) ({ __typeof__(x) _x = (x); __typeof__(y) _y = (y); _x < _y ? _x : _y; })
    for (int c0 = 32 * b0; c0 < ni; c0 += 8192)
        for (int c1 = 32 * b1; c1 < nj; c1 += 8192) {
        if (b1 <= 31 && nj >= 32 * b1 + t1 + 1 && ni >= t0 + c0 + 1 && c1 == 32 * b1) {
            private_C[0][0] = C[(t0 + c0) * 1024 + (32 * b1 + t1)];
        if (nj >= 32 * b1 + t1 + 17)
            private_C[0][1] = C[(t0 + c0) * 1024 + (32 * b1 + t1 + 16)];
        }
        if (ni >= t0 + c0 + 1 && nj >= t1 + c1 + 1) {
            private_C[0][0] *= beta;
        if (nj >= t1 + c1 + 17)
            private_C[0][1] *= beta;
        for (int c2 = 0; c2 < nk; c2 += 32)
            for (int c3 = 0; c3 <= ppcg_min(31, nk - c2 - 1); c3 += 1) {
                private_C[0][0] +=
                    ((alpha * A[(t0 + c0) * 1024 + (c2 + c3)]) * B[(c2 + c3) * 1024 + (t1 + c1)]);
            if (nj >= t1 + c1 + 17)
                private_C[0][1] +=
                    ((alpha * A[(t0 + c0) * 1024 + (c2 + c3)]) * B[(c2 + c3) * 1024 + (t1 + c1 + 16)]);
            }
            if (b1 <= 31 && c1 == 32 * b1) {
                C[(t0 + c0) * 1024 + (32 * b1 + t1)] = private_C[0][0];
                if (nj >= 32 * b1 + t1 + 17)
                    C[(t0 + c0) * 1024 + (32 * b1 + t1 + 16)] = private_C[0][1];
            }
        }
        __syncthreads();
    }
}
```

Figure 7. **The code generated by PPCG for gemm operation**.

Hypothesis

```
__global__ void kernelXor(
  unsigned int key,
  char * input_str_cuda,
  unsigned char * possible_plaintext_str_cuda,
  int input_length){
  int id = blockIdx.x * blockDim.x
              + threadIdx.x;
  if(id >= input_length)
    return ;
  int keyIndex = id % 4;
  int keyCharPtr = ((char *)&key);
  char keyChar = keyCharPtr[keyIndex];
  possible_plaintext_str_cuda[id] =
        keyChar ^ input_str_cuda[id];
}
```

Reference

```
__global__ void kernelXor(
  unsigned int key,
  char * input_str_cuda,
  unsigned char * possible_plaintext_str_cuda,
  int input_length){
  int id = blockIdx.x * blockDim.x + threadIdx.x;
  if(id >= input_length)
    return;
  int keyIndex = id % 4;
  char* keyCharPtr = ((char *) & key);
  char keyChar = keyCharPtr[keyIndex];
  possible_plaintext_str_cuda[id] =
      keyChar ^ input_str_cuda[id];
}
```

Hypothesis

```
#define DATA_TYPE float
#define N 1024
__global__ void gesummv(
  int n,
  DATA_TYPE alpha, DATA_TYPE beta,
  DATA_TYPE* A, DATA_TYPE* B,
  DATA_TYPE* tmp, DATA_TYPE* x, DATA_TYPE* y){

  int i, j;
  i = blockIdx.x * blockDim.x + threadIdx.x;
  j = blockIdx.y * blockDim.y + threadIdx.y;

  if(i < N && j < N) {
    tmp[i] = A[i * N + j] * x[j] + tmp[i];
    y[i] = B[i * N + j] * x[j] + y[i];
  }
  y[i] = alpha * tmp[i] + beta * y[i];
}
```

Reference

```
#define DATA_TYPE float
#define N 1024
__global__ void gesummv(
  int n,
  DATA_TYPE alpha, DATA_TYPE beta,
  DATA_TYPE* A, DATA_TYPE* B,
  DATA_TYPE* tmp, DATA_TYPE* x, DATA_TYPE* y){

  int i, j;
  i = blockIdx.x * blockDim.x + threadIdx.x;
  if(i < N){
    tmp[i] = 0;
    y[i] = 0;
    for(j = 0; j < N; j++){
      tmp[i] = A[i * N + j] * x[j] + tmp[i];
      y[i] = B[i * N + j] * x[j] + y[i];
    }
    y[i] = alpha * tmp[i] + beta * y[i];
  }
}
```

Figure 8. **Examples of the wrong translation results for BabelTower**. Regarding the kernelXor operation, BabelTower successfully performs the conversion of parallel semantics for the sequential loop, but introduces a syntax error for the definition of *keyCharPtr*. Regarding gesummv operation, BabelTower incorrectly parallels both loops, but the inner *j* loop should maintain the sequential loop structure to ensure functional correctness.