Autotuning CUDA: Applying NLP Techniques to LS-CAT

Lars Bjertnes, Jacob O. Tørring, and Anne C. Elster

Norwegian University of Science and Technology (NTNU), Trondheim, Norway
lars.bjertnes@outlook.com, {jacob.torring, elster}@ntnu.no

Abstract. The abstract relation between hardware parameters and program performance makes setting program parameters a difficult task. Without autotuning, software can miss low-level optimizations, resulting in lower performance. Traditionally, time-consuming trial and error search methods have been the staple of autotuning. Applying Natural language processing (NLP) based machine learning (ML) methods to source code as a means to perform autotuning-oriented tasks is a growing topic. Earlier research has, with success, performed a range of different autotuning tasks using multiple source code languages. However, most of the source code data is CPU-oriented, with very little GPU code. The LS-CAT (Large-Scale CUDA AutoTuning) dataset [BTE21] uses CUDA GPU-based kernels and generates a dataset to perform thread-coarsening.
This paper implements several custom NLP-ML pipelines to evaluate ML-based thread-coarsening using the LS-CAT dataset, and a custom scoring function to find the performance impact for any choice. Several model configurations were able to beat both random choice, 0.9400, and only selecting the largest thread-block (1024), 0.9437. Finally, the best model achieves a score of 0.9483, giving an average performance increase and speedup of 0.49 percent over the largest thread-block. Implementing self-attention mechanisms proved to counteract over-fitting, while a multi-label based learning task outperformed other approaches. Compared to previous datasets [Cum+17], the LS-CAT dataset's higher thread-coarsening precision gives a more precise evaluation of the model's performance. The *inst2vec* embedding used in earlier works was unable to correctly parse the CUDA LLVM IR tokens, resulting in high data loss. Approaches to addressing this, and other ideas for future work, are also included.

Keywords: Natural Language Processing · Autotuning · CUDA

1 Introduction

As hardware variation and complexity increased, software is increasingly struggling to keep up with the specificity required to have full system utilization. Low hardware utilization created a high gap between actual program performance and theoretical program performance. The cause is a lack of low-level optimizations, which needs hardware taken into account. There is usually a complex relationship between a specific program parameter and the total change in performance. This means that an extensive search through the parameter space is required in autotuning. Searching through all possible legal combinations of parameters is a highly time-consuming process, as for each time search step, the autotuner compiles and executes the program. Then, based on the results, the subsequent search step is performed until an optimal program variation is found. A better alternative would have an autotuner that can set better parameters without searching, compiling, or executing the program.

Machine learning is a method that is well suited in situations where there is an abstract relationship between the data points, and where there is a large enough dataset and enough data processing power [BH19]. Autotuners using source code-based ML methods require a dataset of source codes and program results based on different parameters. Earlier attempts at doing machine-learned autotuning, range in the task performed, source code language, and dataset. Some of these attempts focused on attempting to perform thread-coarsening on an OpenCL dataset created by *end2end-dl* [Cum+17]. This dataset is lacking in size and general representation of source code. With that in mind, the LS-CAT project [BTE21] created a CUDA-based dataset. The LS-CAT dataset consists of CUDA source code kernels and their runtime data. LS-CAT has more thread-block sizes, semi equivalent to thread coarsening level, and significantly more source code samples. This paper is based on [Bje21] which was written in collaboration with the other two authors.

The primary goal of our work is to apply machine learning and NLP, natural language processing techniques to LS-CAT, and using ML-model selected thread-block sizes to increase the performance. Additionally, we evaluate the impact of both ML-attention-mechanisms and *inst2vec* [BJH18] on the machine-learned model's performance. To our knowledge, this is the first implementation of an end-to-end machine learning pipeline, designed for CUDA source code data. In particular, this is the first implementation applying ML NLP techniques to our LS-CAT dataset, and as far as we know. Our attempt at using the *inst2vec* embedder with CUDA LLVM IR tokens is also novel.

Our approach is also the first to outperform both random choice (0.94) and best default option (0.9437) on a large CUDA-based dataset, with the best model configuration scoring 0.9483. Our findings indicate a generalized learning process, not memorization, implying that CUDA source codes have learn-able abstract features.

The paper is structured as follows: Section 2 presents autotuning and our earlier LS-CAT project as well as the related work for the paper. Section 3 describes the embedding pipeline, while Section 4 describes the ML model. In Section 5 we show the results from some key model configurations. In Subsection 5.1 we discuss and evaluate the results. Section 6 draws a conclusion from our work with outlines for future work.

2 Background and Related Work

In this section we will introduce and discuss previous related work on Autotuning, CUDA, and our LS-CAT CUDA Autotuning dataset, including how to create NLP models to autotune code.

Autotuning is a tool to optimize programs for the underlying hardware given the parameters of the program. These parameters are semantically invariant, but can change the way the program runs, including the extent to which it uses cache, the order of operations, and more. We can use autotuning techniques to generalize the process of finding a good combination by searching through the parameter space. The autotuning process works by initializing the parameters, compiling the program for the targeted machine, running the program, and then measuring the results. The program keeps a record of each combination of parameters' performance on the hardware. We repeat this process until we find a satisfactory result. However, each run of the program can potentially have a high run time. Thus, evaluating a large search space of parameters can therefore be a very time-consuming process. We therefore introduce in this paper a method to predict optimal configurations for a program directly through source-code, to skip the time-intensive process of testing out each configuration.

CUDA [Bil] is a programming environment provided by Nvidia for GPGPU computing on their graphics cards. CUDA functions that run on the GPU are called kernels. Functions are either marked as global if run from the system, or device (GPU) if called from the global kernel. The global kernel needs a block parameter and grid parameter, which is a three-dimensional representation of a collection of threads. Each block should be divisible by 32 (known as warp size) as a warp executes all 32 thread simultaneously (or idle some of them). A block can at most run 1024 threads, which each can do one computation, at the same time. The optimal number of threads per block is not always 1024. E.g., several smaller blocks would have more unhindered register access.

2.1 end2end-dl/deeptune

The motivation for *end2end-dl* [Cum+17] was replacing manually crafted heuristics with a machine-learned model which formulates heuristics based on source code. The dataset consists of 17 individual kernels, executed with different hardware systems and either on the GPU or CPU, and with 6 separate thread coarsening factors. The source code is stored as raw source code, but each line code is turned into a sequence of tokens, which are then turned into embeddings by the model. The model itself is a combination of LSTM cells, which are good at capturing sequenced information and is often used for learning language features in a machine learning context.

They showed that their solution could outperform machine-learned methods relying on human-designed heuristics and features by 14%. *Deeptune* approached it as a multi-classification problem, and their competitor as several binary classifications. Both models scored low when doing thread coarsening on the NVIDIA GPU, most likely due to the small number of training samples.

2.2 NCC

NCC [BJH18] tries to create a general machine-learned model to do classification and regression on source code of different origin languages. However, NLP does not take into account the nature of code structure, and *NCC* proposes their embedding *inst2vec* to take the specificities of source code into account. The general tasks they want to accomplish are device mapping, algorithm classification, run-time prediction, and thread coarsening.

By making their model use the IR, the model should work on languages that can create an IR without creating extra language support themselves. This also made it possible to use source code from many different sources. For example, the code embedding *inst2vec* uses a skip-gram method on the IR lines. This is then combined with the contextual flow graph, which represents how different parts of the code depend on each other. *NCC* then uses a series of recurrent neural networks (RNN), for the model itself.

The setup *inst2vec NCC* scored higher than other models for algorithm classification, at around 95% accuracy. They also outperformed *deeptune* on both device mapping and thread coarsening, using the same dataset. *inst2vec NCC* showed that a combination of using both the dependencies or flow and the sequenced nature of code could yield better results than focusing on just one aspect.

2.3 LS-CAT

The goal of our LS-CAT project [BTE21] was to increase available GPU source code for machine-learned autotuning. While earlier works focused on OpenCL GPU code, there seemed to be no attempts at creating a large CUDA-based dataset. Our project used publicly available source code aggregated from GitHub. We reformatted each project into a collection of executable CUDA kernels, which were executed with a range of different thread block sizes and matrix sizes. The CUDA automatic thread block size tool was evaluated, but lacked sensitivity to matrix sizes. LS-CAT produced around 19 683 kernels, with 20 thread-block sizes, of which 16 are one dimensional. The increased amount of source code could hopefully make automatic thread coarsening possible.

In the LS-CAT dataset for the Nvidia T4, the 1024 thread block size was the superior choice and performed on average 94.438% from the optimal, meaning around a 5.56% speedup can be achieved from always picking the optimal choice. A machine-learned model needs to score higher than 94.438% to give any speedup at all.

3 Embedding pipeline

Any machine learning process consists of several steps. Each step is dependent on the previous and is therefore done in a chronological, often iterative process. The three first sections detail transforming raw data into ML readable data. In the first section, the raw source code data is transformed to an intermediate representation (IR). To transform the IR to numerical data, two different methods *inst2vec* and FastText are tested in the Sections 3.2 and 3.3. Lastly, the regression model is described in Section 4.1.

3.1 Source Code to Intermediate Representation

While pure source code could be used more or less as-is for machine learning, with some small conversions to numeric values, using the IR, intermediate representation would be far superior. In addition, the intermediate representation has many advantages when generalizing the code. Firstly, the variable names are all standardized to simple register references, which reduces the variance between each kernel and should make actual distinctions more easily identifiable. The second advantage of using IR is that the machine-learned model would be source code independent and utilize all source codes that could be transformed to the same IR. CUDA has its intermediate representation, *PTX*, that is only used internally in the NVCC compiler. With the correct settings, the *PTX* file can be extracted.

In the related works section, most of the previous attempts made use of the *inst2vec* pipeline. This pipeline requires Clang LLVM IR, which CUDA can be compiled into. However, compiler linking can be a tedious process, especially when using unfamiliar modules. The Clang CUDA guide was therefore followed closely [Pro]. This guide, however, was not enough to seamlessly create IR from CUDA, probably due to some version or path issues, and some modifications had to be made. Some kernels were unable to be transformed into IR. In total, 19540 out of 20257 were transformed or 96.46%.

3.2 The *inst2vec* Pipeline and *NCC*

The *inst2vec* pipeline is used in several recent papers [Bra+20] [Cum+20], to create embeddings based upon LLVM IR, and is trained using a range of different source codes. Since the data embedding is one of the first steps taken in an NLP context, the efficiency of this *inst2vec* method needs to be evaluated as early as possible.

The *NCC* project provides a pre-trained *inst2vec* model for the embedding process, but it is possible to train the embedder with custom data. Training a new *inst2vec* model with custom data was not relevant for our project for two main reasons. First, all the data should be conserved and not exposed to the model to avoid any form of preemptive over-fitting that could occur. Second, the training itself is a time-consuming process. This would delay all further development and drastically halt any project progress.

While the *inst2vec* was not trained on the CUDA LLVM IR codes, it was still capable of embedding parts of the code. Around 55% of code was not turned into embeddings. In comparison, the OpenCL kernels used initially had approximately 12-13% of non-embedding code. It was neither apparent what kind of data was missing, nor how vital these statements might be. We adapted the *NCC* model as a classifier of the dataset, as we regarded it as one of the more straightforward tasks to complete and evaluate.

3.3 *FastText* Embedding

To evaluate if the *inst2vec* data loss did indeed pose a bigger problem than the potential benefit of adopting the pipeline, independent text embedding had to be tested out.

For this purpose, the tool *FastText* [Boj+16] was chosen. *FastText*, developed by Facebook, has some key features that make it stand out. It has the flexibility of creating custom length vectors based on input – the ability to use both the training methods skip-gram and a chosen bag of word. There are also additional settings, such as learning rate, that can be easily modified. This offers a lot of options, as different vector representations of the kernels could be tested out. The perhaps the most notable feature of *FastText* is that unseen data can also be represented by the embedding process, which was an issue with *inst2vec*. This new unseen data would vectorize, where text with similar semantic and syntactic values turns into similar vectors. A custom text preprocessing pipeline was created to generate *FastText* training data. This could be reused to create a vector representation of any LLVM IR image.

We make the training process significantly faster by storing all the embeddings, as the generation process is only done once. However, this does come with a potential downside, as the embedding layer is completely "frozen" for the entire training sequence. The embedding is therefore unable to "learn". In this case, the last layers and parameters should be able to take the weight of adjustments needed, as shown in [Tam+21].

4 ML model design for LS-CAT

The *NCC* model, which was tested out previously, is LSTM based, and our new model should also be based on LSTM modules as this would make a better comparison between the embedding techniques instead of the later parts of the model.

Our model as shown in Fig. 1, is structured such that the *FastText* embeds are fed directly into the LSTM module. However, the outputted tensor has three dimensions, and both the embedder and dense linear layer works with two dimensions. To solve this conflict, the outputted tensor is transformed. This new tensor is then combined with the matrix information using the embedding module. The resulting tensor goes through a series of linear layers, with each layer reducing the number of neurons until the final layer reduces the tensor

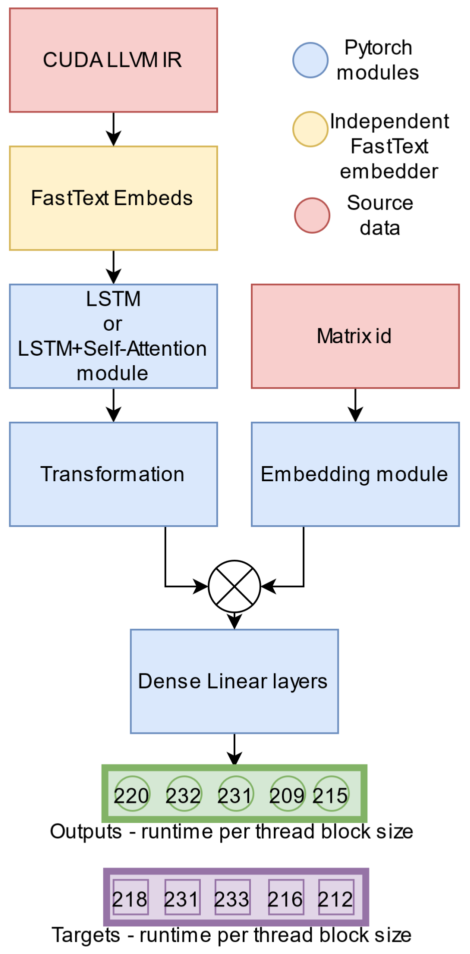
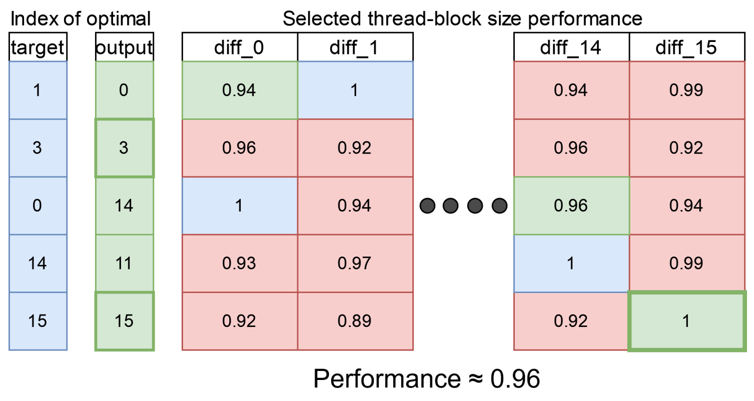


Fig. 1: Our model setup

to the target shape. Another more advanced model was also made, an LSTM Encoder-Decoder with self-attention, the attention mechanism.



Index of optimal		Selected thread-block size performance			
target	output	diff_0	diff_1	diff_14	diff_15
1	0	0.94	1	0.94	0.99
3	3	0.96	0.92	0.96	0.92
0	14	1	0.94	0.96	0.94
14	11	0.93	0.97	1	0.99
15	15	0.92	0.89	0.92	1

Fig. 2: Evaluating the performance of the regression model

4.1 Regression Oriented Learning methods

Regression techniques can be used for this problem, since the score of each thread-block can be expressed as discrete numbers and not just an optimal choice. A matrix IR combination can be given as input to the model, which will produce a series of discrete numbers, one for each thread-block. A prediction would be picking the optimal value, as that number would be linked directly to a specific thread-block size. Alternatively, the model could output a single digit for the input of matrix, IR, and thread-block. This setup would, however, require multiple predictions and results in comparison to finding the optimal thread-block size. The input to the model should also be based on a normalized distribution. It is therefore necessary to transform the target values, using a transformation function. In this case, a modified Softmax function, was applied to the dataset target values.

The target values are now all floating in the range of 0-1, which makes it possible to utilize BCE loss, not in the sense of binary cross-entropy loss, but rather multi-label loss. Each thread-block size can be described as being a deviation from the optimal or partly being the optimal class. The learning task measures the performance by letting the output layer select a potential thread-block size and evaluating the average performance of the choices made. This process is illustrated in Fig. 2.

Selected thread-block sizes are marked as green, the overlap with the optimal thread-block size is marked with a border, non-selected optimal sizes are marked blue. As the model is not explicitly trained to identify the best block size but rather good options, measuring accuracy is not essential.

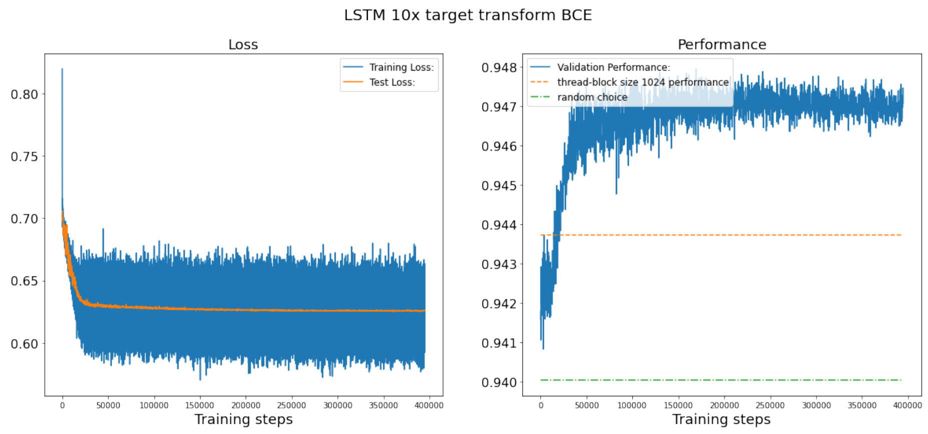


Fig. 3: The BCE LSTM model

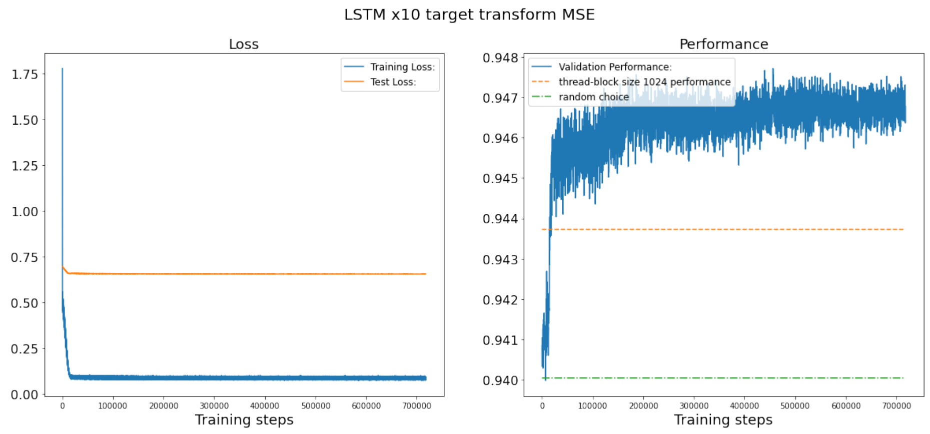


Fig. 4: The MSE LSTM model

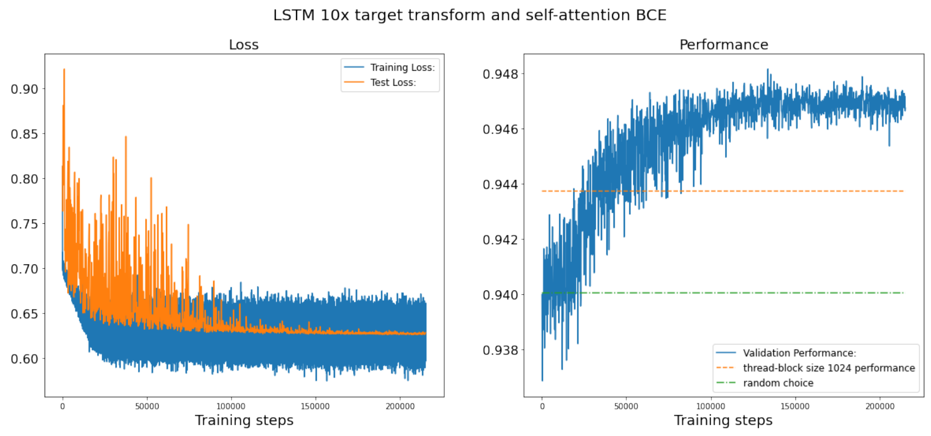


Fig. 5: The BCE LSTM model with self-attention

5 Results

The model based on just LSTM was trained using both BCE multi-label loss and MSE loss, seen in the Figs. 3-4. The performance graphs are somewhat similar, however the loss graphs are very different. In the case of MSE loss, there is a clear case of over-fitting based on the discrepancies between training and testing loss. The BCE loss graph has high oscillation for the training loss, but the test loss remains stable in the middle.

The self-attention based model was also trained using both BCE multi-label loss and MSE loss. The MSE loss made the model behave similar to the earlier non attention based model. The BCE loss, seen in the Fig. 5, has its own characteristics. The gain in performance over time corresponds to the drop in test loss, while the early test loss has a high level of oscillation.

Overall the best configuration was the multi-label loss with a twelve exponent, self-attention, and 256 batch size. Scoring a performance of 0.9483, this exact performance is an outlier and only case of the self-attention networks performing better than those without.

5.1 Analyzing impact of embeddings

The choice of embedding method and embedding sophistication proved to have a significant impact on overall performance in all learning tasks, [BJH18]. Due to the LS-CAT dataset consisting of CUDA LLVM IR, the *inst2vec* method was unable to translate a sufficient amount of tokens. More than half of the IR was lost during the embedding process using the *inst2vec* embedder. Therefore, the sophisticated *inst2vec* embedder had to be replaced with a simpler skip-gram-based embedder. The alternative embedder outperformed the *inst2vec* embedder, most likely due to the loss-less embedding process. If the *inst2vec* embedder were to be modified to handle more CUDA LLVM IR tokens and have the same token loss rate as in its original case of around 12 percent, *inst2vec* might perform better than the simpler skip-gram *FastText* embedder.

To accommodate for the CUDA LLVM IR tokens, the *inst2vec* embedder has to be modified, or another complex embedder could be created using the same principles as those used in the *inst2vec*.

5.2 LS-CAT ML Models Results

Overall both the regression-based tasks outperformed the choice of relying solely on the thread-block size 1024. The second model consisting of a self-attention module between the LSTM encoder decoders performed somewhat worse than the pure LSTM model. However, the semi adversarial behavior of the encoder-decoder attention, where adjustments in the encoder or attention increase the training difficulty for the decoder, increased the generality of the model. In the case of the self-attention with BCE, multi-label loss, the Fig. 5 show these three qualities: The adversity effect is apparent in the loss graph. The test loss is the average of the training loss. The validation performance has a high degree of overlap between the training loss training step-wise. Combined, all of these factors would indicate an actual learning process and not over-fitting. While not scoring higher than the pure LSTM model, machine learning has to create general model solutions to complex problems, and this was displayed to a much higher degree in the self-attention-based model.

The regression method using MSE loss seen in Fig. 4, displayed the highest amount of over-fitting, and the validation training loss hovered high above the training loss.

5.3 LS-CAT ML Model Architecture Variations

Several variations in model dimensions and combinations of loss types were tried out. At the same time, increased model complexity may increase model performance at the cost of training performance and potential over-fitting. Except for the dense output layers, the input size determined directly or indirectly all the other layer's parameter count. Input sizes larger than 80 incurred a significant training speed reduction, and input sizes above 240 would be unfeasible time-wise. No increase in performance was observed at an embedder input size larger than 40. If the models did not use this self-attention model, the input size would not dictate all the dimensions, and different values for sizes, could be tested.

Model depth is the number of layers the model uses in its internal structure. The depth is one way to increase a model's ability to represent abstract relationships between the data and internal data structures. The models relied on 5-10 linear dense layers. Any more layers showed no increase in model performance. This was also the case for an increase in the amount of LSTM layers or amount of stacking. The likely reason these potential improvements failed was the increased distance between the target parameters and the crucial first layer of LSTM cells. The relation between these parts would be a lot more abstract with the added model depth. The multi-target classification was judged not to be a feasible solution to the selection of adequate thread-block sizes

5.4 Evaluation of our LS-CAT Dataset

To evaluate our LS-CAT dataset, one could start looking at the performance of the models or compare the results achieved at LS-CAT with the results from the [Cum+17] OpenCL dataset. However, comparing the results from each dataset would be imprecise due to the difference in select-able thread-block sizes or thread-coarsening levels. The earlier works dataset has only six levels, compared to our LS-CAT dataset, sixteen levels.

Fewer possible choices result in a higher degree of distinction between the choices, and this distinction might both make thread-coarsening appear more lucrative and also easier to perform. The different results from each dataset can therefore not be compared fairly.

For these reasons, our LS-CAT dataset can not be judged by making a simple model result comparison with this earlier dataset. Instead, seeing how the models performed on our LS-CAT, compared to random choice, and relying solely on the best block size would be better. There was enough semantic information in the kernel IR, for a model (0.9483) to outperform both random choices (0.94) and only select the largest thread-block size (1024) (0.9437). Several models gave results indicating strongly that a learning process and not memorization was performed. Also, a model relying on only the matrix run-time and name gave results similar to random choice. The presence of this semantic information implies both that the LS-CAT dataset is valid and that a kernel's source code has sufficiently distinct information that makes abstract learning tasks possible to perform. Taken this into account, the LS-CAT dataset works.

6 Conclusions and Future Work

Performance tuning tasks are difficult to perform effectively manually. However, without tuning, software ends up lacking key low-level optimizations, causing a drop in overall performance. Furthermore, current autotuners rely on extensive search processes, which can end up being more time-consuming than time-saving. As shown in this paper, machine learning can reduce or almost nullify the search process, and in turn, create better autotuners.

Our goal of having a model select thread-block sizes to increase the performance was met. With the best configuration scoring 0.9483, an increase of 0.49 percent over the largest block. The multi-label loss-based regression proved to be the most efficient. While the self-attention-based models had lower overall performance, the network's self adversarial properties increased the learning difficulty and increased the generality.The self-attention model with multi-label loss indicated more strongly a learning process than the other models. Overall the results for the different models indicated that the semantic information in the source code was enough to learn the abstract relationship between relative performance, source code, thread-block size, and matrix size. This would strengthen earlier claims that source code can be used to learn abstract program features.

The embedding process of *inst2vec*, which was deemed significant in earlier projects, had issues with data loss when transforming the intermediate representation to numeric data. The CUDA-based LLVM IR had around five times the data loss as the OpenCL LLVM IR, which the earlier projects utilized. The solution to this was implementing skip-gram-based methods using the tool *FastText*, as this is a fully lossless process. The potential benefit of a more complex embedding process that does not also include high data loss remains, therefore, unexplored on LS-CAT.

Current and Future Work

Ideally, an embedding process more adapted to source code and handling a broader range of tokens should be developed. This sophisticated embedder could be more fairly tested in comparison with lossless skip-gram. This could be potentially done by improving upon the *inst2vec* pipeline and its interpretation of LLVM IR tokens. *inst2vec's* biggest issue revolved around data-loss. There are several ways to mitigate this data loss. For instance, improve the handling for unseen tokens, rather than just dropping them. Or instead, increasing the amount of seen tokens during training would directly diminish data loss. As a last alternative, one could have a custom user-defined parser for the unseen tokens. This user-defined parser would avoid the potential issue and not put too much work on the developers.

Acknowledgements

The authors would like to acknowledge the Faculty of Information Technology, Mathematics and Electrical Engineering and the Department of Computer Science at NTNU for their PhD stipend and support of our HPC-Lab that facilitated the development of this project.

References

[Boj+16] Piotr Bojanowski et al. "Enriching Word Vectors with Subword Information". In: *arXiv preprint arXiv:1607.04606* (2016).

[Cum+17] Chris Cummins et al. "End-to-End Deep Learning of Optimization Heuristics". en. In: *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. Portland, OR: IEEE, Sept. 2017, pp. 219–232. ISBN: 978-1-5090-6764-0. DOI: 10.1109/PACT.2017.24. URL: http://ieeexplore.ieee.org/document/8091247/ (visited on 10/31/2020).

[BJH18] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. "Neural Code Comprehension: A Learnable Representation of Code Semantics". en. In: *arXiv:1806.07336 [cs, stat]* (Nov. 2018). arXiv: 1806.07336. URL: `http://arxiv.org/abs/1806.07336` (visited on 10/31/2020).

[BH19] Tal Ben-Nun and Torsten Hoefler. "Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis". In: *ACM Comput. Surv.* 52.4 (Aug. 2019). ISSN: 0360-0300. DOI: `10.1145/3320060`. URL: `https://doi.org/10.1145/3320060`.

[Bra+20] Alexander Brauckmann et al. "Compiler-based graph representations for deep learning models of code". en. In: *Proceedings of the 29th International Conference on Compiler Construction.* San Diego CA USA: ACM, Feb. 2020, pp. 201–211. ISBN: 978-1-4503-7120-9. DOI: `10.1145/3377555.3377894`. URL: `https://dl.acm.org/doi/10.1145/3377555.3377894` (visited on 10/31/2020).

[Cum+20] Chris Cummins et al. "ProGraML: Graph-based Deep Learning for Program Optimization and Analysis". en. In: *arXiv:2003.10536 [cs, stat]* (Mar. 2020). arXiv: 2003.10536. URL: `http://arxiv.org/abs/2003.10536` (visited on 10/31/2020).

[Bje21] Lars Bjertnes. "Applying Natural-Language-Processing-Based Machine-Learning Techniques to our Large Scale CUDA AutoTuning Dataset". MA thesis. Trondheim, Norway: Dept. of Computer, Information Science, Norwegian University of Science, and Technology (NTNU), 2021.

[BTE21] Lars Bjertnes, Jacob O. Tørring, and C. Anne Elster. "LS-CAT: A Large-Scale CUDA AutoTuning Dataset". In: *IEEE International Conference on Applied Artificial Intelligence (ICAPAI 2021)* 31 (May 2021).

[Tam+21] Thierry Tambe et al. *EdgeBERT: Sentence-Level Energy Optimizations for Latency-Aware Multi-Task NLP Inference.* 2021. arXiv: `2011.14203 [cs.AR]`.

[Bil] Sebastian Jodłowski Bill Fiser. *BEST PRACTICES WHEN BENCHMARKING CUDA APPLICATIONS.* URL: `https://developer.download.nvidia.com/video/gputechconf/gtc/2019/presentation/s9956-best-practices-when-benchmarking-cuda-applications_V2.pdf`. (accessed: 01.10.2020).

[Pro] LLVM Project. *Compiling CUDA with clang.* URL: `https://llvm.org/docs/CompileCudaWithLLVM.html` (visited on 01/24/2021).