Extracting features for HLS kernels using the CodeBERT NL-PL model

Georgios Milis

ECE, NTUA

Embedded Systems Design, Extra Project

Introduction

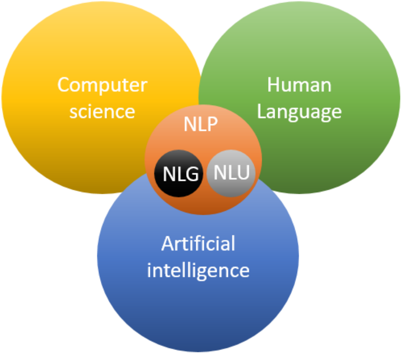
HLS: automatically design digital hardware that implements an algorithm written in a high-level programming language.

Difficulty: DSE to find the optimal `#pragmas` and parameters takes time and resources.

Motivation: language models can be used to extract useful features from applications so that the design space can be pruned. Those features could also be used to build models that predict the performance of new kernels.

What is NLP?

An interdisciplinary subfield of computer science, AI, and linguistics, concerned with how to process and analyze large amounts of natural language data.



Applications, to name a few

- Syntactic analysis
- Language modeling
- Information retrieval
- Data mining
- Machine translation
- Natural language generation
- Text summarization
- Speech processing
- Chatbots

[Jurafsky and Martin, 2021]

Modern AI: deep neural networks

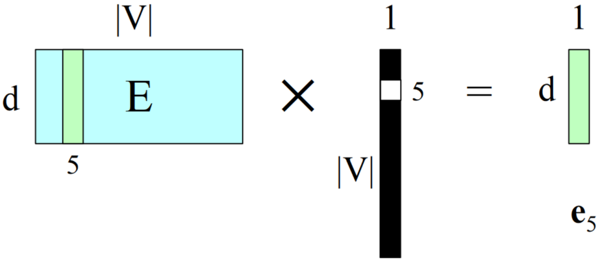
Deep learning is a class of machine learning algorithms that uses multiple layers to progressively extract higher-level features from the raw input.

In the last decade, it has risen to be the standard approach in any intelligence task. Why?

- Data availability
- Hardware acceleration: GPU, FPGA, TPU (Google)

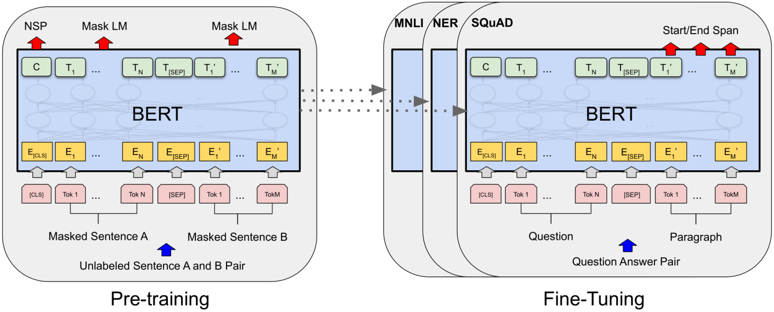
Neural language models

These models are based on neural networks and use continuous, learned, vector representations of words (word *embeddings*) for their computations.



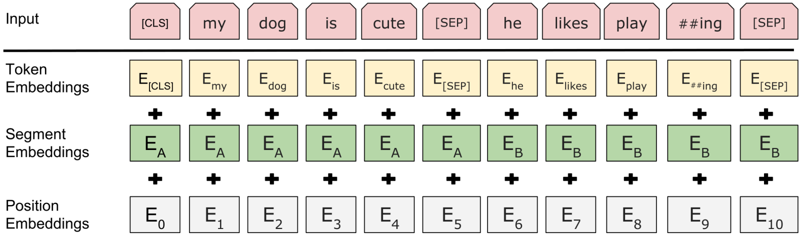
BERT language model

Bidirectional **E**ncoder **R**epresentations from **T**ransformers, by Google Research, 110M parameters [Devlin et al., 2019]



BERT training

- Unsupervised pre-training on:
  1. Masked language modeling: correctly predict a hidden token
  2. Next sentence prediction
- Fine-tuning separate models for each dowstream task



CodeBERT model

A bimodal pre-trained model for programming languages and natural language, by Microsoft Research [Feng et al., 2020].

It uses the RoBERTa architecture [Liu et al., 2019], with 125M parameters. Multimodal models learn implicit alignment between inputs of different modalities (e.g. text-to-image aligmnets).

```python
def _parse_memory(s):
    """
    Parse a memory string in the format supported by Java (e.g. 1g, 200m) and
    return the value in MiB

    >>> _parse_memory("256m")
    256
    >>> _parse_memory("2g")
    2048
    """
    units = {'g': 1024, 'm': 1, 't': 1 << 20, 'k': 1.0 / 1024}
    if s[-1].lower() not in units:
        raise ValueError("invalid format: " + s)
    return int(float(s[:-1]) * units[s[-1].lower()])
```

CodeBERT pre-training

The programming languages of the training dataset:

- Go
- Java
- JavaScript
- PHP
- Python
- Ruby

And the pre-training tasks:

- Masked language modeling
- Replaced token detection

CodeBERT fine-tuning

- Natural language code search
- NL-PL probing
- Code documentation generation
- Generalization to programming languages **not** in pre-training

For example, in natural language code search, the representation of the `[CLS]` token is used to measure the semantic relevance between code and natural language query.
CodeBERT generalizes better to unknown programming languages, in comparison to other models (evaluated on `C#`).

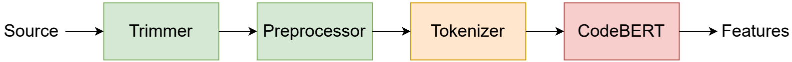
HLS kernel datasets

- MachSuite, [Reagen et al., 2014]: benchmarks for evaluating HLS tools and accelerator-centric architectures
- Rodinia, [Che et al., 2009]: another benchmark suite for heterogeneous computing, targeting multi-core CPU and GPU platforms

Kernels for: graph traversal, linear algebra, FFT, differential equations, neural networks and more. 99 code files in total.

Pipeline

Feature extraction from source code:



The extracted features will be projected in a low-dimensional space and clustered, to uncover potential structure in the dataset.

Trimming + pre-processing: spcl_example_03

```
#include "Example3.h" // Defines N, M, and T

void Stencil2D( float const memory_in[N * M], float
        memory_out[N * M]) {
L1:   float above_buffer[M];
L2:   float center_buffer[M];
L3:   for (int i = 0; i < M; ++i) {
      above_buffer[i] = memory_in[i];
   }
L4:   for (int i = 0; i < M; ++i) {
      center_buffer[i] = memory_in[M + i];
   }
L5:   for (int i = 1; i < N - 1; ++i) {
L6:     for (int j = 0; j < M; ++j) {
        const auto above = above_buffer[j];
        const auto center = center_buffer[j];
        const auto below = memory_in[(i + 1) * M + j];
        constexpr float factor = 0.3333;
        const auto average = factor * (above + center +
                below);
        memory_out[i * M + j] = average;
      }
   }
}
```

```
float above_buffer[M]; float
center_buffer[M]; for (int i = 0; i <
M; ++i) { above_buffer[i] =
memory_in[i]; } for (int i = 0; i < M;
++i) { center_buffer[i] = memory_in[
M + i]; } for (int i = 1; i < N - 1;
++i) { L6: for (int j = 0; j < M; ++j
) { const auto above = above_buffer[j];
const auto center = center_buffer[j];
const auto below = memory_in[(i + 1)
* M + j]; constexpr float factor =
0.3333; const auto average = factor * (
above + center + below); memory_out[i
* M + j] = average; } }
```

Tokenization

[Singh, 2022] A tokenizer splits text into tokens according to a set of rules. The tokens are then converted into indices, which are used to build tensors as input to a model. State-of-the-art transformer models use sub-word tokenizers.

- Code: ["int add(int a, int b) { return a + b; }"]
- Tokens: [['<s>', 'int', 'Ġadd', '(', 'int', 'Ġa', ',', 'Ġint', 'Ġb', ')', 'Ġ{', 'Ġreturn', 'Ġa', 'Ġ+', 'Ġb', ';', 'Ġ}', '</s>']]
- Token indices: [[0, 2544, 1606, 1640, 2544, 10, 6, 6979, 741, 43, 25522, 671, 10, 2055, 741, 131, 35524, 2]]

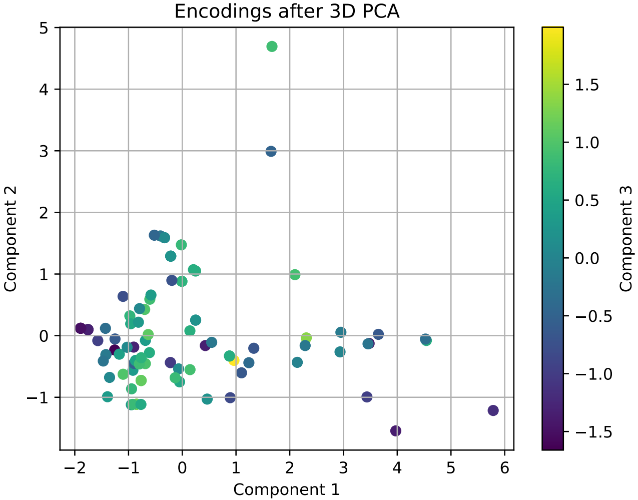
Feature extraction

- Convert index list to tensor:
  `input = tensor(token_ids) #[1, 18]`
- Forward pass:
  `output = CodeBERT(input)  #[1, 18, 768]`
- Keep only the classification head:
  `output = output[:, 0, :]   #[1, 768]`

The shape of the concatenated encodings is [99, 768]. In order to visualize it, we need to project the features in 2 or 3 dimensions.

Apply PCA

Principal component analysis, one of the most common ways of dimensionality reduction.



Comparison with previous results

Features: 110-component vectors containing information about array size, loop structures, and LLVM operations of innermost loops

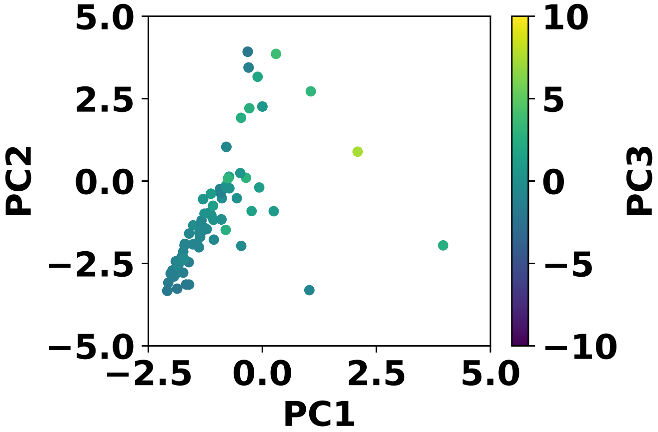
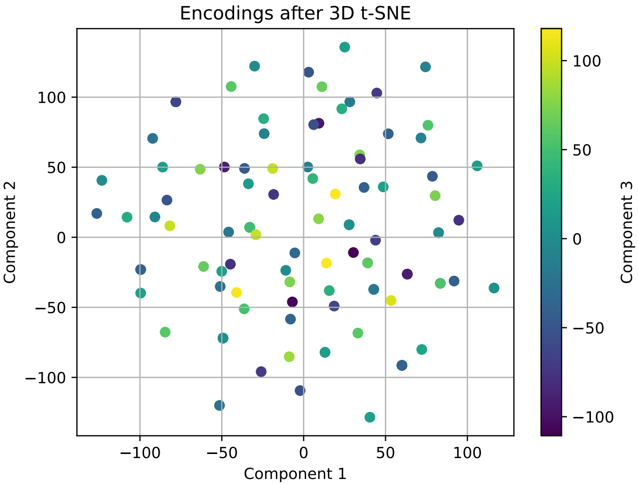


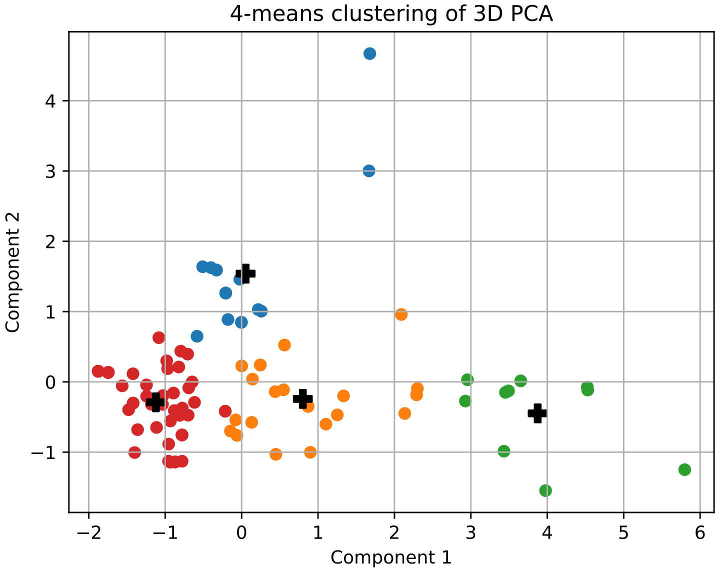
Image provided by Aggelos Ferikoglou.

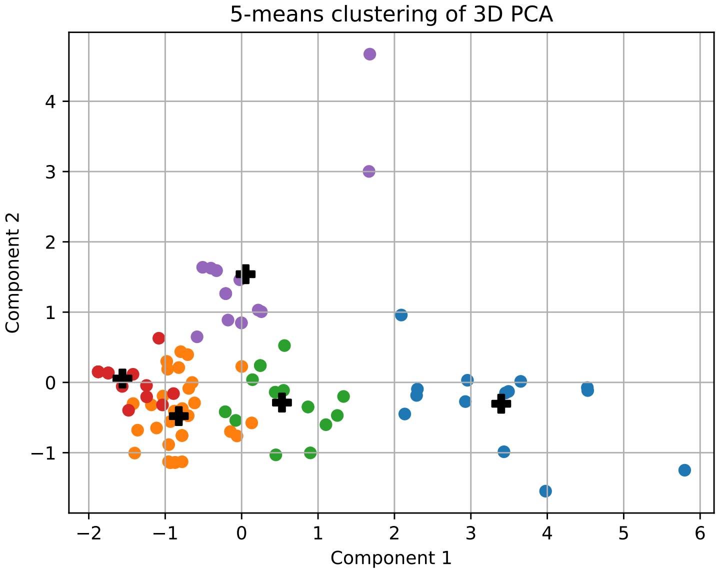
Apply t-SNE

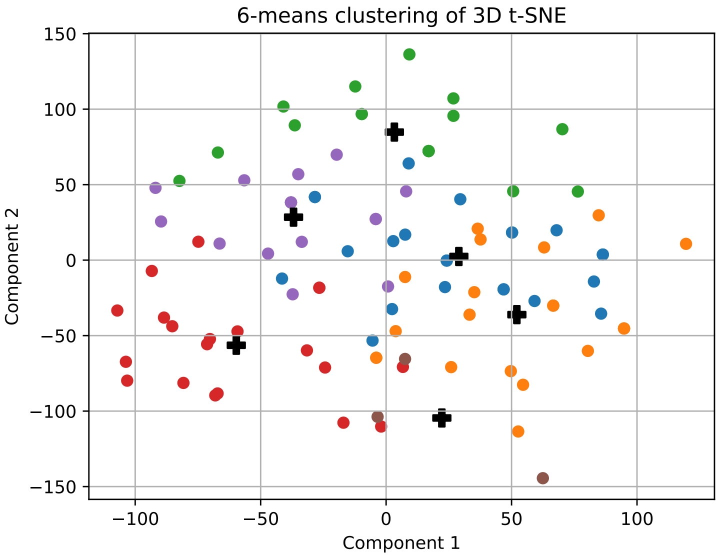
Another visualization with t-distributed stochastic neighbor embedding, a statistical method.



Clustering


4-means clustering of 3D PCA
Component 2
Component 1
4
3
2
1
0
−1
−2
−1
0
1
2
3
4
5
6


5-means clustering of 3D PCA
Component 2
Component 1
4
3
2
1
0
−1
−2
−1
0
1
2
3
4
5
6


6-means clustering of 3D t-SNE
150
100
50
0
−50
−100
−150
Component 2
−100
−50
0
50
100
Component 1

Clusters from 5-means on 3D PCA

aes
cfd_flux_0_baseline_0
cfd_flux_1_tiling_0
cfd_flux_2_pipeline_0
cfd_flux_3_unroll_0
cfd_step_factor_3_unroll_0
cfd_step_factor_4_doublebuffer
cfd_step_factor_5_coalescing
lavaMD_0_baseline
lavaMD_1_tiling_0
lavaMD_1_tiling_1
lavaMD_2_pipeline_0
lavaMD_3_unroll_0
lc_gicov_0_baseline_0
lc_mgvf_1_tiling_0

gemm—ncubed
spmv—ellpack
backprop—0—baseline—back
kmeans—0—baseline
knn—0—baseline
streamcluster_0_baseline_0

spcl_example_00
spcl_example_01
spcl_example_03
spcl_example_05

backprop
fft—transpose
sort—radix
backprop—4—doublebuffer—back
backprop—5—coalescing—back
backprop—5—coalescing—forward
backprop—6—multiddr
backprop—6—multiddr—forward
kmeans—1—tiling
kmeans—2—pipeline
kmeans—3—unroll
kmeans—4—doublebuffer
knn—3—unroll
knn—4—doublebuffer
cfd_step_factor_1_tiling_0
dilate_1_tiling_0
dilate_2_pipeline_0
dilate_3_pipeline_0
lc_gicov_1_tiling_0
lud_1_tiling_0
nw_0_baseline_0
nw_1_tiling_0
nw_2_pipeline_0
nw_3_unroll_0
nw_4_doublebuffer_0
nw_5_coalescing_0
pathfinder_4_doublebuffer_0
pathfinder_5_coalescing_0
streamcluster_1_tiling_0
streamcluster_2_pipeline_1
streamcluster_3_doublebuffer_0

streamcluster_4_coalescing_0
rosetta—3d—rendering
rosetta—spam—filter
serrano—kalman—filter
vitis—convolution

stencil3d
hotspot—0—baseline
hotspot—1—tiling
hotspot—2—pipeline
hotspot—3—unroll
hotspot—4—doublebuffer
hotspot—5—coalescing
hotspot—6—multiddr
kmeans—5—coalescing
kmeans—6—multiddr
knn—5—coalescing
cfd_step_factor_2_pipeline_0

lc_mgvf_0_baseline_0
lud_2_coalescing_0
pathfinder_1_tiling_0
pathfinder_3_unroll_0
srad_0_baseline_0
srad_1_tiling_0
srad_2_pipeline_0
srad_4_double_buffer_0
srad_5_coalescing_0
vitis—tsp

gemm—blocked
md—knn
stencil2d
viterbi
backprop—0—baseline—forward
backprop—1—tiling—back
backprop—1—tiling—forward
backprop—2—pipeline—back
backprop—2—pipeline—forward
backprop—4—doublebuffer—forward
knn—1—tiling
knn—2—pipeline
cfd_step_factor_0_baseline_0

dilate_0_baseline_0
lud_0_baseline_0
pathfinder_0_baseline_0

Conclusions

Leveraging the above results could be beneficial for

- Pruning the design space to some points around the optimal designs for the most similar kernels.
- Construct models that predict the latency and resources of new kernels, based solely on the source code features.

References

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

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2019).
Bert: Pre-training of deep bidirectional transformers for language understanding.

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., and Zhou, M. (2020).
Codebert: A pre-trained model for programming and natural languages.

Jurafsky, D. and Martin, J. H. (2021).
Speech and language processing, 3rd Edition.

Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L., and Stoyanov, V. (2019).
Roberta: A robustly optimized bert pretraining approach.

Reagen, B., Adolf, R., Shao, Y. S., Wei, G.-Y., and Brooks, D. (2014).
Machsuite: Benchmarks for accelerator design and customized architectures.
In *2014 IEEE International Symposium on Workload Characterization (IISWC)*, pages 110–119.

Singh, A. (2022).
Hugging face: Understanding tokenizers.