The Pennsylvania State University
The Graduate School

EXPLORATION OF MACHINE LEARNING BASED ACCELERATION METHODOLOGIES

A Dissertation in
Computer Science and Engineering
by
Huaipan Jiang

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

August 2022

The dissertation of Huaipan Jiang was reviewed and approved by:

Mahmut T. Kandemir
Distinguished Professor of Computer Science and Engineering
Dissertation Advisor
Chair of Committee

Mehrdad Mahdavi
Professor of Computer Science and Engineering

John Sampson
Professor of Computer Science and Engineering

Nikolay V. Dokholyan
Professor of Chemistry and Biomedical Engineering

Chita R. Das
Distinguished Professor of Computer Science and Engineering
Department Head

Abstract

Machine learning (ML) based approaches have recently achieved great successes across diverse application domains, including but not limited to, images processing, drug discovery, natural language processing, gaming, and autonomous driving. Unlike the conventional approaches, such ML-based approaches study the patterns and features from the collected dataset and try to learn the optimal solution based on the extracted features. Ideally, the model will be more accurate if it is trained on a larger dataset with more features learned. The state-of-the-art ML designs can beat humans due to the substantial growth in the computational power of modern hardware architecture and the availability of huge datasets to learn from. For example, a state-of-the-art GPU cluster today can train an ML models with millions of images and millions of parameters within several hours. However, since not everybody has access to such large clusters, the problem of reducing execution latency of ML models on limited compute, memory and storage resources becomes a critical research item.

Unlike most existing studies that employ ML-based approaches to primarily improve the accuracy of the task at hand, the objective of this thesis is to boost the performance of the existing approaches with ML techniques from different aspects. In general, to solve a problem, the user runs an application (algorithm) on the given hardware (system) with the given input data. Our thesis aims to reduce, using ML, the execution time of the given task via data reduction, algorithmic improvement, and system-level optimization, while maintaining an acceptable accuracy. For each of these aspects (data, algorithm, and system), we study example tasks which can be accelerated with guidance from ML. We also report extensive experimental data showing the effectiveness of our proposed approaches.

Table of Contents

List of Figures

List of Tables

Acknowledgments

With this opportunity, I would like to show my great gratitude to my advisor, Dr. Mahmut T. Kandemir, for inviting me to Penn State and guiding me how to do the research.

I would also like to thank my committee members, Dr. Mehrdad Mahdavi, Dr. John Sampson and Dr. Nikolay V. Dokholyan. All of them has been constant support for me during my Ph.D. I enjoyed every and each discussion with them.

Apart from my advisors and committee members, I also want to thank many other labmates, friends, and department staffs: Dr. Haibo Zhang, Dr. Prasanna Venkatesh Rengasamy, Sandeepa Bhuyan, Ziyu Ying, Tulika Parija, Dr. Ashutosh Pattnaik, Dr. Prashanth Thinakaran, Dr. Jashwant Raj Gunasekaran, Dr. Anup Sarma, Prof. Xulong Tang, Cyan Misra, Dr. Shulin Zhao, Sonali Singh, Dr. Jihyun Ryoo, Dr. Chun-Yi Liu, Mengran Fan, Morteza Ramezani, Chia-Hao Chang, Yunjin Wang, Dr. Chen Sun, Aakash Sharma, Vivek Bhasi, Rishabh Jain, Yilin Feng, Kang Yan, Yihe Huang, Yingtian Zhang, Erin Ammerman, Katelen Bair, Amanda Collins, Olivia Ewing, Amy Hasan, Jennifer Houser, Cindy Milliron, John Orfanoudakis, Austin Powell, Annie Royer, etc.

Thank you all

This material is based upon work supported by the National Science Foundation (NSF) under grant 0821527, 1439021, 1439057, 1626251, 1629129, 1629915, 1763681, 1908793, 1931531, 2008398, 2028929 and 1955815, National Institutes of Health (NIH) grant 1R01AG065294, 1R35GM134864, 1RF1AG071675, National Center for Advancing Translational Sciences, NIH, grant UL1 TR002014. The content of the publications in this thesis are solely the responsibility of the authors and does not necessarily represent the official views of NSF or NIH.

This thesis contain materials from “Huaipan Jiang, Anup Sarma, Jihyun Ryoo, Jagadish B Kotra, Meena Arunachalam, Chita R Das, and Mahmut T Kandemir. A

learning-guided hierarchical approach for biomedical image segmentation. In 2018 31st IEEE International System-on-Chip Conference (SOCC), pages 227-232, Sep. 2018.", "Huaipan Jiang, Mengran Fan, Jian Wang, Anup Sarma, Shruti Mohanty, Nikolay V Dokholyan, MehrdadMahdavi, and Mahmut T Kandemir. Guiding conventional protein–ligand docking software withconvolutional neural networks.Journal of Chemical Information and Modeling, 2020.", "Huaipan Jiang, Anup Sarma, Mengran Fan, Jihyun Ryoo, Meenakshi Arunachalam, Sharada Naveen,and Mahmut T Kandemir. Morphable convolutional neural network for biomedical image segmentation.In2021 Design, Automation Test in Europe Conference Exhibition (DATE), pages 1522–1525. IEEE,2021", "Huaipan Jiang, Haibo Zhang, Xulong Tang, Vineetha Govindaraj, Jack Sampson, Mahmut TaylanKandemir, and Danfeng Zhang. Fluid: a framework for approximate concurrency via controlled dependency relaxation. In Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI), pages 252–267, 202", and "Huaipan Jiang, Jian Wang, Weilin Cong, Yihe Huang, Morteza Ramezani, Anup Sarma, Nikolay V Dokholyan, Mehrdad Mahdavi, and Mahmut T. Kandemir. Predicting Protein–Ligand Docking Structure with Graph Neural Network. Journal of Chemical Information and Modeling 2022".

Dedication

To everyone and everything helped me during my Ph.D.

Chapter 1

Introduction

The machine learning (ML) based approaches have helped application writers to improve their codes in many areas during the recent decades, which include Biomedical image processing [1], drug discovery [2], self-driving cars [3], and natural language processing [4]. Such ML-based approaches target mainly on improving the accuracy of the existing methods. However, in many cases, to achieve the desired level of accuracy, application writers build/deploy increasingly deeper neural network (NN) models with lots of parameters, and as such, these models consume a lot of CPU/GPU cycles during training and inference, resulting in being a bottleneck in some cases [5, 6]. Further, the large parameter space can result in enormous memory footprints which make it difficult to implement such ML-based approaches in edge devices or system-on-chip (SoC) platforms. Additionally, in some scenarios (e.g., auto-driving and photo dynamic therapy treatment), the data should be processed in real-time on limited compute resources. Therefore, it may not always be practical or beneficial to replace a given algorithm with a purely ML-based approach; instead, exploring ML strategies to support existing algorithms may be a better alternative. In this thesis, we explore ML methodologies which aim to boost the exist algorithms for different workloads. More specifically, we study how image processing and drug discovery tasks can be boosted with the help of ML.

The prior works on application acceleration can be broadly divided into three main categories. The first category focuses on designing custom systems and hardware platforms to boost the ML workloads. The works in this category [7–9] include GPU-based, TPU-based, FPGA-based and custom accelerators. Within this category, there also exist works [10–13] based on compiler optimizations and task scheduling to complement hardware acceleration. However, hardware customization can be costly and as a result it is not always a viable option. The second category employs approximate computing [14–16] which is based on the idea of using non-exact methods if

the target application can accept less-than-perfect accuracy. The works in this category include attention mechanism [17], quantization [18, 19], and pruning [20, 21]. These works are geared towards reducing execution latency while maintaining reasonable accuracy. Consequently, the most critical issue for this class of approaches is to develop strategies to maintain accuracy during the approximation. In general, it is very difficult for the user to design an optimal approximation strategy for a given application. The third category includes works that are based an application parallelization [22, 23]. Simply put, increasing the number of threads to run an application is expected to reduce execution time. However, beyond a degree of parallelism inter-thread communication and data contention costs dominate, and the gains from parallelism start to diminish.

Based on the observation from the aforementioned approaches, we propose to boos application performance using ML. More specifically, we explore how to use ML to i) reduce the size of the input data, ii) automatically design the approximation strategy for the user, and iii) design efficient ML algorithms to replace the time-consuming conventional approaches. Although we mainly focus on the applications from the image processing and drug discovery domains in this thesis, we believe that ML-guided acceleration of applications can significantly benefit other application domains as well.

In this thesis, we first introduce some example applications that we evaluated in this thesis and then apply ML-based acceleration depending on the tasks these applications are trying to implement. The main contributions of this thesis include:

- We study on a semantic image segmentation task in the biomedical domain and accelerate the existing approaches by reducing the redundant input data. The prior work has proposed a CNN-based custom neural network, called U-net [24], which can distinguish, given an image, between the pixels that belong to the target object and pixels that belong to the background. In this thesis, we remove the redundant computation by first predicting which pixels on the image are useful for the CNN model. More specifically, we only process the pixels that contain the objects and quickly filter the background part of the image to save the overall execution time. Our proposed solution is organized as a hierarchical model which first detects the region of interest on the image and then applies the segmentation operation only on the useful pixels. We present experimental evidence that clearly shows that the proposed approach can expedite U-Net, without any significant drop in accuracy.

- We study the protein-ligand docking problem in drug-discovery [25–29] and design a new algorithm by soliciting help from ML to boost the docking process for a given protein-ligand pair. Existing docking software (including conventional approaches [30–38] as well as ML-based approaches [39–44]) require thousands of iterations of the sampling process. We propose a CNN-based framework to

reduce the number of iterations during the sampling process. Our proposed CNN model can predict if the search process so far has already found enough good poses and, if so, this can lead to early-termination. We also investigate a GNN-based algorithm to generate the docking poses directly without any sampling. Our proposed GNN model takes arbitrary initial docking poses as the input and outputs the refined poses which are expected have lower RMSD to the ground truth pose. With the help of these CNN-based and GNN-based models, the traditional docking software can reduce the execution time while improving the accuracy.

- We study various image classification tasks with approximate computing. Specifically, we implement an eager execution framework which enables the approximate computing for CNN-based applications designed for image classification. This proposed framework can potentially help one reduce the execution latency of CNN, thereby expediting the underlying application that uses CNN. We also propose an performance modeling framework to help the user automatically find the optimal approximation strategy and save the execution time of the image classification tasks.

For each aspect in this thesis, we first introduce the background of the problem and then discuss the implementation of our acceleration strategies. Finally, we present and discuss the experiment results.

Chapter 2

Background

2.1 Semantic Image Segmentation:

Image segmentation is a critical step in many biomedical image processing jobs. One have to classify all pixels on a given image into several categories. where each category indicate a type of the object. For example, in the cell tracking task, the pixels on the input image are divided into foreground (cells) and the background. In the retina layer segmentation [45] task, the pixels from different retina layers will be distinguished by the segmentation software. The semantic segmentation is different from instance segmentation since the former task only classify the category of each object, while for the later, each objects in the same category are distinguished.

There are many traditional image segmentation approaches. One of the most well-known approach is edge detection which include two main steps. The first step is applying the noise removal kernels (e.g., Gaussian) to make the pixels value to be more smooth. In the second step, one can apply the gradient kernels (e.g., Sobel) to calculate the derivatives of each pixel along different directions. A pixel with high total gradient will be considered as the pixel ride on the edge. Some recent studies also propose to detect the edge pixels by selecting the local maximum/minimum pixel value. This approach is also referred as ridge detection [46]. As the neural networks play more and more important role in the image processing, some prior works [47–49] implement CNNs to predict which pixels are belong to the edge. for all the aforementioned edge detection approaches, after the edges are detected, one can connect the edges and then fill the interior area to finally generate the segmentation mask.

Although the edge detection based segmentation approaches can fast locate the edges on the image, they are suffer from the accuracy issues during the edge detection. More

specifically, they will generate some false positive edges for some random objects or miss some edges for the target objects. To improve the accuracy of the segmentation process, many Convolutional Neural Networks have been employed in the biomedical image segmentation problems. Ciresan et al. [50] employs a traditional classification CNN with a sliding-window method on a given image. Here, each pixel is represented by its neighboring pixels. FCN [51] proposed an more efficient model which includes multiple pixel-level prediction modules to predict the category of each pixel. Ronneberger et al. proposed U-net [24] based on FCN. U-net include a down-ward path and an up-ward path of convolution layers which organized a "U-shaped" architecture. The high resolution feature map in the down-ward path are concatenated with the low resolution feature map in the up-ward path. SegNet [52] employs a similar "U-shaped" architecture. The pooling index of the pooling layers are recorded to guide the deconvolution layer. Seyedhosseini et al. [53] employed cascaded hierarchical models on images with binary ground truth, Stollenga et al. [54] applied LSTM [55] structures on the 3D image efficiently, in an attempt to increase parallelism. StarDist [56] predict the shape of each cell object by predicting a "star-convex polygon" for every pixel. This approach improve the accuracy of generating the segmentation results for the crowded cells. However, none of these approaches targets execution time, which is a critical factor in the inference of biomedical image segmentation. Cellpose [57] employed the U-net like CNN with human labeled cell shape as annotation to classify the pixels inside the cells. Although it reduce the execution time when compare to process on the entire image, it requires expensive human labor.

To reduce the execution time of the deep and large CNNs, recent works proposed attention models which allocate the regions of interest (RoI) first on the image. The later segmentation model only focus on the pixels of RoIs. For example, Yolo [58] and Mask R-CNN [59] first employ exist CNN models to generate the feature maps of the image, and then select the RoIs with different shape and scales. Unfortunately, selecting the best region from thousands of candidates is not a straightforward task.

2.2 Protein-Ligand Docking in Drug Discovery

Docking-based virtual screening and pose selection algorithms provided efficient and effective approaches to accelerating the process of drug discovery by circumventing long and expensive trial-and-error assays. Recent studies proposed many protein-ligand docking approaches based on calculating the total energy of all the atoms for a docking complex within the force field. In the traditional docking approaches, there are two main steps involved which is the *sampling* and the *scoring*. Sampling is how to generate the binding poses for the given protein-ligand pair, whereas, the scoring is calculate the binding energy for such binding pose.

During the sampling step, most well-know docking software employ randomly search algorithms (e.g., Monte Carlo). These software include ProDock [38], MCDOCK [37] and MedusaDock [30–33, 60]. On the other hand, some software include GOLD [36] and AutoDock [34] employ a genetic algorithm [61] based approach. Additionally, FlexX [62] employs a fragment-based algorithm [63], which get benefit from the hydrophilic hits. In each iteration of the sampling step, the docking software employ the scoring function to evaluate how likely the poses is stable. Indeed, the docking accuracy is significantly dominated by the reliability of the scoring function. There are mainly two types of scoring functions [64] used in the traditional docking software: *physically-based* [65–71] and *knowledge-based* [72–76]. The former usually calculate the inter action between the atoms in the complex based on the force field parameterization. This force field parameterization might be different in each software. Then, the overall scoring of the complex is calculated by organizing all the interactions. On the other hand, the knowledge-based score functions are based on the statistic results from the previous studies. The limitation of this method is how to reuse such prior knowledge in the new reference state.

In the recent years, several machine learning based scoring functions are also proposed. These work usually take the 3D coordinate of each atom in the complex as the input. These 3D coordinate will be convert to the 3d-grid tensor or graph-based data. These pre-processed data can be accepted by the CNN models or GNN models. The machine learning models will calculate the interaction between the atoms in the input with the convolution operations. After that, several fully-connected layers are implemented to generate the final predict probability which indicate if the input complex is a stable binding or not.

2.3 Approximation Computations

Many recent studies have proposed approximate computation models for improving performance [14] and/or saving energy cost [14, 77] for the applications which can tolerate errors of the data. These approximation models usually forwarding the unfinished component data to execute with inaccurate values. Working with such unreliable data will elide the execution of certain tasks. Another flavor of approximation includes scheduling tasks in parallel under weakened synchronization assumptions [78]). In general, to make the approximate computation approach be viable, the internal approximate version of data should be close enough to the final data. Otherwise, the subsequent task will got significantly wrong results. As the result, the user should set a pre-defined threshold. Start from 1990s, there are lots of substantial works which start the consumer task before the producer task finished [79–84]. Such mechanism is referred as eager execution. Those works include investigating disjoint executions which yields higher parallelism degrees [85], selectively eager execute on

different path of the program flow [86], employing compiler support to enable control speculation, data dependence speculation, and predication [87], and implementing the eager execution opportunities in the deep learning frameworks [88].

2.4 Performance Model

Performance profiling models are critical for understanding the architectural bottlenecks and hotspots in the system and improve the performance. Williams et al. [89] proposed a roofline model to improve the performance for parallel floating-point applications. Hong et al. [90] implemented an integrated performance model for GPUs, which predicts the energy consumption and execution time of a given application. Lv et al. [91] proposed a model checking system targeting multicore platforms with the goal of analyzing the memory behavior and predicting the performance bottlenecks. Stengel et al. [92] employed a cache-memory performance model to quantify the bottlenecks in stencil computations, and Hoefler et al. [93] predicted the performance of parallel applications. Konstantinidis et al. [94] proposed a performance predictor that can automatically estimate the performance of the CUDA kernels running on GPUs. Gast et al. [95] presented a Markovian model that can be used to evaluate applications executing on large-scale heterogeneous system.

Recently, machine learning techniques have been adopt in designing the performance models to predict the behavior of applications running on various types of hardware. Yoo et al. [96] proposed to use ML to automatically identify the hotspot of each function in an application. Yoo et al. [97] also suggested a job status prediction strategy for scientific clusters. Similarly, Kaufman et al. [98] investigated a learned performance model for a compiler targeting Tensor Processing Units (TPUs). Ardalani et al. [99] adapted a two-level ML approach to predict the performance of applications running on GPUs, given the profiling results collected from CPUs. Jaggard et al. [100] implemented a framework to monitor the network performance for ML applications. Wang et al. [101] proposed an approach which creates multiple deep learning models with various hyperparameters and profiled them using the Roofline model. Dev et al. [102] propose power-aware characterization and schedule the kernel using SVM.

Chapter 3

Investigating Acceleration Approaches for Biomedical Image Segmentation Applications by Reducing Data Size

3.1 Introduction

In this Chapter, we introduce how to accelerate an existing application by reduce the size of data with the guide of machine learning techniques. We study on the image segmentation task for the cell tracking, where the segmentation framework will classify the pixels on the image and determine if a pixel belongs to the cell or the background. Here, we focus on accelerating a well-know state-of-the-art image segmentation CNN model "U-net", which is mainly designed for biomedical dataset. We motivate our approaches by observing the redundant computation in the current implementation. Based on our motivation results, we propose two acceleration methods to boost the segmentation process. The first method is a tiling based approach which first tile the images into the chunks and the quickly classify which chunk contains the cells. After that, we efficiently combined the tiles contain the cells and generated the segmentation output. This method is mainly designed for on chip systems since it is easy to implement and only require limited computation and storage resources. In the second method, we combined the idea of edge detection to guide the CNN to generate the segmentation mask. We first detect the edges on the image and then use our morphable convolution framework to classified the pixels on the edges. This approach can significantly reduce the execution time and can be implemented on any platforms. In general, our goal for both methods is reducing the execution time while

Datasets	Pixel-wise OR	Tile-wise OR
PhC-C2DH-U373	6.47%	59.01%
Fluo-C2DL-MSC	6.05%	47.73%
Fluo-N2DH-SIM+	6.30%	65.83%
Fluo-C3DH-H157	6.83%	38.53%
Average	6.41%	52.77%

Table 3.1: Pixel-wise and tile-wise object ratios (OR).

maintain the acceptable level of accuracy.

3.2 Motivation

We motivated our approaches based on the following observations.

- We go over multiple cell tracking datasets and finds that the cells only occupy a small ratio of the entire image. As shown in Table 3.1, there are only 6.41% of pixels in average belongs to the cells cross those datasets. If we tile the image into several chunks
- We find that identifying if the image contains the object is faster and easier than segmenting the entire image. A simple CNN model like AlexNet [103] can accurately identify whether an image contains cells, while we need to employ complex CNN models like U-net to clearly classify each pixels on the image.
- We observe that detecting the edges on an image can be done with simple edge detection kernels. Although they generate low accuracy edges, such edge detection process can be done in a very short time period when compare with segmentation CNNs.

3.3 Tiling based Segmentation Approach

Our first approach for boosting the segmentation process is based on tiling. The entire framework consists two steps. As shown in Figure 3.1, we first use a classifier (Alexnet) to scan all the chunks to detect if any cell objects within a chunk. The chunks are identified as containing cells will be sent to our the segmentation framework (U-net) to generate the segmentation masks. We will finally combine the segmentation masks of all the chunks together to produce the segmentation result of the entire image. Note that, although we use AlexNet and U-net in our approach, it is possible that we use any type of classifier and segmentation networks, depending on the dataset. We propose three tiling strategies in this work.

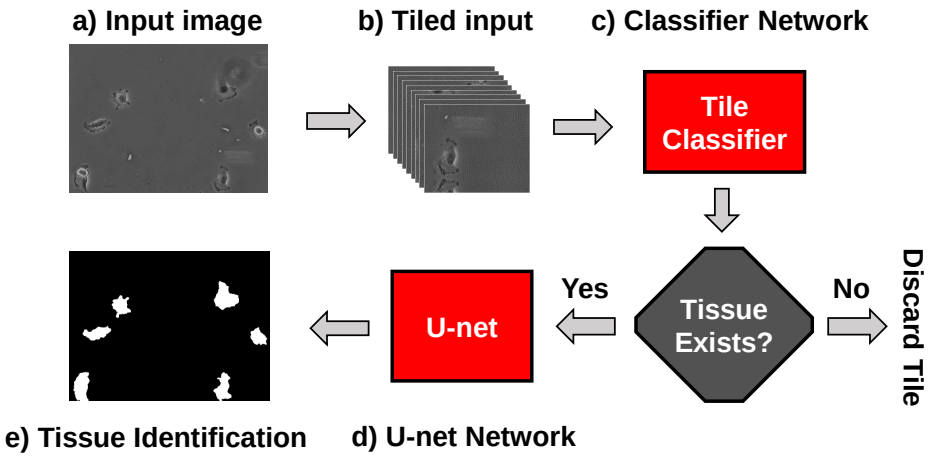


Figure 3.1: High-level view of our tiling-based framework.

3.3.1 Fixed Tiling

The first tiling strategy we propose is *fixed tiling*. In this tiling strategy, as shown in Figure 3.2a, each image is divided into tiles of the same size. It is easy to see that, increasing the tile size will result in fewer tiles to be processed by the classifier (AlexNet) and can result in only a few tiles to be discarded as the likelihood of each larger tile to contain a cell increases. On the other hand, decreasing the size of a tile will result in many tiles to be discarded while increasing overall execution time which is not preferable for overall performance.

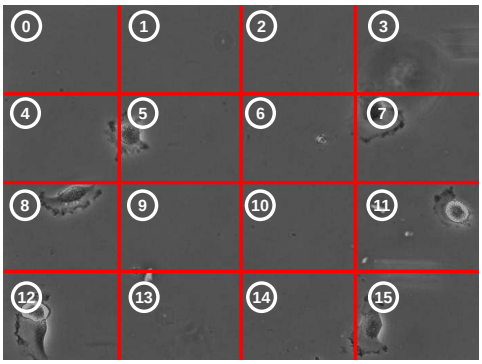
3.3.2 Adaptive Tiling

It is to be noted that, the fixed tiling strategy will face a trade-off challenge. If the tile size is too big, then its behavior comes close to U-net. If on the other hand, the tile size is too small, the overall execution time will increase. Unfortunately, it is not trivial to easily determine an "optimal" tile size to employ, when using the fixed tiling strategy for different input datasets. Motived by this observation, we next propose *adaptive tiling*. As shown in Figure 3.2b, in this second tiling strategy, we first divide each image into large tiles and send those tiles to the classifier. Depending on whether the tile is detected (by the classifier) to contain a tissue or not, each of the larger tiles can be further divided into smaller tiles. These smaller tiles are sent through the classifier once again. And, this process continues in an iterative manner, where, at each step, a larger tile is further divided if it is detected to have a tissue. However, there is a potential issue which can limit the execution speed of the adaptive tiling strategy. The output feature map generated by U-net is smaller in size than its input size. This is because of the shrinkage during the convolution layers. For example, the 3*3 convolution kernel in U-net causes an image size loss of 2 pixels in both height and width. To obtain a segmentation map of size n*n, we need to feed U-net with a

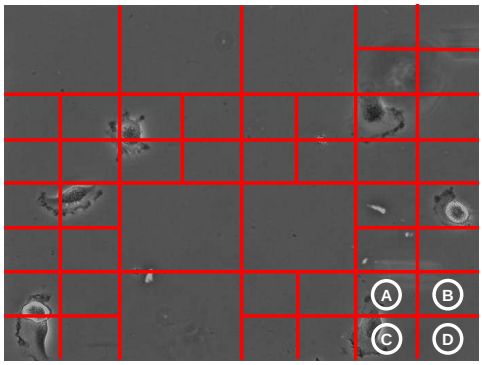
Algorithm 1: Scratching from the scratch map

Input : A scratch map M with size of n by m and all the entries are 0 or 1

Initialization:;
Neighboring_region_list ← ∅;
for *each elements* $M[i, j]$ ***in*** M **do**
    if $M[i, j]$ *is not in any connected regions now* **then**
        call **BSF** start from $M[i, j]$ and get a neighboring region R;
        Neighboring_region_list.push_back(R);
Rectangle_boxes ← ∅;
for *each neighboring region* R ***in*** *Neighboring_region_list* **do**
    $T1$ ← the largest rectangle box of 1s at the up-left of R;
    $T2$ ← the largest rectangle box of 1s at the up-right of R;
    $T3$ ← the largest rectangle box of 1s at the down-left of R;
    $T4$ ← the largest rectangle box of 1s at the down-right of R;
    $T5$ ← The remaining area of 1s in R;
    Rectangle_boxes.push_back(T1);
    Rectangle_boxes.push_back(T2);
    Rectangle_boxes.push_back(T3);
    Rectangle_boxes.push_back(T4);
    Rectangle_boxes.push_back(T5);
Return *Rectangle_boxes*;



(a) This shows the fixed tiling strategy. In this case, the image is divided into 4 by 4 tiles. Among these, tiles 0, 1, 2, 9, 10, and 13 do not contain any part of any cell, while tiles 3, 4, 5, 6, 7, 8, 11, 12, 14, and 15 contain some parts of the cells. Consequently, first group of the tiles will be omitted.



(b) This shows the adaptive tiling strategy. In this case, for those tiles which contain some parts of the cells will be further divided into 2 by 2 smaller tiles. For example, tile 15 in (a) will be divided into tiles A, B, C, and D. Since only A and C contain cells while B and D do not, tile B and tile D will be discarded.

Figure 3.2: Fixed 3.2a and adaptive 3.2b tiling strategies.

padded tile, which is of size (n+184)*(n+184). Figure 3.3 illustrates this issue using an example. Hence, as the number of tiles increases, each tile itself becomes smaller in size; however, the overhead of the total padded area becomes relatively larger and this causes an increase in execution time due to the redundant operations on the overlap areas between the neighboring tiles. Consequently, it is not clear whether the adaptive tiling strategy could generate any better result than the fixed tiling strategy.

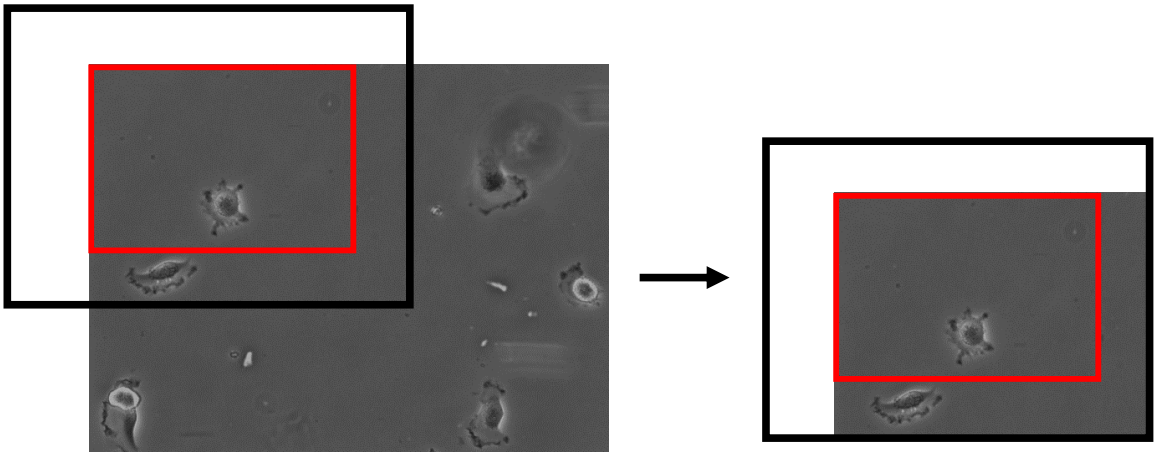


Figure 3.3: To segment the tile in the red box, we feed U-net with the black box as input.

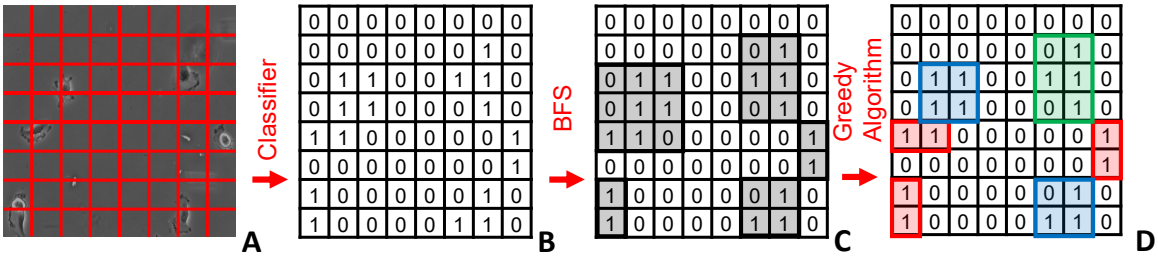
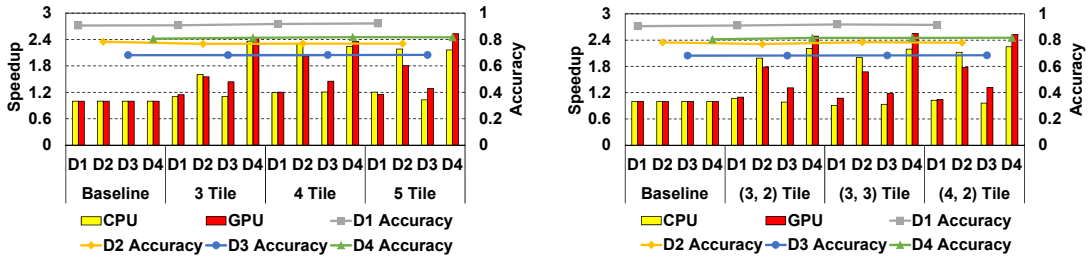


Figure 3.4: This shows the scratch tiling strategy. First, the raw image (A) is divided into 8 by 8 tiles, and the resulting tiles are then sent to the classifier to generate the scratch map (B). After that, we employ BSF to accumulate all the neighboring regions as shown in (C). Finally, we use a greedy algorithm to obtain the rectangle boxes in (D).

3.3.3 Scratch Tiling

To address the above mentioned limitation for the adaptive tiling, we propose a more flexible tiling strategy that scratches the surrounding area of the objects from the image directly. We call this strategy *scratch tiling*. In this tiling strategy, we first divide the image into very fine-grained tiles and then send them to the classifier. The classifier generates a "scratch map" for each image, as shown in Figure 3.4. In the scratch map, an entry '1' indicates that the corresponding tile in the image contains a cell, while a '0' indicates no cell existence in the tile. We can then scratch some rectangle boxes (the colored rectangles in Figure 3.4D) from the scratch map to cover all the 1s in the map, and then send the rectangles boxes to U-net. Unlike the fixed tiling strategy, the adjacent rectangles here can be of *different sizes* depending on the tissue sizes.

To feed all the rectangle boxes containing 1 in the scratch map to U-net, we apply a



(a) Fixed Tiling. **(b)** Adaptive Tiling.

Figure 3.5: Speedup and accuracy results with the fix tiling and adaptive tiling. The bar graph shows speedup, and the line graph shows accuracy. "n Tile" indicates that the image is divided into n-by-n tiles. "(n, m) Tile" indicates that the image is first divided into n-by-n, each tile is further divided into m-by-m sub-tiles if it contains tissue.

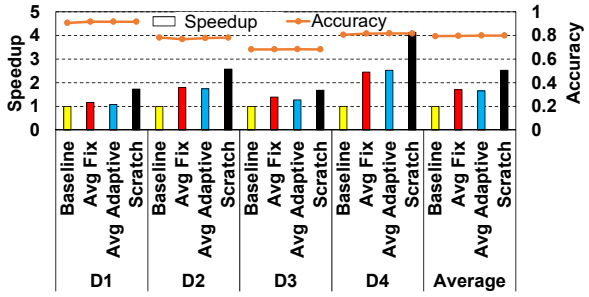


Figure 3.6: Performance and accuracy results with the scratch tiling (in Tesla GPU).

scratching algorithm (presented as Algorithm 1 above). As can be observed from this algorithm, we employ Breadth First Search (BFS) to accumulate all the neighboring regions of 1s. Then, in each region, we use a greedy algorithm to scratch the rectangle boxes from each corner of the regions as large as possible. In the end, we reorganize the results of all the chunks together to obtain the final image segmentation result for the whole image, as we did in the previous two tiling strategies.

3.3.4 Results

We plot the results of all three tiling strategies in Figure 3.5 and Figure 3.6. As one can be observed, across all the four datasets, the fixed tiling strategy achieves a 1.64x and 1.70x speedup on CPU and GPU, respectively, on average; and, the adaptive size tiling strategy achieves a 1.56x and a 1.65x speedup on CPU and GPU, respectively. Lastly, the scratch tiling strategy achieves 1.68x to 4.12x and an average 2.52x speedup which is a significant improvement based on the previous two tiling strategies.

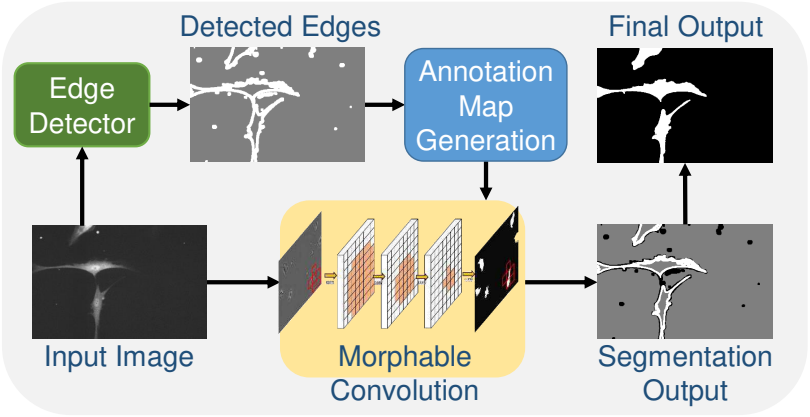


Figure 3.7: Overview of the framework

3.4 Edge Detection based Segmentation Approach with Morphable Convolutional Neural Network

Recall that we employed a tiling based framework to boost the biomedical image segmentation tasks. Here, we also propose to improve the segmentation processes by an edge detection based method. During the inference phase of segmentation, similar to other attention models, our framework first identifies the RoIs and then computes the segmentation mask for the candidate regions, which organized in a *hierarchical manner*. However, our RoIs are *not* necessarily of rectangle-shaped. As shown in Figure 3.7, the input images are first fed to an *edge detector module*. After the edges in an image have been generated, they are sent to the *annotation generator module*, which generates the layer-wise annotation maps for the segmentation network. The annotation map is a matrix, of the same height and width, as the output feature map, but its entries are either "0" or "1". In this context, 1 indicates that this pixel belongs to the RoI, whereas 0 means that pixel does not belong to the RoI. The annotation map guides the segmentation network on the problem of which pixels need to be classified, instead of indiscriminately classifying all the pixels. After all the pixels near the border have been classified, We fill up the inside area and generate the segmentation results. Figure 3.8 illustrated an example of using Morphable U-net for segmentation on an image from Fluo-C2DL-MSC dataset.

3.4.1 Annotation Map Generation

Each convolution layer in our morphable convolutional neural network framework receives an input tensor $I \in \mathcal{R}^{C \times H \times W}$ and an annotation map $A \in \{0,\ 1\}^{H' \times W'}$ as the input. The annotation map guides the framework to calculate results on specific regions of the output tensor (to make it simple, we assume that our input is 2D images for the remaining of our paper). The annotation map is a matrix consisting of 0s and

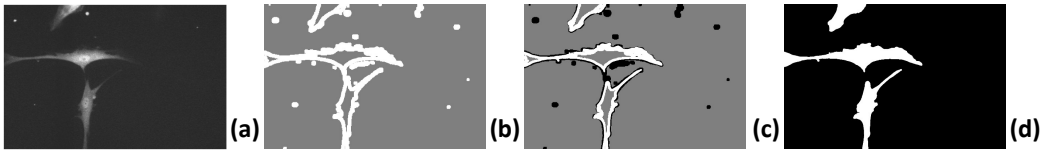


Figure 3.8: The workflow of applying the Morphable U-net on Fluo-C2DL-MSC dataset. We first take the input image (a) and detect the edges of the objects (b) on the images. Those pixels which is detected as edges will be considered as the annotation. We then use Morphable convolution neural network to only classify the pixels which annotated as 1 on the image (c). The white part in (c) are classified as "foreground", the black part are classified as "background", whereas, we did not perform any operation on the grey part since those area are omitted. Finally, we fill up the inside area of each objects to generate the segmentation mask for the image (d).

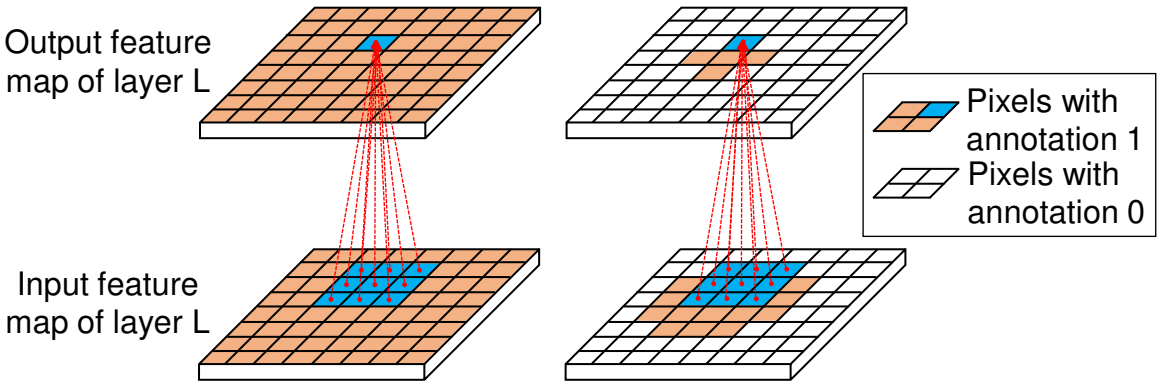


Figure 3.9: Feature map and annotation for a convolution layer. Only calculating colored pixels.

1s. As shown in Figure 3.9, the conventional convolution layer calculates the results for all the pixels on the output feature map, while our morphable convolution only computes the results for the pixels with a corresponding 1 on the annotation map. As one can expect, the annotation map for each layer is different. For example, in Figure 3.9(b), layer L is a convolution layer with a filter size of 3. In order to calculate the result of the blue pixel p on the output feature map, we need to first get the value of 9 pixels on the input feature map of layer L (colored as blue). Hence, on the annotation map of layer L-1, all the corresponding pixels of those 9 pixels should be 1.

3.4.2 Morphable Convolution Implementation

We implemented morphable convolutional neural network framework in CAFFE [104], which is a C++ based machine learning framework. We redefined the parameters of

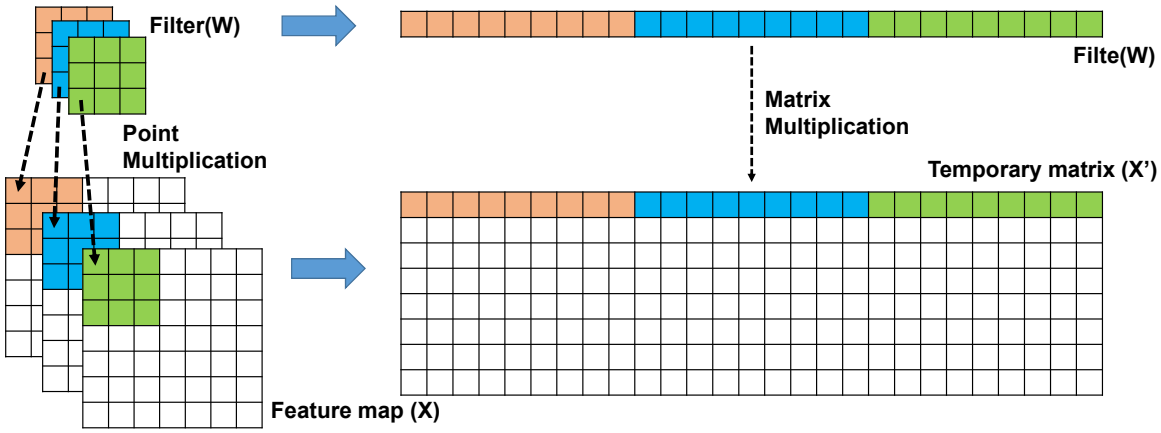


Figure 3.10: Converting the feature map into a temporary matrix.

	D-I	D-II	D-III
Canny Edge Detector segmentation	0.538	0.699	0.639
CNN Edge Detector segmentation	0.495	0.686	0.690
Original U-net	**0.933**	0.737	0.691
Morphable U-net (Canny)	0.931	0.743	0.715
Morphable U-net (CNN)	0.931	**0.747**	**0.717**

Table 3.2: IoU results.

the convolution layer in CAFFE, and also modified the underlying implementation of the convolution operation. For convolution, one of the most common approaches is *im2col*, which *converts convolution to matrix multiplication*. As shown in Figure 3.10, the input feature map is duplicated and copied to a temporary matrix where each row corresponds to a pixel on the output feature map. After performing the matrix multiplication, we obtain the output feature map. More specifically, let us denote the input feature map as $X \in \mathcal{R}^{C \times H \times W}$, the convolution filter as $W \in \mathcal{R}^{K \times K \times C \times C'}$, and the output feature map as $Y \in \mathcal{R}^{C' \times H' \times W'}$. We first convert X into a temporary matrix $X' \in \mathcal{R}^{H' \times W' \times K \times K \times C}$. For each pixel $p \in \{H' \times W'\}$ in the output feature map Y, we convert $K \times K$ values from X to X'

3.4.3 Results

To show the benefits of employing our morphable convolution framework on CNNs, we compare Canny-based and CNN-based morphable U-net against the original U-net. The former uses Canny edge detector to obtain the annotation while the later approach use CNN edge detector to generate the annotation. In both approaches, the annotation is later send to the morphable convolutional framework to guide the convolution operators. Note that we use the *same trained weights* for *all* versions of the U-net framework (Original, Canny-based and CNN-based). In this experimental analysis, we

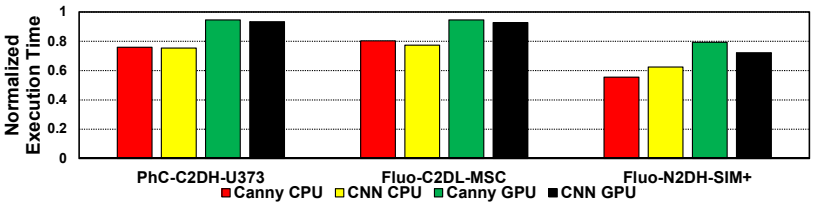


Figure 3.11: Performance of Morphable U-net against original U-net over three datasets. We evaluate two Morphable convolution U-net (Canny-based and CNN-based) on both CPU and GPU. Each experiment results have been normalized to the correspond baseline original U-net.

use normalized execution time as our latency metric, and IoU (Intersection over Union) as our accuracy metric. We show latency results in Figure 3.11 and accuracy results in Table 3.2. For all three datasets, two morphable U-nets achieve same level of accuracy with the original U-net. All CNN based segmentation approaches are outperform than the edge detection based softwares in terms of the accuracy. It is important to point out that the accuracy result for morphable convolution might has small difference from the original U-net, since some pixels on the border area have different predictions result by edge detector and U-net. Another observation is that the morphable convolution based U-net achieves higher speedups with all the three datasets on both the CPU and GPU implementations. It can also be observed that the CPU implementation achieves more relative speedup, since the serial annotation generation process (running on CPU) contributes more to the total execution time of GPU implementation. Here, the latency results of both morphable CNN implementations already include the edge detection and annotation generation time.

For all results but Fluo-N2DH-SIM+, CNN-based U-net has higher speedup than Canny-based U-net as CNN does better job in edge detection. Canny-based U-net achieves better CPU result for Fluo-N2DH-SIM+ dataset, since the cells in that dataset are simple, such that Canny generates less false edges. Also, MATLAB does not support GPU version of edge detection, which causes Canny-based U-net result in higher execution latency on the GPU platform.

3.5 Discussion

In this section, we want to discuss an potential extension for our approach. In general, both tiling-based segmentation and edge detection based segmentation are hierarchical frameworks, where a former network decides if a later network will be invoked. This mechanism can be extend to a multiple networks structure, where an selection model decides which segmentation network to be invoked. Additionally, we can integrate these hierarchical frameworks as an end-to-end framework. Such end-to-end model

can be trained automatically and the accuracy of the selection model will potentially be higher than a separate hierarchical framework.

Chapter 4

Improving Conventional Protein-Ligand Docking Algorithm with Machine Learning Techniques

In this chapter, we focus on the protein-ligand docking problems for the drug-discovery, and propose to employ the CNN and GNN model to improve the docking algorithm over the existing docking software (e.g., MedusaDock [30–33, 60]). We design a CNN model which can predict if a protein-ligand complex pose is a stable docking. We further use such CNN model to boost the MedusaDock by reducing the number of attempts to run MedusaDock. Our approach can also increase the docking accuracy by helping more protein-ligand pairs find at least one good docking pose. We also designed a GNN model which can generate the good docking pose directly without the sampling process. The user takes the initial docking pose as the input and use our GNN model to improve the docking pose. Both machine learning methods (CNN and GNN) improved the docking accuracy and reduce the docking time.

4.1 Introduction

Recent efforts have achieved significantly success in biochemistry. One of the most important area in biochemistry is drug discovery where people design new drug molecules to make specific interaction with the protein molecules. The broad area of computational drug discovery has seen significant advancements during the last decade, achieving impressive results for identifying drugs targeting different diseases [25–29]. The traditional drug discovery process typically takes multiple years and costs billions of dollars [105–108]. To accelerate the drug discovery process, various computational

docking approaches [30, 31, 34, 36, 37, 62] have been proposed, in the past, for virtual drug screening. Most existing computational docking methods are based on physical force fields and customized potential functions. Unfortunately, docking software usually suffers from *both* low accuracy and long latency.

In general, existing computational docking protocols include two main steps – *sampling* and *scoring*. The first step involves sampling as many ligands poses as possible in a reasonable (or user-specified) number of steps. Since searching the whole conformational space is impractical, mainly due to a large number of flexible rotamers of the ligand and protein side chains, many docking software adopt heuristic strategies to reduce the searching space. For example, MedusaDock [30–33], MCDOCK [37], and ProDock [38] employ Monte Carlo (MC) [109] algorithms. In comparison, AutoDock [34, 35, 68, 110] and GOLD [36] adopt genetic algorithms (GA) [61], which start first with an initial pose and then progressively identify more precise poses. In contrast, FlexX [62] employs a fragment-based method [63], which starts the search with fragments with low binding affinities.

In the scoring step, the free energy of the protein-ligand complex is calculated to evaluate the binding affinity. The accuracy of the scoring function dictates the accuracy of the docking. There are two major types of scoring functions: *physics-based* [65–71], *knowledge-based* [72–76], or hybrid [111–114]. Physics-based scoring functions compute the interactions among atoms based on force fields. For example, MedusaDock [30–32, 60] employs MedusaScore [33] as the scoring function, which takes into consideration of covalent bonding, Van der Waals (VDW [115]) interaction, hydrogen-bonding, and solvation effects. On the other hand, knowledge-based scoring functions calculate the energy based on known knowledge extracted from experimentally-solved protein-ligand 3D structures. The primary limitation of the knowledge-based scoring functions is the difficulty of evaluating the binding affinity of unknown protein-ligand complexes. As a result, although knowledge-based scoring functions improved the docking accuracy for known protein-ligand complexes, they suffer from limited accuracy for complexes that have not been solved before.

Recently, various machine learning-based scoring functions have been proposed. For example, Wang et al. [116] employ a random-forest scoring function to evaluate the goodness of a protein-ligand docking pose. AGL-Score [117] learns the protein-ligand binding affinity based on the algebraic graph theory. Cruz et al. [118] extract the learning features based on the protein-ligand atoms-pair interconnection, and predict the binding affinity with random-forest (RF) and gradient boosting trees (GBT). AtomNet [39] maps the atoms near a binding site into a 20 Å grid and uses a 3D-CNN model to predict the binding probability for each protein-ligand complex. Similarly, Ragoza et al. [40] also build a 3D-CNN model to predict the protein-ligand binding affinity. Kdeep [41] employs a 3D-CNN model to predict the binding affinity of a protein-ligand complex based on its 3D structure. TopologyNet [42] employs a

hierarchical structure of convolutional neural network to estimate the protein-ligand binding affinity upon mutation. Graph neural network (GNN) based approaches have also been proposed to target the protein-ligand scoring problem. For instance, Lim et al. [43] employ two GNN models with attention mechanisms to calculate docking energies. The first model takes the structure of the protein-ligand complex as the input graph, whereas the second one takes, as input, the protein and ligand structures, separately. The binding affinity is inferred by the subtraction between the two models. In comparison, Morrone et al. [119] investigate the ranking of a docking pose from the docking software as a feature to predict the score of that pose. Torng et al. [120] compute the total energy of both protein and ligand with different GNNs. TorchMD-NET [44] study an equivariant transformer to predict the properties of the molecules. The output features of the two networks are then concatenated together to predict the docking energy. Although these machine learning-based approaches improve the scoring, they rely heavily on the pose sampling strategies adopted in conventional docking software. More specifically, to obtain the final docking pose of a protein-ligand complex, all previous machine learning-based approaches evaluate all the ligand poses sampled by conventional docking software and then select the near-native ones[1]. As a result, it may still take hundreds of runs of conventional docking software to obtain a good ligand pose.

Motivated by these observations, in this thesis, we propose two approaches to improve overall process of the protein-ligand docking. More specifically, we proposed a 3D-CNN based framework "MedusaNet" to reduce the number of iterations during the sampling step and save the overall docking time. We also proposed a GNN based framework called "MedusaGraph" which can generate a docking pose directly from an initial docking pose without the sampling process. Both mechanism improved the docking accuracy while reduce the docking time.

4.2 MedusaDock

We take MedusaDock as the existing protein-ligand docking application as the baseline and explaine how to use machine learning method to improve it. MedusaDock starts with a stochastic rotamer library of ligands generation, then shifts and rotates the rotamers into different poses. Based on the Monte Carlo algorithm and score function, MedusaDock tries to search for the best-fit poses within the docking boundary box. During this search process, both protein side-chain and ligand will be repacked to imitate the realistic protein ligand docking procedure. Additionally, to optimize the searching algorithm and reduce the computation of score function, MedusaDock separates the search step into *coarse docking* and *fine docking*. In coarse docking,

[1]The pose, which is similar to the crystallographic structure, and is also referred as 'good pose'.

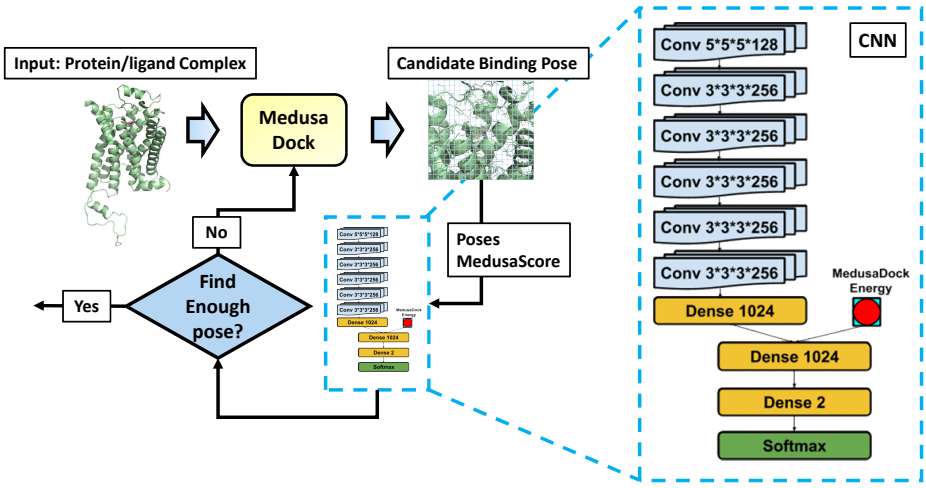


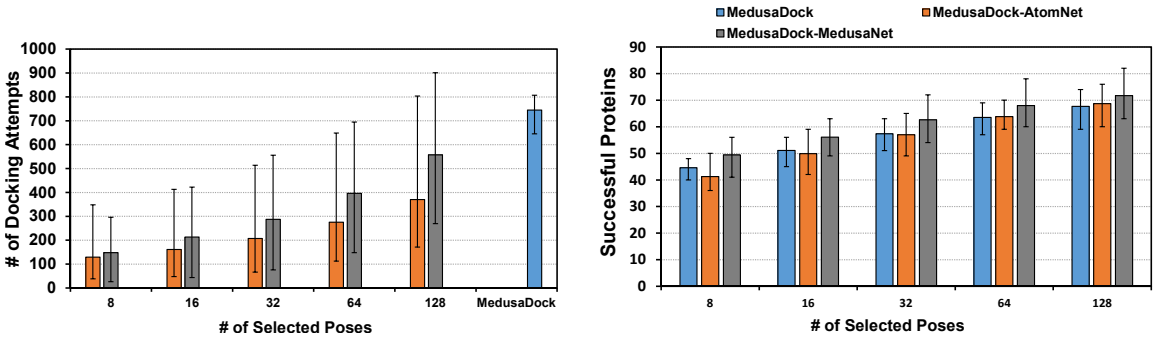
Figure 4.1: This figure shows the structure of our MedusaNet model (right) and how it can be used to improve MedusaDock (left). The structure of MedusaNet model includes 6 convolution layers and followed by 3 fully-connected layers. The energy score of MedusaDock is added as a feature for the second fully-connected layer. In the combined framework, MedusaDock takes a protein-ligand complex and generates some candidate poses with an energy score. The MedusaNet evaluates each of the poses generated by MedusaDock. The framework stops if MedusaNet determines that MedusaDock has generated a sufficient number of good poses – otherwise, MedusaDock is executed again with a new random seed.

MedusaDock quickly identifies several good poses, and those poses are carefully checked during the fine docking step. To search the docking poses of a protein-ligand complex with MedusaDock, we have to keep running MedusaDock with different random seeds until we cannot find a pose with lower energy. For some proteins, we need to run MedusaDock for thousands of times, even if some good poses have already been found.

4.3 MedusaNet

4.3.1 Using CNN to Improve MedusaDock

In this section, we propose a 3D CNN model to predict binding probability. The structure of the CNN model is shown in Figure 4.1. Unlike previous CNN based work, we include the MedusaScore as the input feature of the CNN. The input of this 3D CNN model is the 3D image of the protein-ligand structure and the output is a probability score which indicate how likely this docking structure is a good docking pose.



(a) MeudsaDock run less attempts with CNN guiding.

(b) Number of proteins found good pose by each approach.

Figure 4.2: Comparison between the performance of the original MedusaDock and MedusaDock guided by different CNN models. **(a)** The number of docking attempts performed by different MedusaDock versions. **(b)** The number of protein-ligand complexes that have good ligand poses generated by different versions of MedusaDock. Good poses have RMSD < 2 Å and bad poses have RMSD > 2 Å. The test set is composed of 100 randomly selected protein-ligand complexes from the PDBbind test set. Note that these 100 proteins are not used for training. We applied cross-validation for 10 times. The bars indicate the average results and the error bars show the minimum/maximum results across 10 validations.

4.3.2 Improve MedusaDock with MedusaNet

To optimize the docking process for MedusaDock, we further propose a strategy to use MedusaNet to improve the MedusaDock. Recall that the MedusaDock only generate the binding energy of a given pose. To propose the best docking pose, one requires to run MeudsaDock for multiple attempts until they cannot find a binding pose with lower energy. More specifically, each time select a random seed for the Monte Carlo algorithm and use MedusaDock to propose the several poses. We record the pose with the lowest energy. This processing will be terminated when the global lowest energy is converge. Finally, we will select a specific number of poses based on the docking energy.

In our new strategy (As shown in Figure 4.1), we will use MedusaDock to evaluate the poses generated in each attempt. The good poses will be collected. The entire process will be terminated until a specific number of good poses have been collected. This new strategy can help to reduce the total number of attempts of MedusaDock running. Also, it can help more proteins find at least one good binding pose.

4.3.3 Results

We employ two CNN models (AtomNet and our MedusaNet) to guide the MedusaDock and test both models by combining them with MedusaDock through the

aforementioned strategy in a dataset composed of 100 randomly selected protein-ligand complexes from PDBbind testing set and compare both CNN-augmented versions of MedusaDock (MedusaDock-AtomNet, MedusaDock-MedusaNet) with the original version of MedusaDock. Recall that we randomly choose the training and testing set for 10 times in the cross-validation, the 100 randomly selected complexes are not used for model training in each validation. We observe that the original MedusaDock performs 745 docking attempts averagely among 10 cross-validation sets to achieve a convergent minimum energy (Figure 4.2). On the other hand, MedusaDock-AtomNet, MedusaDock-MedusaNet only need to perform 370 and 557 docking attempts, respectively, when 128 good poses are selected. If we only select 8 good poses, the number of needed docking attempts can slump to 129 and 148, respectively, as compared to original 745 attempts. Out of the 100 protein-ligand complexes, MedusaDock, MedusaDock-AtomNet, and MedusaDock-MedusaNet can generate near-native ligand pose (RMSD < 2 Å) for 44.6, 41.3, and 49.4 complexes on average, respectively, when 8 poses are selected. If we select 128 poses for each complex, MedusaDock, MedusaDock-AtomNet, and MedusaDock-MedusaNet can propose at least one good pose for 67.7, 68.7, and 71.7 complexes on average, respectively. Further, for a large number of selected poses (e.g., 128), CNN-guided MedusaDock might need more attempts compared to the original MedusaDock. For example, on one validation set, MedusaDock-MedusaNet takes 1.27× attempts compared to original MedusaDock to select 128 good poses for each complex. This result indicates that, for some proteins, performing docking attempts until the energy converges is not a sufficient condition for termination. From Figure 4.2(a), one can also observe that the error bars of the number of docking attempts is very large, which indicates that some proteins will take many attempts of MedusaDock running while other proteins only require a few attempts. This is because the MedusaDock employs a random searching algorithm (Monte Carlo). For some proteins, MedusaDock find good poses at early attempts while for other proteins, the good poses are unfortunately proposed in the later attempts. Also, for some complex proteins, MedusaDock can only find a few good poses in total. As the results, we have to run much more attempts for the "Unlucky" proteins than those "lucky" proteins

4.4 MedusaGraph

In this section, we propose a GNN based framework MedusaGraph to generate the final docking pose of a protein-ligand pair without the long-time sampling process. MedusaGraph includes two GNNs. The first network predicts the best docking pose for a protein-ligand pair from an initial docking pose, and the second one evaluates the output pose from the first network and predicts if the pose is near-native. Our extensive evaluations reveal that we can efficiently predict "good docking poses" *without*

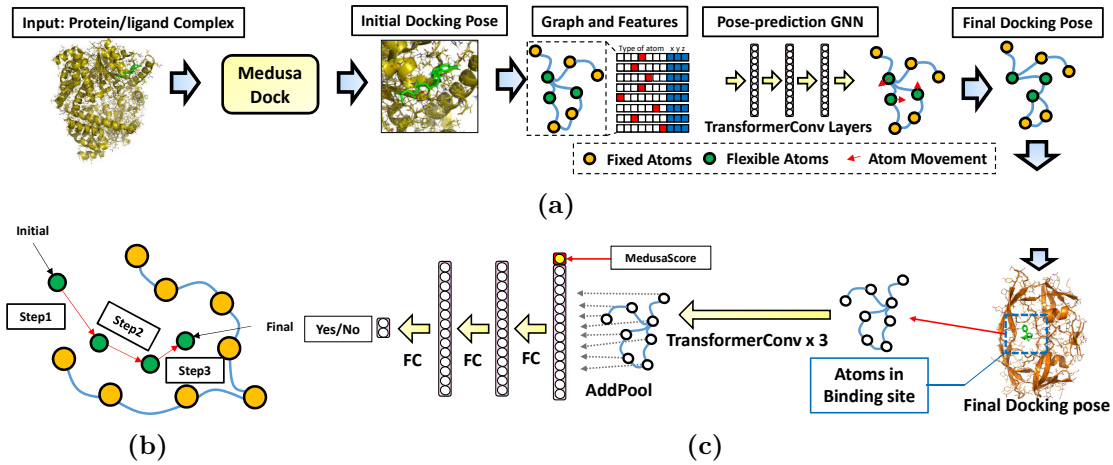


Figure 4.3: (a). The flow of the pose-prediction GNN model. We first run MedusaDock on a protein-ligand complex to achieve a candidate docking pose. Then, the pose-prediction model calculates the movement of each flexible atom in the complex. Flexible atoms include all the ligand atoms and the receptor atoms that are near the receptor surface. Finally we obtain the final docking pose based on the movement prediction. The pose-prediction GNN model contains several TransformerConv layers. (b). Multi-step pose-prediction. An atom moves from the initial location to the final location step-by-step. (c). The flow of the pose-selection GNN model. The final poses generated by the pose-prediction model travel through three TransformerConv layers and three fully-connected layers. The final output is the Yes/No neuron indicating the docking probability.

sampling thousands of possible docking poses. The method section is divided into two parts where each part corresponds to a GNN. In each part, we first explain how to prepare the data for the model training/testing, and then introduce the detail of the model. After that, we give an example to illustrate how we use MedusaGraph to obtain the docking pose for a protein-ligand complex.

4.4.1 Pose Prediction

4.4.1.1 Data Preparation

For the pose-prediction GNN model, we compiled a dataset based on the PDBbind 2017 refined set [121] using the strategy outlined in Wang et al. [31]. The protein-ligand complexes with less than two rotatable bonds as well as the proteins that have more than one ligand have been removed. We have also removed the proteins with missing residue or duplicated residues. The final dataset contains 3,738 protein-ligand complexes. To train the GNN model, we have randomly selected 80% of the proteins as *training data* and the remaining 20% have been reserved as *testing data*. To make sure the training set and the testing set do not share similar proteins, we use CD-HIT [122, 123] to cluster the proteins with CD-HIT's default setting (sequence

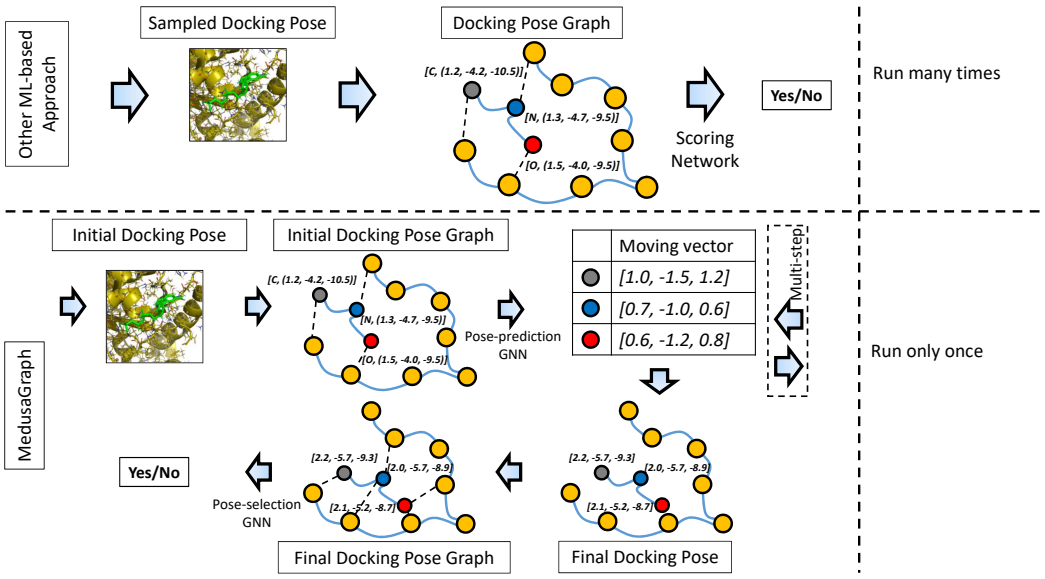


Figure 4.4: An example of generating the docking pose with MedusaGraph and other ML-based approach.

identity cut-off as 0.9) before we do the random splitting. In our method, we use MedusaDock to generate an initial docking pose for each protein-ligand complex. More specifically, We run MedusaDock with a given random seed to perform one iteration of the sampling process to generate several candidate docking poses as the initial docking pose. Note that the binding site information is already given in the PDBbind dataset. The initial pose is then translated into a *graph representation*, where each vertex represents an atom in the complex and the edges in the graph indicate the connection between the nodes (e.g., the covalent bond or the interactions between nearby atoms). The input feature of the pose-prediction model is a N by 21 tensor, where N indicates the number of atoms in the complex. The feature for each vertex has a length of 21. The first 18 elements represent a categorical feature that indicates the type of the atom. The last 3 elements include the 3D coordinate of the atoms (x, y, z) in the initial pose. To construct the graph from the protein-ligand complex, we add an edge between the atoms if there is a covalent bond between these two atoms. We also add an edge between a protein atom and a ligand atom if their distance is less than 6Å since the nearby atoms will have a higher chance to interact with each other. We select this 6Å threshold inspired by some previous work. [32, 40, 43, 124] The edge features in the graph include the distance between the vertices and the type of the connections (protein-ligand, protein-protein, or ligand-ligand). we list the node features and the edge features in Table 5.3.

	Index	Features
Node Features	1-18	Atom type (N, C, O, S, Br, Cl, P, F, I)
	19-21	3D Coordinates in X, Y, Z
Edge Features	1	Ligand-ligand Distance
	2	Protein-ligand Distance
	3	Protein-protein Distance

Table 4.1: Summary of the node features and edge features. For the atom type feature, we consider the same type of atoms in protein and ligand as different atom types. We use one-hot encoding for atom types where only the corresponding index has the value of one while other indices contain zero value.

4.4.1.2 Graph Neural Network Model

As shown in Figure 4.3a, our first GNN, called the *Pose Prediction GNN*, takes the initial pose of a protein-ligand pair as the input. During the docking, the position of the ligand atoms will move while the protein atoms are more likely to be immobile. Based on this observation, we divide the nodes in the graph into two parts – *fixed nodes* and *flexible nodes*. The pose-prediction graph neural network is a vertex regression model which calculates the movement for the flexible nodes and outputs the moving vector (x, y, z), which indicates the movement along each axis. We use the TransformerConv layer [125] to implement this network. The transformer convolution layer employs the attention mechanism which captures the importance between each pair of atoms. Also, it includes the edge features (e.g., edge type, distance) as inputs feature. As a result, the TransformerConv can more precisely compute the interactions between the atoms in a protein-ligand complex. The TransformerConv layer calculates the output feature of each node using the following equation:

$$x_i^{'} = W_1 x_i + \sum_{j \in N(i)} \alpha_{ij}(W_2 x_j + W_3 e_{ij}),$$

where x_i is the input feature vector of the node i, $x_i^{'}$ is the output feature vector for node i, and $N(i)$ is the set of the neighboring nodes for node i. The attention matrix α_{ij} is calculated using:

$$\alpha_{ij} = \text{softmax}\left(\frac{(W_4 x_i)^\top (W_5 x_j + W_3 e_{ij})}{\sqrt{d}}\right),$$

where x_i indicates the input feature of node i, e_{ij} is the edge feature of edge $\langle i, j \rangle$, d is the hidden size of the node feature, and all Ws are "learnable" weight matrices. The attention matrix α_{ij} helps the network distinguish between the importance of different neighbours for each and every atom. The output feature vector of the first three TransformerConv layers has 256 hidden neurons while the last TransformerConv layer has an output size of three (which indicate the movement in x, y, z axis). We

then add up the coordinates of the flexible nodes in the initial pose with the moving vector of those nodes to obtain the coordinate of the flexible nodes in the final docking pose. Unlike the equivariant networks [44], we have fixed the location of the protein atoms and the coordinate of the ligand atoms are related to the protein atoms. As a result, rotating the entire protein-ligand complex does not affect the final complex structure prediction. Similar to some human body pose prediction work [126, 127], we use L1-loss [128] as the loss function as below:

$$L = \sum |x_c^i - x_1^i - x^i| + |y_c^i - y_1^i - y^i| + |z_c^i - z_1^i - z^i|$$

, where (x_1^i, y_1^i, z_1^i) is the initial coordinate of the i-th atom, (x_c^i, y_c^i, z_c^i) is the coordinate of the i-th atom in the x-ray crystal structure, and (x^i, y^i, z^i) is the moving vector predicted by MedusaGraph for the i-th atom. During the training, only the flexible nodes contribute to the loss function, since we only want to predict the movement of the flexible nodes.

4.4.1.3 Multi-step Pose Prediction

In our exploration, we observe that our pose-prediction GNN model cannot accurately estimate the movement of some atoms (from an initial location to ground-truth location). This is because our estimation is highly based on the interactions between the atoms. Ideally, for each atom, the model will calculate the force from other atoms and calculate which direction this atom will move, and how long it will move. However, for some atoms which are far away from the ground-truth pose, we need additional iterations to simulate the movement from the initial location to the final location. As a result, we propose a multi-step pose prediction mechanism to calculate the final location of each atoms step-by-step (as shown in Figure 4.3b). More specifically, we divide the path from the initial location to the final location into several steps, and we train multiple models to predict the atom movement in each step. The output of the i-th model will be the input of the (i+1)-th model. The output of the last model (for all atoms) will be considered as the final predicted pose. It is important to note that we also change the connection between the protein atoms and the ligand atoms after each iteration of the atom movement prediction. This is because, the location of the atoms are changed, and some nearby atoms could become far away from each other while some other atoms could come closer.

4.4.2 Pose Selection

4.4.2.1 Data Preparation

We also compile a pose-selection dataset from the PDBbind pose prediction dataset [2], which is generated as described in the previous section. After we obtain the initial docking pose (the graph structure) for each complex, we apply our pose-prediction GNN to the initial docking poses and obtain the final docking poses for each complex. We divide all the final poses into two groups: (i) good poses and (ii) bad poses. Good poses have RMSD values that are less than or equal to 2.5 Å with respect to the crystallographic structure, and bad poses have RMSD values that are greater than 2.5 Å. We choose the split threshold as 2.5 Å based on the choice of the previous works. Usually, this threshold is selected between 2 Å to 4 Å [2, 40, 43]. This dataset will be used to train and evaluate our pose-selection GNN model. The pose-selection GNN model is expected to identify if a pose is a good one or not. The training/testing set split remains the same as the dataset for pose-prediction. The poses for the same protein will be included in either the training set or the testing set.

4.4.2.2 Graph Neural Network model

After the final docking pose is generated, our second GNN, referred to as the *Pose Selection GNN*, will predict if such a pose is a good pose or not. This network is basically a graph binary classification model. The input feature format is the same as the first network, which means we translate a pose into the graph representation. As shown in Figure 4.3c, our model includes 3 **TransformerConv layers** to calculate the features of each and every node based on its neighbours. After that, the features of the flexible nodes are added together with an add-pooling layer. The MedusaDock energy (MedusaScore) of that pose is concatenated into the feature vector after the pooling layer. This MedusaScore can be obtained when we generate the initial docking pose with MedusaDock. Finally, there are 3 fully-connected layers at the end of the network to predict the probability of each pose, i.e., if it is a good pose or not. The output of the network is a two-neuron tensor, which indicates (in a probabilistic fashion) whether the pose is a good one or a bad one.

4.4.3 An Example Docking Process with MedusaGraph

To better explain the flow of our proposed framework, in this section, we go over an example use of MedusaGraph to obtain a docking pose for the protein-ligand complex depicted in Figure 4.4 and compare it against the prior ML-based docking approach. In the prior ML-based approaches, a user first generates the sampled docking poses

for the protein-ligand complex using a conventional docking software. After that, the sampled poses are converted into the docking pose graphs. These graphs are then input to the scoring networks to predict the docking probability of each corresponding docking pose. Based on the prediction results, the user can select the good poses from among the sampled poses. Note that this process is typically repeated many times since a single running of the conventional docking software might not be able to find any good pose.

On the other hand, in our proposed MedusaGraph, after MedusaDock generates the initial pose for the protein-ligand complex, the first step is to convert the initial pose into the graph-structure (complex graph). In this example, three ligand atoms (the blue, red, and grey nodes) are considered *flexible nodes* whereas the remaining atoms (the orange nodes) are *fixed nodes*, which represent the protein atoms.[2] We add edges between the atom pairs with covalent bonds. We also add an edge between a protein atom and a ligand atom if they are close, since nearby atoms have a higher chance of affecting each other. Each node contains the type of the atom and the 3D-coordinate as "node feature", and each edge contains the type of the connection and the distance as "edge feature". This graph structure annotated with the mentioned features is then fed into the pose-prediction GNN model to predict the movement of each atom. The output of the model is a 3 by 3 tensor, indicating the (x, y, z) of the moving vector for the 3 flexible nodes. We can compute the final position of each flexible atom by adding the initial coordinate and the moving vector together. For example, for the nitrogen atom (the blue node) with an initial 3D-coordinate of (1.2, -4.2, -10.5), the GNN model predicts its moving vector as (1.0, -1.5, 1.2), and the final position is calculated as (2.2, -5.7, -9.3). To apply our multi-step prediction, we reconstruct the complex graph with the calculated coordinate as the initial coordinate, and iteratively feed it into the pose-prediction model. After we obtain the final docking pose, we construct a final docking pose graph (similar to the docking pose graph in other ML-based approaches) and send the graph representing the final docking pose to the pose-selection GNN model. This GNN model in turn predicts the probability that this final docking pose is a good pose. We want to emphasize that, compared with other ML-based approaches, MedusaGraph does not have to be invoked multiple times to ensure at least one good pose is sampled. This is because, even if an initial pose generated by MedusaDock is a bad pose, the pose-prediction GNN model will very likely convert it into a good pose.

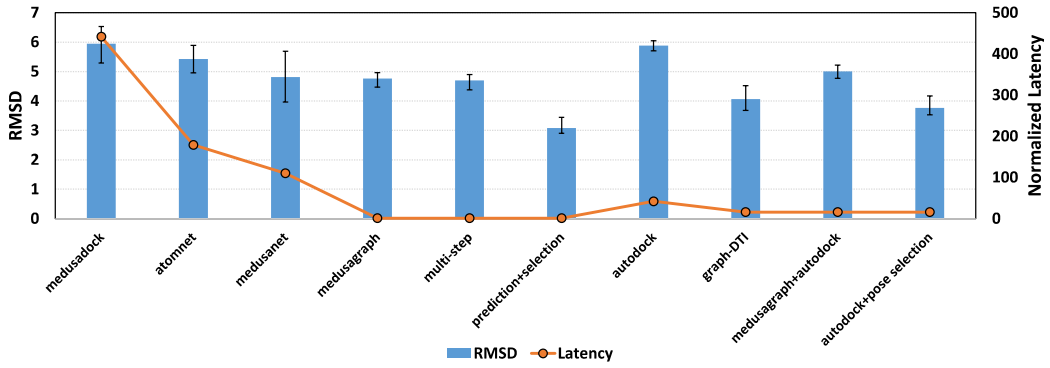


Figure 4.5: Average RMSD of the output poses for each approach. For the normalized latency, we set the execution time of one running of MedusaDock as 1, and normalized all other approaches. We randomly split the dataset into a training set and a testing set ten times to perform the cross-validation and the error bar indicates the min/max RMSD value of each cross.

4.4.4 Results

4.4.4.1 Comparison of MedusaGraph against Existing Pose Prediction Schemes

We first evaluate the quality of the poses generated by our pose-prediction GNN model trained on the PDBbind pose-prediction dataset. In this context, we measure the "goodness" of the pose by RMSD with respect to the X-ray crystal pose (ground truth pose). Additionally, since our model can predict the final docking pose directly, we expect our approach to be much faster than other existing approaches. We compare our approach against two state-of-the-art docking frameworks (MedusaDock [30, 33] and Autodock Vina [110]). To find the best docking pose for each protein-ligand complex, MedusaDock and Autodock Vina need to sample different candidate poses and calculate the energy score of each candidate. We run MedusaDock and Autodock Vina with different random seeds until we cannot find a pose with a lower energy score. We also compare MedusaGraph against two Convolutional Neural Network (CNN) based approaches (AtomNet [39] and MedusaNet [2]) and a Graph Neural Network (GNN) based approach (Graph-DTI [43]). Since these neural network models can only do scoring, we utilize conventional docking approaches (MedusaDock and Autodock Vina) to generate the poses for them. For the CNN-based approaches (AtomNet and MedusaNet), we generate poses with MedusaDock. Specifically, we run MedusaDock with different random seeds and stop when the CNNs determine they have found 8 good poses. Finally, for Graph-DTI, we run AutoDock Vina [110] with a random seed of 0 and set the exhaustiveness parameter to 50 to ensure that AutoDock searches the conformational space thoroughly.

[2]Recall from pose-prediction section that a flexible node refers to the ligand atoms while a fixed node refers to the protein atoms.

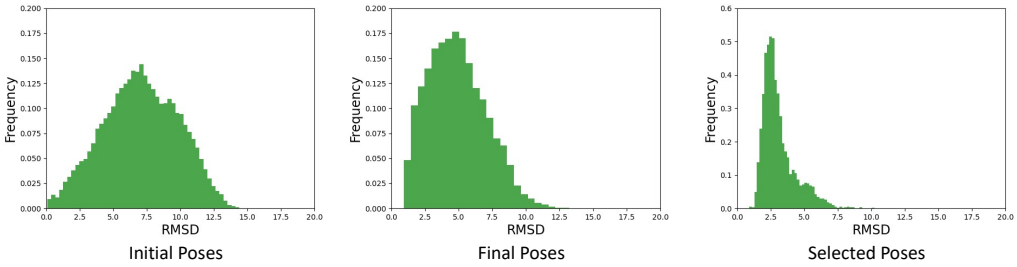


Figure 4.6: Histogram of the RMSD of all poses for initial docking poses generated by MedusaDock, the final docking poses generated by the 3-step pose-prediction model, and the poses selected by the pose-selection model.

We report, in Figure 4.5, the *average RMSD of output poses* for each of the approaches tested. We also report the normalized latency of each method. Note that the execution time of the neural network is usually negligible compared to conventional docking software (MedusaDock and AutoDock). As a result, we consider the latency of one run of MedusaDock as 1 unit and normalize the latencies of all other approaches. One can observe from these results that (i) MedusaGraph achieves significant speedups compared to the other approaches, and (ii) MedusaGraph's output poses have similar average RMSD values to other approaches (less than 0.7Å). Further, if we apply our pose-selection model to the output poses, the average RMSD of the output poses drops to 2.9, which is much better than those obtained by using the other methods. This indicates that it is easy to distinguish the near-native pose from the poses generated by the pose-prediction GNN model. (iii) MedusaGraph can also be applied to other docking software (e.g., Autodock) to improve their generated poses.

4.4.4.2 Statistical Analysis of Predicted Poses

We observe that the average RMSD of the poses generated from the pose-prediction GNN model is around 5Å. This is far away from the crystal structure. However, this result is still good enough considering that these values are the average RMSD of all the poses. On the other hand, one can also be interested in the "percentage of good poses" among all generated poses. In other words, *have our pose-prediction models improved the percentage of good poses compared to the original MedusaDock output poses?* To answer this question, we have also evaluated the distribution of the poses we generated. We plot the histogram figure for the RMSD of all selected poses in Figure 4.6. It can be observed that the RMSD distribution of the poses generated by the pose-prediction model has a significant shift to the left compared to the original poses, indicating that our pose-prediction model can *improve* the poses generated by MedusaDock. After we have applied the pose-selection models to the output poses, most poses have a small RMSD as shown in the rightmost figure.

We have also calculated the percentage of the poses with an RMSD value less than 2.5Å

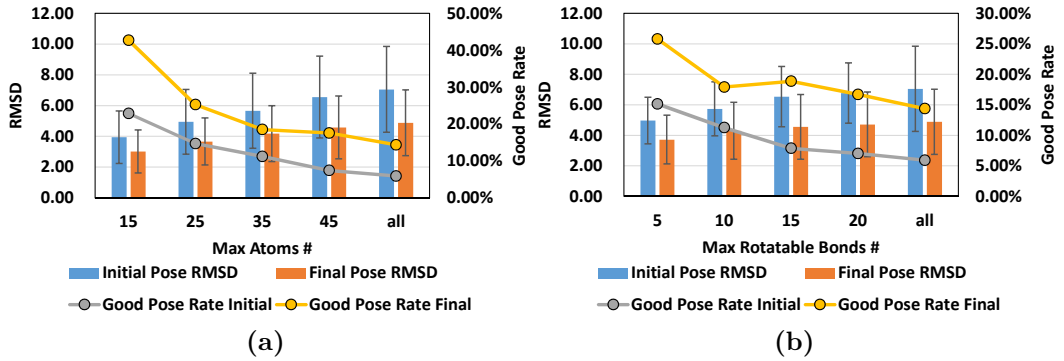


Figure 4.7: We show the average RMSD of initial poses and final docking poses for the complexes with different properties. We also calculate the good pose (with RMSD less than 2.5Å) rate in initial docking poses and final docking poses. (a) results of the complexes with a different number of atoms, and (b) results of the complexes with a different number of rotatable bonds.

(which we consider a "good pose") among all the poses generated by each approach. 5.9% of the original poses generated by MedusaDock have their RMSD less than 2.5Å. After we have applied our pose-prediction model, 14.4% of the poses are less than 2.5Å. Additionally, 37.6% of poses are near-native if we use the pose-selection model. These results clearly show that our pose-prediction model can help the protein-ligand complexes find more near-native poses. Further, our pose-selection model can help to choose good poses among all candidate poses.

4.4.4.3 Study of Ligands with Different Properties

As also discussed in prior studies [2, 32, 124], some protein-ligand complexes are easier to find good poses than others. This is mainly because the flexibility of each complex can be different from the others. In general, if a ligand has more atoms, or the ligand has more rotatable bonds, the resulting complex makes it more difficult to find good poses. In this part of the evaluation, we first classify all the complexes we evaluated in our approaches based on the number of atoms and the number of rotatable bonds. We then show the accuracy results of our approach with each class of the complexes in Figure 4.7. It can be seen from this plot that the complexes with ligands that have fewer atoms and rotatable bonds result in low-RMSD poses. However, it should be emphasized that, for all classes of complexes tested, our approach can improve the final docking poses in terms of the RMSD value.

4.4.4.4 Evaluation of Pose-Selection Models

Our pose-selection GNN model can select the good poses from among all generated poses, to potentially improve the final pose. In this part of our evaluation, we performed

Evaluation Metric	Accuracy			AUC		
	avg	min	max	avg	min	max
Medusadock	N/A	N/A	N/A	0.474	0.462	0.489
Atomnet	0.741	0.628	0.872	0.863	0.849	0.885
Medusanet	0.855	0.705	0.93	0.893	0.868	0.915
Autodock Vina	N/A	N/A	N/A	0.615	0.592	0.636
Graph-DTI	0.895	0.836	0.953	0.906	0.876	0.933
Pose selection	0.914	0.855	0.954	0.892	0.866	0.923
Pose prediction+selection	0.958	0.940	0.981	0.960	0.943	0.985

Table 4.2: Evaluation of MedusaGraph against other approaches in terms of classification Accuracy and AUC. We evaluate these approaches on the PDBbind test set. Note that the accuracy for MedusaDock and Autodock Vina is marked as N/A because the scoring function of MedusaDock and the affinity score of Autodock cannot be used to distinguish between a good pose and a bad pose. It can only be used to compare the goodness of two poses.

two computational experiments to test the robustness of the pose-selection GNN model. We first train the pose-selection model (and three state-of-the-art machine learning based models) on the original poses generated by MedusaDock. Secondly, we train the pose-selection model on the poses generated by our pose-prediction GNN model. We show the accuracy and AUC in Table 4.2. Here, *Accuracy* is defined as the fraction of the total poses an evaluated approach correctly predicts. On the other hand, *AUC* is the area under the ROC curve, which is widely used to determine how well a binary classification model can distinguish between two groups of data. It is defined as the true-positive rate against the false-positive rate. Hence, an AUC of 1 indicates a perfect model, while an AUC of 0 means an entirely wrong model. A model with an AUC of 0.5 is considered to be close to a random selection. We can observe from these results that our pose-selection GNN model performs better on the poses generated by the pose-prediction model than the poses generated by MedusaDock, meaning that it is easier to select good poses from the poses generated by our pose-prediction GNN model than selecting from the initial set of poses.

4.4.4.5 Evaluation on External Dataset: CASF

To evaluate the effectiveness of MedusaGraph on external datasets, we train our graph neural network model with the aforementioned Pdbbind dataset and test it with the CASF dataset [129–131]. CASF contains 285 proteins with their corresponding ligands, and it also provides the optimal docking structure for each protein-ligand pair. When training the GNN model with Pdbbind dataset, we remove the proteins that are also included in the CASF dataset to avoid data bias. From Table 4.3, we can observe that the proposed poses predicted by MeusaGraph are better than other approaches, indicating that MeusaGraph can work on different docking power benchmarks that

	MedusaDock	Autodock	Pose prediction	Pose prediction+selection
RMSD	5.491	5.617	4.331	2.812

Table 4.3: Average RMSD of all poses generated by each approach.

are widely used in the drug discovery community.

Chapter 5

Accelerating Image Classification CNN Models with Approximation computing based on Performance Modeling

In this chapter, we analyze the CNN based image classification pipeline and observe that there is significant scope to reduce computations by approximating each convolution layer in the CNN model while keep the acceptable accuracy. The proposed *Fluid* design takes advantage of eager execution by allowing the consumer task starts its execution before the producer complete, and can benefit any types of applications which involve data-dependency. We proposed program language define, compiler support and the runtime system implementation to enable the Fluid framework. Additionally, to help the user set the approximation aggressiveness of each convolution layer, we design and proposed a performance modeling mechanism which include both latency model and accuracy model to automatically tune the optimal approximation parameters to achieve the trade off between the accuracy and execution time based on the user requirement. With our proposal, a user can easily get the latency reduction for the CNN based image classification application without the human labor and domain knowledge.

5.1 Introduction

Many recent works have examined models of approximate computation as a means of improving performance [14] and/or energy efficiency [14, 77] for error-tolerant and

self-correcting applications, by coping with unreliable components [15], or executing inherently stochastic algorithms [16]. Common approaches among these models include eliding the execution of certain tasks, or replacing an approximable task with an entirely different computation that is easier to execute [132]. A different flavor of approximation has been studied in the form of data-race tolerant and other scheduling-robust algorithms under weakened synchronization assumptions [78]). For an approximate computation approach to be viable, the intermediate approximated values consumed by the following tasks must be close enough to the values generated by precise computation. Note, however, that once approximation is allowed and soundness constraints have therefore been relaxed, there is no obvious reason that the communicated value in question only be visible at the ***end*** of an approximate computation; *the time at which an approximate output becomes visible to a subsequent consumer is at least as amenable to approximation as the process of the production of the value itself.*

This observation leads directly to the exploration of forwarding the eager executed data. Conceptually, the execution of any workflow can be considered as a sequence of interdependent kernels and each kernel, in turn, as a sequence of interdependent tasks. Each task can be viewed as consuming a set of inputs and can act in turn as a producer of the values consumed by other tasks. In a precise computation, any particular execution represents a schedule, serial or parallel, that obeys the dependencies expressed among these producers and consumers. In an approximate computation, however, there can be opportunities to forward the approximate data between the tasks. The key challenge in describing and exploiting these opportunities lies in annotating the dependency between the tasks, as well as ensuring some user-defined quality controllers at the boundaries between approximate and precise computations.

To help the user approximate the CNN applications for image classification with eager execution, there are several issues have to be addressed. Firstly, in a given CNN application, we have to determine the aggressiveness of the approximation for each convolution layer (which is the latency dominate part of the entire CNN application). As discussed in some previous studies, each convolution layer have different sensitive to the final accuracy if we approximate the such layer. As the result, if we approximate too much on a accuracy-sensitive layer, it will failure the overall accuracy; on the other hand, to achieve the optimal latency drop, we should approximate more on the non-sensitive and time-consuming layers. Secondly, the framework to determine the hyper-parameter of the CNN approximation will include a large number of parameters. For example, each convolution layer may have different approximate ratio which leads to exponential approximation choices for the entire. Additionally, each hardware require different models to estimate the latency as the convolution layer may have different behaviours on different hardware. Thirdly, the framework should be light-weighted to avoid the run-time overhead.

To addressed such issues, we first proposed an approximation framework (named "Fluid") which enable the eager execution. We then proposed a performance model to help the user make the approximation selection of Fluid during the compilation time. This performance model design includes both accuracy prediction and latency estimation. The accuracy prediction model will bonded with the given CNN application while the latency estimation model is bonded with the given hardware architecture. More specifically, the producer of the CNN application should take the responsibility to generate this accuracy model. Such accuracy model should work on any hardware architecture for this CNN model. On the other hand, the hardware designer will propose the latency model to estimate the execution time of convolution layers with different parameters (include both convolution parameters and approximation parameters) which works for any CNN application. This performance model framework can significantly reduce the parameter space since the user is not required to design a performance model for each hardware-application combination. Instead, they can get the accuracy model for the CNN application and coupled with the latency model designed for the hardware architecture. Additionally, our performance model is performed during the compilation to avoid the extra storage and runtime computation.

5.2 Fluid: Eager Execution Framework for Approximate Computing

We proposed an approximate framework, named Fluid, based on Eager Execution for approximate computing. The following sections in this chapter will implement the approximation based on this framework. In the remaining part of this section, we will first introduce the programming laguage definition, compiler support and runtime support of Fluid. We then introduce several applications which might potentially get benefit from Fluid. Note that although this chapter mainly focus on the image classification tasks, Fluid can also approximate the applications in other areas.

5.2.1 Fluid Execution

In this paper, we primarily focus on the potential improvements in concurrency within a single approximable region. Our approach employs a “guard” that manages the tasks of starting, terminating, and restarting. The start and end of each task is controlled by associated “valve” functions that communicate the satisfaction of eagerness and quality constraints to the guard managing that task.

Concept	Definition
Fluid Valve	A condition function which returns *true* or *false*. It can be used to control the start and end of a task.
Fluid Guard	A processing entity that manages the execution sate of a Fluid task based on its valves or data dependence.
Fluid Member	A Fluid method or Fluid data.
Fluid Class	A type of class with Fluid members that encapsulates both the data and code subject to approximation.
Fluid Object	An instance of a Fluid class.
Fluid Method	A function defined in a Fluid class. Fluid methods may call non-Fluid methods, but the reverse is restricted.
Fluid Data	A data structure declared as Fluid. It can only be accessed by Fluid methods while in a non-final state.
Fluid Task	A dynamic instance of a Fluid method. Its execution is managed by a guard. It can be triggered by other tasks.
Fluid Region	Each Fluid object defines a Fluid region which is represented as a graph where each vertex corresponds to a Fluid task, and each edge is labeled by a Fluid data. Only leaf tasks can have end valves. Each Fluid object has a scheduler. Each Fluid task has a state machine.

Table 5.1: Major concepts in the Fluid programming paradigm.

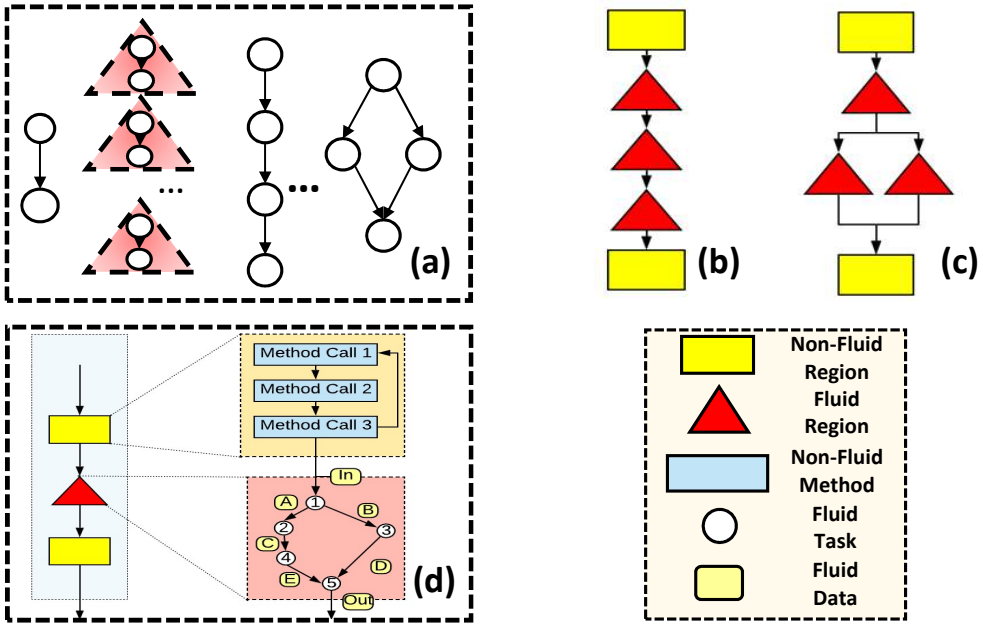


Figure 5.1: (a). Different types of task graphs in applications, red triangles are Fluid regions while the white circle represent tasks; (b). Multiple Fluid regions between non-Fluid regions; (c). A Fluid region with multiple output regions; (d). Tasks invocations within a Fluid/non-Fluid region.

5.2.1.1 Overview of Fluid Execution

Our Fluid framework builds atop an object-oriented programming model and introduces the new features described in Table 5.1. We list the Fluid concepts in the left column and their corresponding definitions in the right column.[1]

To see Fluid's major concepts in action, consider a high-level view of a program as

[1]Note that, while the way we envision and implement the Fluid execution blends very well with object-oriented computing, we believe the Fluid data concept can be exploited in other programming paradigms as well.

a set of data-dependent tasks which consist of some serial invocation of methods according to the program logic. Figure 5.1(d) shows such a breakdown, expanding an inter-region sequence, showing an approximate (Fluid) region sandwiched between two precise, sequential task regions. While each rectangular region contains some fixed ordering of task invocations, each *Fluid region* consists of multiple schedulable Fluid tasks and their associated static dataflow graph. The nodes of this graph correspond to *tasks* (dynamic instances of methods) and an edge from node A to node B captures the *dataflow* between them. A valve function is associated to the *Fluid Data* between two tasks to determine whether this data is ready for consuming. Note that, unlike in the precise program, the valve function may return true even the data is not (yet) fully produced. A task in the Fluid region can start its execution as soon as *all* of the valves that control its input data are *satisfied.* Each leaf task in a Fluid region has *end valves* that collectively constitute its associated *quality function.* Eagerly computed data cannot leave a Fluid region until satisfying the quality function. It is interesting to note that setting all valves to require the completion of antecedents within the dataflow graph will result in a "precise execution" of the entire task graph. Since multiple valves (attached to different edges) can be satisfied independently (and in parallel), multiple tasks can execute *concurrently* in a Fluid region, resulting in what we call “Fluid Concurrency” (more on this later).

Consider the example shown in Figure 5.1(d). Execution starts with a *Non-Fluid region* and generates data In. In is sent to a *Fluid region* as the input. Within the Fluid region, task1 receives In as the input and generates A and B. At some point, task2 and task3 take their inputs and generate C and D, respectively. Note that task2 and task3 may start their executions *before* task1 has finished. Also, task2 and task3 can be started at different times and run in parallel. Later, task4 takes C as the input and generates E. Finally, when the valve functions are satisfied, task5 takes D and E as input and generates Out. If Out meets the end quality check, this Fluid region has finished and the next region starts.

5.2.1.2 Fluid Concurrency

Our proposed Fluid programming paradigm enables a new type of concurrency, called “Fluid Concurrency”, which comes in two flavors: *Intra-Region Concurrency* and *Inter-Region Concurrency.* In the former one, the different tasks in a Fluid region can be executed concurrently if the corresponding valves evaluate to true. Note, however, that valve satisfaction only implies the corresponding task becomes *scheduleable* – exactly when it starts its execution depends on resource availability as well as the underlying scheduling strategy employed (Section 5.2.5 discusses our runtime system).

The initial data that triggers the execution of a region is *non-Fluid*, and the output data resulting from the execution of a region is also *non-Fluid.* As such, fluidity (that is,

$$
\begin{aligned}
\mathit{FluidStmt} &:: \mathit{FluidDef} \mid \mathit{PragmaStmt} \\
\mathit{FluidDef} &:: \texttt{\_\_Fluid\_\_}\ \mathit{class} \\
\mathit{PragmaStmt} &:: \mathit{DataPra} \mid \mathit{ValvePra} \mid \mathit{CountPra} \mid \mathit{TaskPra} \\
\mathit{DataPra} &:: \#\textbf{pragma data}\ \{\texttt{data\_type}\ d;\} \mid \\
&\quad\ \#\textbf{pragma data}\ \{\texttt{data\_type}\ *d;\} \\
\mathit{CountPra} &:: \#\textbf{pragma count}\ \{\texttt{data\_type}\ ct;\} \\
\mathit{ValvePra} &:: \#\textbf{pragma valve}\ \{\texttt{data\_type}\ v(para...);\} \\
\mathit{TaskPra} &:: \#\textbf{pragma task}\ <<< task\_name, SV, EV, \\
&\quad\ \mathit{Inputs}, \mathit{Outputs} >>> \mathit{func}()
\end{aligned}
$$

Figure 5.2: Syntax of the Fluid Language.

Fluid Concurrency) is *confined* within the boundaries of a *Fluid region*. Consequently, even an otherwise sequential (single threaded) program can take advantage of the Fluid concurrency offered by our programming paradigm. While a Fluid region has only one input, it can have multiple outputs, each driving a Fluid or non-Fluid region. As a result, multiple Fluid regions can execute concurrently, as depicted in Figure 5.1(b), leading to inter-region concurrency.

In comparison, in a conventional multi-threaded/parallel programming, execution is governed by/constrained by strict data synchronizations between independent parallel units, which limits potential parallelism due to the requirement of ensuring full accuracy. In our Fluid design, on the other hand, we can exploit concurrency between fluid tasks by the approximation of data (which is controlled by valves). We want to emphasize that, it is also possible for a program to take advantage of *both* conventional parallelism and Fluid parallelism, e.g., each individual thread of a multi-threaded program can employ intra- and inter-region Fluid concurrency. In our experiments presented later, we also evaluate such parallel application programs.

5.2.2 Program Language Support

In this section, we describe the syntax and semantics of the proposed pragma-based Fluid extensions to C++, and provide a concrete example of fluidizing a real piece of code.

5.2.3 Syntax and Semantics

Figure 5.2 describes the syntax of the Fluid language extensions. For a given statement in an application source code, the programmer can employ fluidity through explicit use of *FluidDef* and *PragmaStmt*.

FluidDef: Adding the `__Fluid__` keyword at the class declaration declares a class to be a Fluid class. A Fluid class must satisfy the following properties: i) it must contain

a public *Region()* method; ii) it must contain at least one Fluid data member and one Fluid method. Invoking *Region()* will, among other initialization functions, construct a *Fluid Task Tree* based on static data dependencies among the class's Fluid method functions (also referred to as "tasks"). The Fluid data will be shared between a parent and a child. Only Fluid methods can take Fluid data as parameters; iii) there should be only one root task for the task tree, and there should at least one leaf task; and iv) a task can only be scheduled in the *Region()* function; other non-Fluid methods cannot invoke a Fluid method. However, a Fluid method can invoke a non-Fluid method.

PragmaStmt: We use **#pragma data**, **#pragma count**, and **#pragma valve** to annotate predefined types, and we use **#pragma task** to schedule a task in our Fluid framework.

The data pragma declares a Fluid data member that will be shared between two Fluid tasks, whose value may be used to trigger the execution of dependent Fluid tasks. There are two different ways to make the declaration. If the Fluid data is a variable, (e.g., an integer x), we can indicate that x is a Fluid data directly by using "**#pragma data** {`int` x};". Second, we can also indicate that an array is a Fluid object. Take an integer array "$int * A$", as an example. We declare a Fluid data by "**#pragma data** {`int` *d};", and then initialize d by "$d \leftarrow init(A)$", meaning that d is a Fluid data, and the value of d corresponds to the values of the elements in A.

The count pragma provides introspection on the state of Fluid data by counting related events or tracking key statistics. For example, it can be used to monitor the number of updates performed on a Fluid data in a Fluid task, and it can also be used to record the average value of an array of Fluid data. Specifically, a counter *ct* is declared as a predefined type `__count__<T>`. The template type T can be any generic type offered in C++. For example, we can declare an integer count *ct* by "**#pragma count** {`int` ct};". We can use *ct* to monitor *the number of updates* to Fluid data x by invoking "*ct*++;" after each update of x.

The valve pragma includes the declaration of a *valve*, which is a predefined class in our framework. It will check whether a Fluid data is satisfied and returns either true or false at any given time. For example, a *count valve* (`valveCT`) accepts two parameters: a count *ct*, and a threshold t. x monitors the number of updates to a Fluid data d. The check function, on the other hand, keeps comparing the value of x with a programmer-defined "threshold" t, and the count valve is said to be *satisfied* when $x > t$, which means that the Fluid data d has been updated more than t times.

The task pragma is used to invoke a Fluid task. A task is a proxy of a member function in the Fluid class, and it consists of two parts: a *guard (<<< >>>)* and a *function (func())*. The guard has five fields. The first field (*task_name*) names the task. The second field (SV) indicates the start valve set. The semantics is that $func()$ *cannot* start its execution until *all* its valves are satisfied. The third field, EV, is a

set of valves for the end condition of this task – collectively, these valves implement an *output quality* function. Recall from the discussion in previous section that only a leaf task in a Fluid region can contain a non-empty set of end valves. A task is considered as having produced an output of sufficient quality only when *all* end valves are satisfied. Satisfaction of this quality property is used in re-execution decisions and task/region-level descheduling operations performed by the runtime system described in detail in Section 5.2.5. The fourth ($Inputs$) and fifth ($Outputs$) fields are two sets of Fluid data, which refer to the inputs and outputs of this task. Note that the input and output data for each task determine the topology of a Fluid region. For example, if a Fluid data d is listed in the $Outputs$ field of a task $t1$ and the $Inputs$ field of task $t2$, we can infer that there is a *data dependency* between $t1$ and $t2$. Task $t1$ is the parent node of task $t2$ in the *task tree* of this Fluid region. The function part *func()* must be a Fluid method that is a member of this Fluid class.

5.2.4 Compiler Support

We implemented a source-to-source translator from scratch that *automatically* maps a pragma-based fluidized application code into an equivalent C++ code. Since our task scheduler always works with a valid topological sort of data dependencies, the fluidized code will be correctly compiled even if all pragmas are ignored – but in that case it will *not* make any use of approximate concurrency. Note also that a fluidized program compiled by the Fluid framework can specify the task scheduling declarations in any order and would still execute correctly. To show how our translator works in practice, we focus on the edge detection code from Figure 5.3. Our compiler automatically translates this user-written code to the equivalent C++ code shown in Figure 5.4.

Firstly, in the declaration of a Fluid class, the pragmas are unwrapped, and each type is translated to our pre-defined data type (lines 3 to 8). We add a "TaskScheduler" ts at the end of the declaration, which will be utilized in future scheduling of the tasks. ts also provides internal interfaces for binding Fluid methods and Fluid data into schedulable task-execution functions, and for generating task objects that couple the guard and task-execution functions together.

Secondly, our compiler translates the task scheduling statements. For example, task $t1$ is translated to the code encapsulated between lines 20 and 22. The first statement (line 20) binds the Fluid method function with its parameters. The second statement (line 21) generates a new task-execution function, with internal task-scheduler interface, and associates it with the Fluid data members specified in its input and output sets. The third statement (line 22) uses an internal function $newTask()$ to construct a new, schedulable task entity, visible to the task scheduler, by coupling the guard thread and the task-execution function together. If the task contains start or end valves (e.g., $t2$), we make a new instance of the valves before we construct the task. That is, in lines

```
1   __Fluid__ class EdgeDetection{
2   public:
3     #pragma data {Image *d1;}
4     #pragma data {Image *d2;}
5     #pragma data {Image *d3;}
6     #pragma count {int ct;}
7     #pragma valve {ValveCT v1;}
8     #pragma valve {ValveCT v2;}
9     void Gaussian(Image *input_img, Image *output_img, count ct);
10    void Sobel(Image *input_img,  Image *output_img);
11    void Region();
12    Image *input_img, *img_after_Gaussian, *output_img;  };
13  void EdgeDetection::Region () {
14    d1->init(input_img);
15    d2->init(img_after_gaussian);
16    d3->init(output_img);
17    ct.init(0);
18    #pragma task <<<t1, {}, {}, {d1}, {d2}>>>Gaussian( input_img, img_after_gaussian, ct);
19    v1.init(ct, 0.4*input_img->size);
20    v2.init(ct, input_img->size);
21    #pragma task <<<t2, {v1}, {v2}, {t1}, {d2}, {d3}>>>Sobel(img_after_gaussian, output_img);
22    sync(t2);  };
23  void main() {
24    EdgeDetection *S1 = new EdgeDetection, *S2 = new EdgeDetection;
…     …
25    S1->Region();
26    S2->Region();
27    sync();  }
```

Figure 5.3: Programmer-level code using Fluid pragmas.

24 and 25, we create two new instances of ValveCT ($v1$ and $v2$) with the parameters of the valve. The valve will be passed to the task in line 28 by setting the parameters of its guard.

Thirdly, since some of the tasks take count as a parameter, we slightly change the variable type for the count. When we pass the count to a method, we also pass the address of the memory where we store the count value. We use "$ct.ct()$" to obtain the address of the memory where we store the value. Also, for each Fluid method that takes count as a parameter, we change the type from count to the corresponding type of pointer. For example, since ct is a count variable with the type of "int", we change its type to "int*" in line 9.

5.2.5 Runtime Support

We provide runtime support that manages the execution of each Fluid task. Specifically, our runtime system includes three inter-related components: *guard thread*, *Fluid states*, and *Fluid state machine*. Each task passes through specific states during its execution, as specified by the Fluid state machine. The execution of each task is controlled by its own guard thread based on the state machine. The guard thread is launched upon the initiation of a Fluid task and is terminated when the Fluid task finishes its execution.

As shown in Figure 5.5, a Fluid task is always in one of seven states: (**I**) initialization,

```
1   class EdgeDetection{
2   public:
3     fluid::data *d1;                    // #pragma data {RgbImage *d1;}
4     fluid::data *d2;                    // #pragma data {RgbImage *d2;}
5     fluid::data *d3;                    // #pragma data {RgbImage *d3;}
6     fluid::count<int> ct;               // #pragma count {int ct;}
7     fluid::ValveCT v1;                  // #pragma valve {ValveCT v1;}
8     fluid::ValveCT v2;                  // #pragma valve {ValveCT v2;}
9     void Gaussian(Image *input_img, Image *output_img, int *ct);
10    void Sobel(Image *input_img, Image *output_img);
11    void Region();
12    Image *input_img, *img_after_Gaussian, *output_img;
13    fluid::TaskScheduler *ts; }
14  void EdgeDetection::Region () {
15    d1->init(input_img);
16    d2->init(img_after_gaussian);
17    d3->init(output_img);
18    ct.init(0);
19    //#pragma task <<<t1, {}, {}, {d1}, {d2}>>>Gaussian(input_img, img_after_gaussian, ct);
20    auto tpb__t1 = std::bind(&EdgeDetection::Gaussian, this, input_img, img_after_Gaussian, ct.ct());
21    auto tp__t1 = ts->NewFunc<decltype(tpb__t1)>({d1}, {d2}, tpb__t1);
22    fluid::Task *t1 = ts->NewTask->newTask({}, {}, tp__t1);
23    //#pragma task <<<t2, {v1()}, {t1}>>>Sobel(img_after_gaussian, output_img);
24    auto v1_ = v1.init(ct, 0.4* input_img->size);
25    auto v2_ = v2.init(ct, input_img->size);
26    auto tpb__t2 = std::bind(&EdgeDetection::Sobel, this, img_after_Gaussian, output_img);
27    auto tp__t2 = ts->NewFunc<decltype(tpb__t1)>({d2}, {d3}, tpb__t2);
28    fluid::Task *t2 = ts->newTask({v1_}, {v2_}, tp__t2);
29    sync(t2); };
30  Void main() {
31    EdgeDetection *S1 = new EdgeDetection, S2 = new EdgeDetection;
...   ...
32    S1->Region();
33    S2->Region();
34    sync(); }
```

Figure 5.4: Code from Figure 5.3 after pragma translation.

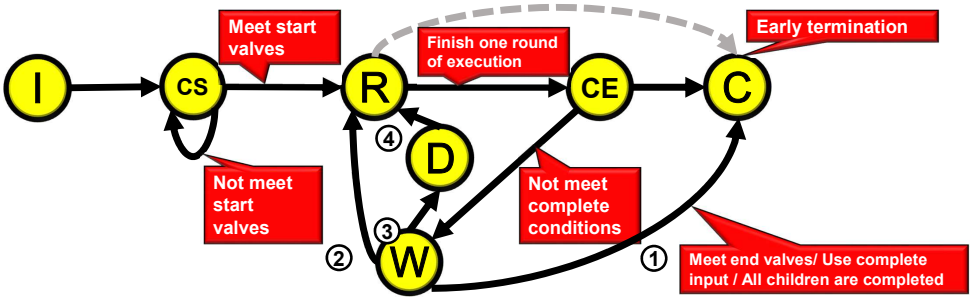


Figure 5.5: State machine for a Fluid task.

(**CS**) start checking, (**R**) running, (**CE**) end checking, (**C**) completion, (**W**) waiting, and (**D**) dependence-stalled (waiting for antecedent tasks to update task inputs before re-executing). A Fluid task is initialized in **I** state when the execution flow reaches its constructor within a *Region*() call. A separate guard thread is created by our framework to provide the guard functionality for each new task. This guard thread continuously checks the start valves before the task can start its execution. Note that, although all Fluid tasks are initialized at the beginning of a Fluid region, most of the tasks will be *blocked* in the **CS** state until their start valves are satisfied. Once all valves are satisfied, the task is eligible to run (**R**). When a Fluid task finishes

Application	Producer	Consumer	How to fluidize	tot/pragma (app)	tot/pragma (region)
K-means [133]	Assign Cluster for each pixel	Re-calculate the cluster centers	Start calculating the center before all pixels are assigned a cluster	489 / 12 / 2.5%	146 / 11 / 7.5%
Bellman-ford (BF) [134]	One relax iteration	Next relax iteration	Start next relax iteration before relax all vertices	188 / 13 / 7.0%	85 / 13 / 15.0%
Graph Coloring (GC) [135]	Find local maximum vertex	Color the vertices	Coloring selected nodes before find out all local maximum vertices	307 / 8 / 2.6%	118 / 7 / 5.9%
Edge Detection (ED) [133]	Noise removal filter	Edge detection	Start detecting edges with noisy images	245 / 9 / 3.7%	128 / 8 / 6.2%
FFT [133]	Sin/Cos value	Calculate FFT	Calculate FFT with approximate sin/cos values	459 / 17 / 3.7%	180 / 16 / 8.9%
DCT [133]	Cos value	calculate sum	Calculate sum with approximate cos values	325 / 14 / 4.3%	246 / 13 / 5.3%
Neural Network (NN) [136, 137]	Previous layer	Next layer	Start next layer before all feature calculated	427 / 17 / 4.0%	263 / 16 / 6.1%
MedusaDock (MD) [2, 33]	Calculate Docking energy of poses	Select lowest energy poses	Start selecting poses when the a portion of the poses are processed	200 / 9 / 4.5%	148 / 8 / 5.4%

Table 5.2: Characteristics of our fluidized workloads.

its execution, it will be in the **CE** state where the end conditions are checked. The transition from the **CE** state to the **C** state can be triggered by any of the following three conditions: i) a non-empty set of end valves all return true (i.e., this task is a leaf node and the output satisfies all quality measure); ii) all inputs to this task were, prior to the start of the execution, computed with non-Fluid values (which means that the output of this task is already identical to that which would have been produced in a precise execution); or iii) all the descendant Fluid tasks of this task in the task graph are in the completion state. If none of these three conditions are satisfied, the Fluid task transitions to the **W** state, waiting for further signals and preparing for potential re-execution.

5.2.6 Evaluation

We evaluate our proposed Fluid framework using eight applications. We want to emphasize that our goal in this section is *not* to defend a particular fluidization strategy for a given benchmark, and we do not claim that the particular approach used in each benchmark to fluidize it is the best one.[2] Instead, our goal is to demonstrate that Fluid computation can be used to encode various approximate computing opportunities across different applications, and that doing so can lead to reductions in execution times without much loss in accuracy. Further, we do not exhaustively tune the parameters associated with each valve – the parameters are selected illustratively rather than optimally.

5.2.6.1 Applications and Methodology

Applications: We use a 20-core Intel Xeon-based platform with 32GB memory in our experiments. We evaluate Fluid on eight applications from different areas considering both *intra-kernel* and *inter-kernel* level fluidization. Table 5.2 lists the important characteristics of the applications used in this study. For each application, we show the corresponding producer/consumer tasks and how we fluidize the application. We

[2]We postpone automatic fluidization of non-Fluid applications to a future study. The application programs in this work have been hand-fluidized.

also list the number of pragmas in the Fluid version of the code and its ratio to the entire program lines. At a high level, we can divide our applications into 4 classes, based on the type of the task graphs they possess. As shown in Figure 5.1(a), the first class of applications only have two tasks in a single Fluid region, where the first task is the *producer* and the second one is the *consumer*. This class includes i) edge detection where we first remove the noise on the image and then catch the edges on the image, and ii) MedusaDock [33, 138], which first calculates energy for each docking pose (in the context of drug discovery) and then selects several lowest energy poses. The second class of applications contain multiple Fluid regions, each containing a producer task and a consumer task. This class includes K-means and Graph coloring, both iterating over invocations of a producer-consumer pair. The third class includes Neural Network (NN) and Bellman-Ford which contain multi-task chains within a single Fluid region – within the Fluid region, all but the first and last tasks in the chain are both an eagerly invocable consumer and a producer of data for another dependent task. For the last class of applications, the task dependency graph features either or both of multi-consumer (two or more child tasks with independent start conditions on the same data structure) and multi-producer (a task dependent on multiple valves, each relating to updates to a different data structure) topologies. This class includes FFT and DCT where we calculate the value of cos/sin functions. K-means, Edge detection, FFT and DCT are implemented based on Axbench [139]. The graph coloring is based on Big-graph with the implementation in [135]. We used our own implementations for Bellman-ford and NN, and used the implementation of MedusaDock from [33].

Inputs: K-means uses three input images with different pixel diversities. For Bellman-Ford and graph coloring, we generate input graphs with different sizes and densities to study the input sensitivity of our framework. For edge detection, we choose three EM images from a publicly-available dataset [140, 141]. For FFT and DCT on the other hand, we generate input vectors/tensors with different sizes. For NN, we use Mnist [136] as the testing data with different batch sizes, and we use pdbbind [142] to evaluate MedusaDock.

Error Metrics: For K-means, we compute the squared Euclidean distance to the cluster centroid for each pixel and sum those distances together. For Bellman-Ford, we first normalize the path length for each destination vertex to the actual shortest path, and then compute the average error. For graph coloring, the error metric we employ is the graph's spectral number, *normalized* to the spectral number produced by the original (already approximate) algorithm. For edge detection, we use PSNR [143] (Peak Signal to Noise Ratio) as our error metric. Additionally, we use normalized MSE (Mean Squared Error) of the output as the error metric for FFT and DCT. Finally, We use prediction accuracy as the error metric for NN and MedusaDock. When comparing the fluidized version of an application to the baseline, we calculate the normalized accuracy as: $ABS(fluid\_ErrorMetric - baseline\_ErrorMetric)/baseline\_ErrorMetric$

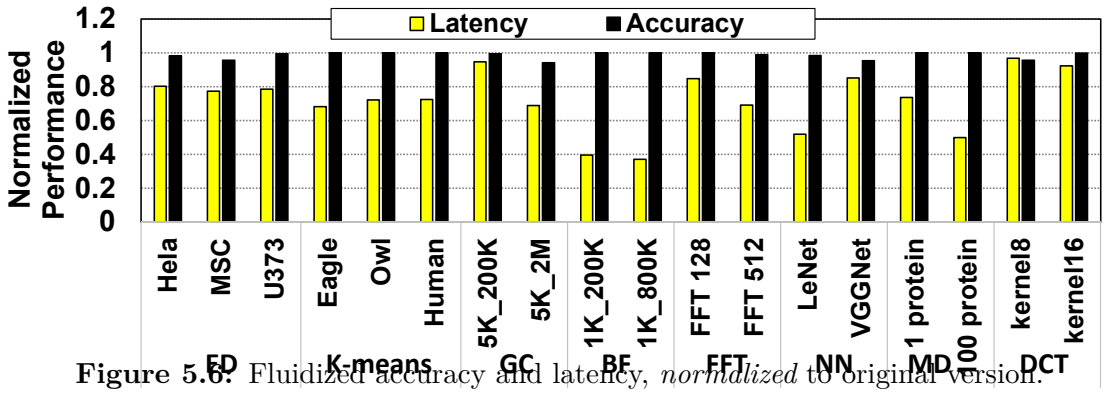


Figure 5.6: Fluidized accuracy and latency, *normalized* to original version.

5.2.6.2 Fluid Execution Time and Accuracy

Figure 5.6 plots the normalized latency and accuracy results for all applications. Here, we used the percentage valve for the start condition of the consumer task, which means that the dependent tasks start their executions when a certain fraction of the payload of the producer task has completed. We observe that the fluidized version of Bellman-Ford matches the precise output: Since, for non-pathological graphs, each vertex tends to only update its neighbors very few times [144], skipping some of the execution does not affect the final result in any significant way. Fluidized K-means also exhibits an accuracy which is similar to precise execution: In K-means, it is known that most pixels are unlikely to change their cluster membership after the first few iterations [145]. Note that the benefit of Fluid for K-means comes from overlapping the execution of the producer (assign cluster) and the consumer (re-calculate centroid) tasks, not from reducing the number of epochs. For our graph applications (Bellman-Ford and Graph Coloring), we observe that the Fluid framework achieves better speedups on dense graphs (5K_2M/1K_800k) compared to sparse graphs (5K_200K/1K_200K), as dense graphs require more computation (in the original, non-Fluid version) and, consequently, the fluidized version can skip more computations. Similarly, for FFT, DCT, MedusaDock and NN, the larger input sizes get more benefits than the smaller input sizes as the payload for large input vectors is comparatively greater than for a small input vector. On average, Fluid brings 23.4% execution time improvements across all eight applications.

5.3 Performance Modeling for Approximate Computing

After we designed and implemented the Fluid approximation framework, we will help the users to approximate the CNN models with Fluid and use our performance model to determine how to approximate. Note that the Fluid framework highly depend on the user domain knowledge to decide the approximation policy and strategy, while the

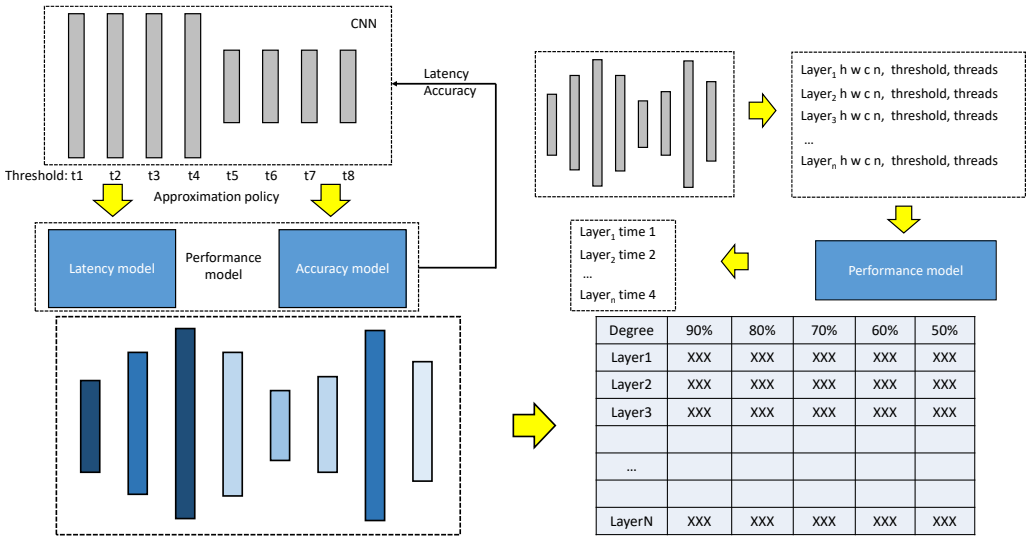


Figure 5.9: Overview of the accuracy prediction model.

performance model can automatically select an approximation method which provide the optimal performance and acceptable accuracy. In this section, we will introduce the detail of our performance model frameowkr and how we use the performance model to improve the performance of the CNN applications. We also evaluated our framework on three CNNs on two dataset and prove that our approach can improve the image classification task.

5.3.1 Motivation

There are three key points to be considered for solving the image classification task: the application (CNN model), the dataset, and the hardware platform running the application. Before we build the performance modeling framework to optimized the image classification pipeline, we should consider at which stage of the pipeline the user should invoke our framework. Indeed, to avoid the runtime overhead for the approximation, we should determine the approximation policy before we run the image classification system. As a result, the accuracy prediction and latency estimation in our performance modeling framework should be provided before we really run the CNN application with the actual input data. It is important to notice that, the structure of the CNN, the weight of the model and the hardware architecture we run the application are known before we run the CNN application with the input data. Additionally, the accuracy of the model is highly rely on the CNN model (and its pretrained weights) while the latency only depends on the hardware. Based on such assumption, we propose to build a accuracy prediction model for the given CNN model (with its weights) and a latency estimation model for the given hardware before we run the CNN application instead of build performance model for any application-hardware combination. Also, such method is efficient since we can complete the design before

Algorithm 2: Profiling the accuracy loss for each layer of a CNN model.

Input: A Convolutional Neural Network $G \equiv \{L_i | i \in 1,\ 2\ ...\ n\}$
Some sample input images $I \equiv \{I_i | i \in 1,\ 2\ ...\ k\}$

Initialization: $O_{gt} \leftarrow G.forward(I)$
for *each layer* L_i *in* G **do**
  for r *in all possible approximation rate* **do**
    Set the approximation rate of layer L_i as r and get G_i^r $O \leftarrow G_i^r.forward(I)$
    $err \leftarrow |O - O_{gt}|$
    $Model[i, r] \leftarrow err$
Return $Model$;

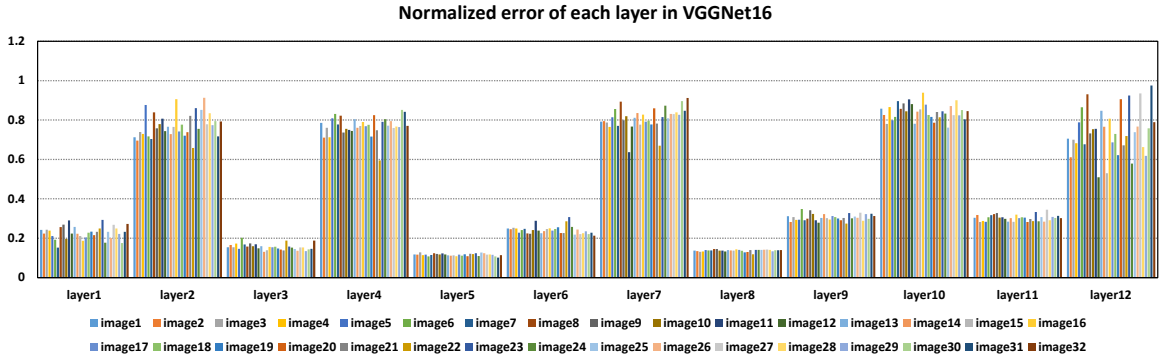


Figure 5.10: Normalized error of different images for each convolution layer under approximation. We set the approximation rate as 50%.

we run the application.

5.3.2 Accuracy Prediction

Before we studying the accuracy of a CNN model, we first look into how the accuracy will be effect if we only use part of the input for a single convolution layer. In the rest of the paper, we define the term "approximation rate", "approximation degree" and "approximation aggressiveness" as the percentage of the workload that we really compute for a convolution layer. Suppose there are two convolution layers L_1 and L_2, which are computed as follow:

$$O_1 = I \odot W_1 \tag{5.1}$$

$$O_2 = O_1 \odot W_2 \tag{5.2}$$

Where I indicate the input feature map of L_1, O_1 and O_2 indicate the output feature maps for L_1 and L_2, respectively, the W_1 and W_2 are the weight matrices for L_1 and L_2 respectively. One should note that the output of L_1 is the input of L_2. If L_2 only

consume part of the input data (in the other word, we only partially compute L_1), the accuracy will be effected. The question we would like to ask is, what is the relation between the approximation and the final accuracy loss. Take the following case as an example, suppose we only run 50% of L_1 and L_2 only consume 50% of the input data, the equation of the convolution calculation will be:

$$O_1^1 = I \odot W_1^1 \tag{5.3}$$

$$O_2' = O_1^1 \odot W_2 \tag{5.4}$$

$$O_2' = I \odot W_1^1 \odot W_2 \tag{5.5}$$

Here $W_1 = [w_1^1, W_1^2]$ which indicate that W_1 is the concatenation of w_1^1 and W_1^2 along the output channel dimension. We can estimate the error of the approximation by calculating (2) - (5):

$$O_2 - O_2' = I \odot W_1 \odot W_2 - I \odot W_1^1 \odot W_2 \tag{5.6}$$

$$O_2 - O_2' = I \odot [W_1^1, W_1^2] \odot W_2 - I \odot W_1^1 \odot W_2 \tag{5.7}$$

$$O_2 - O_2' = [O_1^1, O_1^2] \odot W_2 - O_1^1 \odot W_2 \tag{5.8}$$

$$O_2 - O_2' = O_1^2 \odot W_2 \tag{5.9}$$

$$O_2 - O_2' = I \odot W_1^2 \odot W_2 \tag{5.10}$$

The normalized error will be $(I \odot W_1^2 \odot W_2)/I$. This indicate that, if the I follow the same distribution, the error should be similar for different inputs.

To verify the previous assumption, we evaluate each convolution layers in VGGNet16 and calculate the error under the approximation with different input data. Figure 5.10 shows the normalized error of each convolution layer in VGGNet16 [137]. One can observe that, although different convolution layers have variance of error sensitivity, for each single layer, different input image does not have significant impact on the accuracy loss. This indicates that the weight matrix is the only factor which affect the accuracy of a CNN model.

Based on the previous assumption and observation, we design a accuracy prediction model which only rely on the trained weight of a given CNN application. As shown in Figure 5.9 We use some example image data to profile the accuracy sensitivity of each convolution layer and use such profiling results for the entire performance modeling framework. The detail of the profiling process is shown in Algorithm 2. In general, for each convolution layer, we gradually increase the approximation aggressiveness (what is the percentage we compute for that layer), and record the accuracy loss.

	Feature	Type
input	height	int
	width	int
	channel	int
	batch size	int
output	height	int
	width	int
	channel	int
	relu?	bool
	pad?	bool
	rate	float
	scale	float
	threads	int
	index	int

Table 5.3: Features included for training the latency estimation model.

5.3.3 Latency Estimation

The second part of the performance model include a latency estimation model which predict the execution time of each single convolution layer with a given approximation degree.

Table 5.3 list the configuration features for the latency estimation. In general, to estimate the latency of a convolution layer, we should include the parameter of that layer (i.e., input/output channel, height, width, batch size). We also include the number of threads for running the CNN model as the feature to learning the performance of a given system under specific computing resources. Additionally, We include the approximation rate for each convolution layer since the entire framework target on proposing the optimal approximation strategy (approximation rate for each layer). Finally, we indicate the index of each convolution layer since the topology of the network will effect the performance.

To collect the training data for the latency estimation model, we create a simulation dataset. We run a large mount of single convolution layer on a given platform with different convolution parameters, number of threads and approximation ratio. To run a single convolution layer with the given approximate ratio, we set the timestamp at the start of the execution of the layer and end the timestamp when the computation amount reaches that ratio. Our simulation dataset contains 50K data points which will be used to train the latency estimation model. We also create a real-model dataset from VGGNet [] to evaluate the correctness of our simulation dataset. This is because, for the convolution layer with exactly parameter might have different execution time when running it dependently or running it in the real model. We running VGGNet with different batch size and different number of threads. We also

set different approximation ratio of each convolution layer of the VGGNet. This real-model dataset contains 8K data points where each data points indicate a convolution layer in the VGGNet with different settings.

We show the overview of the latency estimation model in Figure 5.8. It is mainly a multi-layer perceptron (MLP) model with 4 fully-connected layers and each layer contains 128 neurons. The input of the model has a size of $N \times F$ where N is the number of data points and F is the features size (13 in our case). The output of the model is a numerical value which indicate the estimated execution time of each convolution layer. Before we feeding the model with the data, we normalized the data by StandardScaler provided by sklearn library. During the training, we use Adam optimizer with the learning rate as 0.001. We select mean square error (MSE) loss as the loss function. We train the model for up to 5000 epochs or if the loss get converged.

5.3.4 Performance Model during Compilation

After we obtained both accuracy model and the latency model, we assign the optimal aggressiveness of the approximation for each convolution layer follow Algorithm 3. At the beginning, we set the threshold (the percentage of the workload we will run of a given convolution layer) to 100%. After that, we iteratively reduce the threshold of the least sensitive layer by the step d. The sensitivity s_i of layer i is defined by:

$$S_i = (exe(t_i) - exe(t_i - d))/(err(t_i - d) - err(t_i))$$

; where t_i is the current threshold of layer i, $exe(t)$ is the function which estimate the execution time of a convolution layer if we only run $t\%$ of the workload whereas $err(t)$ is the function to predict the accuracy loss if we only run $t\%$ of a given convolution layer. The former function is provided by the latency model while the later can be obtained from the accuracy model. In general, the sensitivity of a layer indicate the cost-efficiency if we reduce the threshold of a layer. In each iteration, we first select the most efficient layer to reduce its threshold and then evaluate the accuracy of the entire model with some pieces of example data. We continue this loop until the accuracy loss break the tolerance set by the user. This mechanism automatically tune the threshold for each layer during the compilation time and does not include the runtime overhead.

Algorithm 3: Identify the optimal approximation solution for a given CNN.

Input: A Convolutional Neural Network $G \equiv \{L_i | i \in 1,\ 2\ ...\ n\}$
Some sample input images $I \equiv \{I_i | i \in 1,\ 2\ ...\ k\}$
Accuracy loss upper-bound max_err
Step of approximation rate for each layer r
A latency estimation model $Model_l$
An accuracy prediction model $Model_a$

Initialization:
$O_{gt} \leftarrow G.forward(I)$
for *each layer* L_i *in* G **do**
  $th[i] \leftarrow 100\%$
  $error'[i] \leftarrow 0$
  $error[i] \leftarrow Model_a(i, 100\% - r)$
  $time'[i] \leftarrow Model_l(i, 100\%)$
  $time[i] \leftarrow Model_l(i, 100\% - r)$
  $efficient[i] \leftarrow (time'[i] - time[i])/error[i]$
$O \leftarrow O_{gt}$
while $|(O - O_{gt})/O_{gt}| \leq max\_err$ **do**
  $i \leftarrow arcmax(efficient)$
  $th[i] \leftarrow th[i] - r$
  Set the approximation rate of all layers of G with th and get G_{th}
  $O \leftarrow G_{th}.forward(I)$
  $error'[i] \leftarrow error[i]$
  $error[i] \leftarrow Model_a(i, th[i] - r)$
  $time'[i] \leftarrow time[i]$
  $time[i] \leftarrow Model_l(i, th[i] - r)$
  $efficient[i] \leftarrow (time'[i] - time[i])/error[i]$
Return th;

5.3.5 Experiments

5.3.5.1 Evaluation on Simulation Dataset

We first evaluate the latency estimation model on the simulation dataset. We collected the data by running the convolution layers with different configurations (convolution parameter, system parameter, approximation parameter) and thus generated a dataset contains 50K data points. After that, we randomly select 80% of the data as the training set while the remaining 20% as the testing set. We train our latency estimation model with the data in the training set and evaluate the data in the testing set. We plot the scatter plots for the estimation of the testing set in Figure 5.11. We can observe that our latency estimation model achieved a RMSE of 0.29ms and a R2 score of 0.90. This indicate that our performance model can achieve a great success for

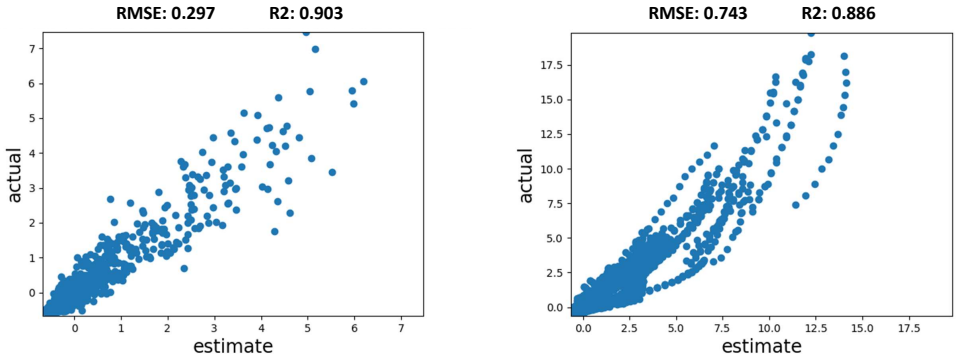


Figure 5.12: Correlation results for real-model dataset.

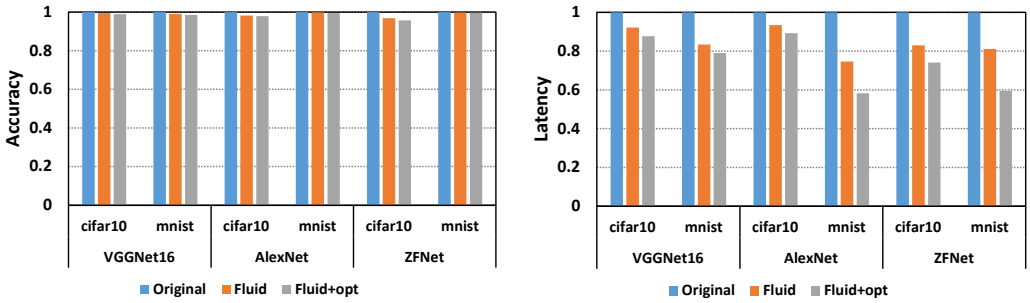


Figure 5.14: Normalized latency.

estimating the execution time of a convolution layer with a given approximation rate.

5.3.5.2 Evaluation on Real-model Dataset

The second evaluation focus on estimating the data points in the real-estimation dataset. To collect the data points in the real-model dataset, we run the VGGNet16 for multiple times with different number of threads and set different approximation degree for each convolution layer. This results a dataset which contains 10K data points. In this case, we train our latency estimation model on the simulation dataset generated in the previous section, and evaluate our model with the real-model dataset. The reason we do this evaluation is that we want to train a model with a general dataset and works on the estimation of the convolution layers in any CNN. We plot the scatter plots for the estimation of the testing set in Figure 5.12. We can observe that our latency estimation model achieved a RMSE of 0.74ms and a R2 score of 0.88. This results indicate that our latency estimation model trained with a general simulation dataset can work on predicting the execution time of a convolution layer from any other CNN models.

CNN	Dataset	Approximation rates
VGG16	cifar10	100, 100, 65, 95, 60, 85, 95, 60, 75, 100, 60, 60, 90
	mnist	70, 70, 70, 80, 70, 70, 80, 75, 80, 80, 70, 70, 80
AlexNet	cifar10	85, 100, 85, 70, 70
	mnist	50, 75, 45, 45, 45
ZFNet	cifar10	80, 65, 65, 65, 70
	mnist	80, 45, 45, 45, 60

Table 5.4: Optimal approximation rate of each convolution layer for different CNN models on each dataset

5.3.5.3 Improving CNN with Performance Modeling

Finally, we evaluate our end-to-end framework guided by the performance model during the compiler session. We test the framework on three CNN models with three approaches. The first approach is the baseline method where we run the entire CNN model without approximation. The second approach is an stat-of-the-art approximation approach [146] which allows fixed approximation degree for each convolution layer. The third approach employed our framework which automatically determine the approximation aggressiveness before running the CNN model. We set the accuracy tolerance as 5% and the framework will figure out the optimal approximation rate of each convolution layer in the given CNN to achieve the best latency performance. We plot the accuracy results in Figure 5.13 and latency results in Figure 5.14. We list the optimal approximation rate of each convolution layer generated by our framework in Table 5.4. We can observe that our performance model can help the user get a better performance compare to other approaches. More specifically, we can achieve 22.2% latency reduction on average, while keep the same level of the accuracy, compared to the baseline approach. When compared to [146], we get 9.4% additional latency reduction. We achieve the latency gain since our approach can allow the user approximate more on the higher sensitivity layer and take less approximation on less sensitivity layer.

5.4 Discussion

In this section, we would like to discuss how our performance model guided approach compared against other approximation strategies. As we mentioned earlier, there are several existing approximation studies target on CNNs, which include pruning, quantization and knowledge distillation. However, in this thesis, we employ the eager execution as the approximation method. Compare with other approximation strategies have the following advantages. Firstly, some CNN model approximation methods require to re-train the model. For example, the pruning methods first remove

part of the values from the weight matrix. To recover the accuracy, the user then re-train the pruned weight matrix of the model with the training data. The knowledge distillation mechanism also require the re-training for the student model to recover the accuracy. Secondly, the quantization requires the hardware support to implement the mathematics operators with less precision data. Additionally, we can accommodate eager execution with other approximations (e.g., quantization). More specifically, the user can simultaneously issue the calculation with different precision. The next layer can start the execution before the most precise results completed. As a result, we choose the eager execution as the approximation method in this thesis for the Fluid framework because it is easy to implement without re-training and the requirement of hardware support. Finally, we want to emphasize that, our performance model can also adopt with other approximation strategies.

Chapter 6

Conclusions and Future Work

6.1 Summary of Dissertation Contributions

Machine learning has become an integral part of the technology influencing many application domains. Thus, optimizing the computational consumption as well as improving the existing pipeline execution performance with machine learning techniques has been a critical area of research in recent years. This dissertation aims to understand and address some of the key bottlenecks in improving the performance for the applications in different areas with the help of machine learning methodologies.

First, we identify computational inefficiencies in the biomedical image processing tasks where users redundantly compute the unimportant areas. To address these inefficiencies, we propose tiling based scheme to minimize the redundant computation; and edge-detection based method to quickly detect the edges on an image and then only process the pixels near the edges. Further, we examine the performance benefit from the proposed two schemes on three cell tracking dataset.

Second, we target on the protein-ligand docking problem in drug discovery, and propose a CNN based approach (MedusaNet) and a GNN based approach (MedusaGraph), to improve the throughput of the docking. The existing docking software rely on a significantly long sampling process which limit the efficiency of the docking software. With our proposed techniques, the user can detect a good docking pose for a protein-ligand pair at very early stage.

Third, we explore the availability of the approximation in CNN applications for image

classification. Motivated by these, we propose Fluid to take advance of consuming partial of the input feature maps for each convolution layer. We implement our design with a language support, compiler support and runtime support. We further design a performance modeling framework to help the user to automatically identify the optimal approximation solution for a given CNN application on Fluid.

6.2 Future Research Directions

As we discussed in previous chapters, this thesis aim to accelerate applications with machine learning techniques in terms of reducing input data size, improve the algorithm and optimize the system. To explain our contribution, we applied and evaluate our designs on biomedical image processing, protein-ligand docking and CNN approximation. We believe our approaches can also accelerate other applications in many different domains. While this thesis already many application accelerating scenarios in the above chapters, it is still while to extend our design considerations in some other potential direction and benefit more applications. As a result, we summarize some future research directions and discuss them in the following sections.

6.2.1 Apply Reinforcement Learning on Protein-ligand Docking

As we discussed in chapter 4, the main bottleneck of the protein-ligand docking process is the sampling step, where the users have to samples thousands of candidate docking poses for each protein-ligand pair and then use the scoring function to select the best docking pose among them. Although our MedusaNet can significantly reduce the searched pose among and the MeudsaGraph can directly refine the candidate pose without the sampling process, such approaches still have the limitation. In general, when the MedusaGraph predict the movement of each atoms, we did not applied any geometry restraints for the movement. It is possible that the output ligand is not in a reasonable conformation after the movement (e.g., the bond angle or the bond length is incorrect). Our current solution is to utilize some molecule optimization software (e.g., openbabel) to optimize the bond lengths and bond angles. To solve that issue, one future research direction is apply the reinforcement learning to predicting the movement of each atoms during the docking process. In reinforcement learning, we can set up what is the legal movement. For example, we can rotate the molecule along a rotatable bond. We can also shift or rotate the entire molecule. However, we cannot broke an existing covalent bond between two atoms. By employing the reinforcement learning technique, we can directly predict the final docking pose for a protein-ligand pair while guarantee the generated pose is always in a reasonable conformation.

6.2.2 Exploration of the Cross-docking

Recently, most of the research on protein-ligtand docking focus on self-docking instead of cross-docking. In the self-docking, the initial location of the atoms in the binding site is already prepared as the atom location in the ground-truth pose. On the other hand, in the cross-docking, the protein is initialized as an arbitrary pose and we will predict the docking structure after the docking process. The docking pocket will change its conformation when docking with different ligand. As a result, cross-docking will be a more difficult problem than self-docking. In the future research direction, we would like to apply the pose generation approaches (e.g., MedusaGraph, Reinforcement Learning) on the cross-docking dataset. Compare to previous studies, our methods moving the atoms in the protein-ligand complex directly and are flexible to the confirmation change of the binding pocket.

Bibliography

[1] Jiang, H., A. Sarma, J. Ryoo, J. B. Kotra, M. Arunachalam, C. R. Das, and M. T. Kandemir (2018) "A learning-guided hierarchical approach for biomedical image segmentation," in *2018 31st IEEE International System-on-Chip Conference (SOCC)*, IEEE, pp. 227–232.

[2] Jiang, H., M. Fan, J. Wang, A. Sarma, S. Mohanty, N. V. Dokholyan, M. Mahdavi, and M. T. Kandemir (2020) "Guiding Conventional Protein–Ligand Docking Software with Convolutional Neural Networks," *Journal of Chemical Information and Modeling*, **60**(10), pp. 4594–4602.

[3] Bojarski, M., D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, et al. (2016) "End to end learning for self-driving cars," *arXiv preprint arXiv:1604.07316*.

[4] Amodei, D., S. Ananthanarayanan, R. Anubhai, J. Bai, E. Battenberg, C. Case, J. Casper, B. Catanzaro, Q. Cheng, G. Chen, et al. (2016) "Deep speech 2: End-to-end speech recognition in english and mandarin," in *International conference on machine learning*, pp. 173–182.

[5] Ryoo, J., M. Fan, X. Tang, H. Jiang, M. Arunachalam, S. Naveen, and M. T. Kandemir (2019) "Architecture-Centric Bottleneck Analysis for Deep Neural Network Applications," in *2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, IEEE, pp. 205–214.

[6] Boroumand, A., S. Ghose, B. Akin, R. Narayanaswami, G. F. Oliveira, X. Ma, E. Shiu, and O. Mutlu (2021) "Google neural network models for edge devices: Analyzing and mitigating machine learning inference bottlenecks," in *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, IEEE, pp. 159–172.

[7] Nabavinejad, S. M., S. Reda, and M. Ebrahimi (2022) "Coordinated Batching and DVFS for DNN Inference on GPU Accelerators," *IEEE Transactions on Parallel and Distributed Systems*, **33**(10), pp. 2496–2508.

[8] Jouppi, N., C. Young, N. Patil, and D. Patterson (2018) "Motivation for and evaluation of the first tensor processing unit," *ieee Micro*, **38**(3), pp. 10–19.

[9] Bai, L., Y. Zhao, and X. Huang (2018) "A CNN accelerator on FPGA using depthwise separable convolution," *IEEE Transactions on Circuits and Systems II: Express Briefs*, **65**(10), pp. 1415–1419.

[10] Ma, L., Z. Xie, Z. Yang, J. Xue, Y. Miao, W. Cui, W. Hu, F. Yang, L. Zhang, and L. Zhou (2020) "Rammer: Enabling Holistic Deep Learning Compiler Optimizations with {rTasks}," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pp. 881–897.

[11] Cummins, C., Z. V. Fisches, T. Ben-Nun, T. Hoefler, M. F. O'Boyle, and H. Leather (2021) "Programl: A graph-based program representation for data flow analysis and compiler optimizations," in *International Conference on Machine Learning*, PMLR, pp. 2244–2253.

[12] Arunarani, A., D. Manjula, and V. Sugumaran (2019) "Task scheduling techniques in cloud computing: A literature survey," *Future Generation Computer Systems*, **91**, pp. 407–415.

[13] Zhang, P. and M. Zhou (2017) "Dynamic cloud task scheduling based on a two-stage strategy," *IEEE Transactions on Automation Science and Engineering*, **15**(2), pp. 772–783.

[14] Miguel, J. S., J. Albericio, N. E. Jerger, and A. Jaleel (2016) "The Bunker Cache for Spatio-value Approximation," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-49, IEEE Press, Piscataway, NJ, USA, pp. 43:1–43:12.
URL `http://dl.acm.org/citation.cfm?id=3195638.3195690`

[15] Han, J. and M. Orshansky (2013) "Approximate Computing: An Emerging Paradigm For Energy-Efficient Design," in *IEEE ETS*.

[16] Bottou, L. (2010) "Large-scale machine learning with stochastic gradient descent," in *Proc. 19th Int. Conf. Comput. Statist.*

[17] Huang, W., D. He, X. Yang, Z. Zhou, D. Kifer, and C. L. Giles (2016) "Detecting Arbitrary Oriented Text in the Wild with a Visual Attention Model," in *Proceedings of the 2016 ACM on Multimedia Conference*, MM '16, ACM, New York, NY, USA, pp. 551–555.
URL `http://doi.acm.org/10.1145/2964284.2967282`

[18] Tung, F. and G. Mori (2018) "Deep neural network compression by in-parallel pruning-quantization," *IEEE transactions on pattern analysis and machine intelligence*, **42**(3), pp. 568–579.

[19] Xu, Y., Y. Wang, A. Zhou, W. Lin, and H. Xiong (2018) "Deep neural network compression with single and multiple level quantization," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 32.

[20] Zhuang, Z., M. Tan, B. Zhuang, J. Liu, Y. Guo, Q. Wu, J. Huang, and J. Zhu (2018) "Discrimination-aware channel pruning for deep neural networks," *Advances in neural information processing systems*, **31**.

[21] He, Y., X. Zhang, and J. Sun (2017) "Channel pruning for accelerating very deep neural networks," in *Proceedings of the IEEE international conference on computer vision*, pp. 1389–1397.

[22] Dong, D., Z. Xu, W. Zhong, and S. Peng (2018) "Parallelization of molecular docking: a review," *Current Topics in Medicinal Chemistry*, **18**(12), pp. 1015–1028.

[23] Song, L., Y. Wang, Y. Han, X. Zhao, B. Liu, and X. Li (2016) "C-Brain: A deep learning accelerator that tames the diversity of CNNs through adaptive data-level parallelization," in *Proceedings of the 53rd Annual Design Automation Conference*, pp. 1–6.

[24] Ronneberger, O., P. Fischer, and T. Brox (2015) "U-net: Convolutional networks for biomedical image segmentation," in *International Conference on Medical Image Computing and Computer-Assisted Intervention*, Springer, pp. 234–241.

[25] Ruddigkeit, L., R. Van Deursen, L. C. Blum, and J.-L. Reymond (2012) "Enumeration of 166 billion organic small molecules in the chemical universe database GDB-17," *J. Chem. Inf. Model.*, **52**(11), pp. 2864–2875.

[26] Declerck, P. J., F. Darendeliler, M. Goth, S. Kolouskova, I. Micle, C. Noordam, V. Peterkova, N. N. Volevodz, J. Zapletalová, and M. B. Ranke (2010) "Biosimilars: controversies as illustrated by rhGH," *Curr. Med. Res. Opin.*, **26**(5), pp. 1219–1229.

[27] Hunt, J. P., S. O. Yang, K. M. Wilding, and B. C. Bundy (2017) "The growing impact of lyophilized cell-free protein expression systems," *Bioengineered*, **8**(4), pp. 325–330.

[28] Di Stasi, A., S.-K. Tey, G. Dotti, Y. Fujita, A. Kennedy-Nasser, C. Martinez, K. Straathof, E. Liu, A. G. Durett, B. Grilley, H. Liu,

C. R. Cruz, B. Savoldo, A. P. Gee, J. Schindler, R. A. Krance, H. E. Heslop, D. M. Spencer, C. M. Rooney, and M. K. Brenner (2011) "Inducible apoptosis as a safety switch for adoptive cell therapy," *N. Engl. J. Med.*, **365**, pp. 1673–1683.

[29] Convertino, M., J. Das, and N. V. Dokholyan (2016) "Pharmacological chaperones: design and development of new therapeutic strategies for the treatment of conformational diseases," *ACS Chem. Biol.*, **11**(6), pp. 1471–1489.

[30] Ding, F., S. Yin, and N. V. Dokholyan (2010) "Rapid flexible docking using a stochastic rotamer library of ligands," *J. Chem. Inf. Model.*, **50**(9), pp. 1623–1632.

[31] Wang, J. and N. V. Dokholyan (2019) "MedusaDock 2.0: Efficient and Accurate Protein-Ligand Docking With Constraints." *J. Chem. Inf. Model.*, **59**(6), pp. 2509–2515.
URL http://www.ncbi.nlm.nih.gov/pubmed/30946779http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=PMC6597311

[32] Ding, F. and N. V. Dokholyan (2013) "Incorporating backbone flexibility in MedusaDock improves ligand-binding pose prediction in the CSAR2011 docking benchmark." *J. Chem. Inf. Model.*, **53**(8), pp. 1871–9.
URL http://www.ncbi.nlm.nih.gov/pubmed/23237273http://pubs.acs.org/doi/abs/10.1021/ci300478yhttp://pubs.acs.org/doi/10.1021/ci300478y

[33] Yin, S., L. Biedermannova, J. Vondrasek, and N. V. Dokholyan (2008) "MedusaScore: an accurate force field-based scoring function for virtual drug screening," *Journal of chemical information and modeling*, **48**(8), pp. 1656–1662.

[34] Morris, G. M., R. Huey, W. Lindstrom, M. F. Sanner, R. K. Belew, D. S. Goodsell, and A. J. Olson (2009) "AutoDock4 and AutoDockTools4: Automated docking with selective receptor flexibility," *J. Comput. Chem.*, **30**(16), pp. 2785–2791.

[35] ——— (2009), "AutoDock4 and AutoDockTools4: Automated docking with selective receptor flexibility," .
URL http://www.ncbi.nlm.nih.gov/pubmed/19399780http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=PMC2760638

[36] Verdonk, M. L., J. C. Cole, M. J. Hartshorn, C. W. Murray, and R. D. Taylor (2003) "Improved protein¨Cligand docking using GOLD," *Proteins: Struct., Funct., Bioinf.*, **52**(4), pp. 609–623.

[37] Liu, M. and S. Wang (1999) "MCDOCK: a Monte Carlo simulation approach to the molecular docking problem," *J. Comput.-Aided Mol. Des.*, **13**(5), pp. 435–451.

[38] Roisman, L. C., J. Piehler, J. Y. Trosset, H. A. Scheraga, and G. Schreiber (2001) "Structure of the interferon-receptor complex determined by distance constraints from double-mutant cycles and flexible docking." *Proc. Natl. Acad. Sci. U. S. A.*, **98**(23), pp. 13231–13236.
URL http://www.ncbi.nlm.nih.gov/pubmed/11698684http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=PMC60853

[39] Wallach, I., M. Dzamba, and A. Heifets (2015) "AtomNet: a deep convolutional neural network for bioactivity prediction in structure-based drug discovery," *ArXiv:1510.02855*.

[40] Ragoza, M., J. Hochuli, E. Idrobo, J. Sunseri, and D. R. Koes (2017) "Protein–ligand scoring with convolutional neural networks," *J. Chem. Inf. Model.*, **57**(4), pp. 942–957.

[41] Jiménez, J., M. Skalic, G. Martinez-Rosell, and G. De Fabritiis (2018) "K deep: Protein–ligand absolute binding affinity prediction via 3d-convolutional neural networks," *J. Chem. Inf. Model.*, **58**(2), pp. 287–296.

[42] Cang, Z. and G.-W. Wei (2017) "TopologyNet: Topology based deep convolutional and multi-task neural networks for biomolecular property predictions," *PLoS Comput. Biol.*, **13**(7), p. e1005690.

[43] Lim, J., S. Ryu, K. Park, Y. J. Choe, J. Ham, and W. Y. Kim (2019) "Predicting drug–target interaction using a novel graph neural network with 3D structure-embedded graph representation," *Journal of chemical information and modeling*, **59**(9), pp. 3981–3988.

[44] Thölke, P. and G. De Fabritiis (2022) "TorchMD-NET: Equivariant Transformers for Neural Network based Molecular Potentials," *arXiv preprint arXiv:2202.02541*.

[45] Srinivasan, P. P., S. J. Heflin, J. A. Izatt, V. Y. Arshavsky, and S. Farsiu (2014) "Automatic segmentation of up to ten layer boundaries in SD-OCT images of the mouse retina with and without missing layers due to pathology," *Biomed. Opt. Express*, **5**(2), pp. 348–365.
URL http://www.osapublishing.org/boe/abstract.cfm?URI=boe-5-2-348

[46] Magnusson, K. (2011), "Cell tracking for automated analysis of timelapse microscopy," .

[47] Wang, R. (2016) "Edge detection using convolutional neural network," in *International Symposium on Neural Networks*, Springer, pp. 12–20.

[48] Liu, Y., M.-M. Cheng, X. Hu, K. Wang, and X. Bai (2017) "Richer Convolutional Features for Edge Detection," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[49] Yu, Z., C. Feng, M.-Y. Liu, and S. Ramalingam (2017) "CASENet: Deep Category-Aware Semantic Edge Detection," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[50] Ciresan, D., A. Giusti, L. M. Gambardella, and J. Schmidhuber (2012) "Deep Neural Networks Segment Neuronal Membranes in Electron Microscopy Images," in *Advances in Neural Information Processing Systems 25*, Curran Associates, Inc., pp. 2843–2851.

[51] Long, J., E. Shelhamer, and T. Darrell (2015) "Fully Convolutional Networks for Semantic Segmentation," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[52] Badrinarayanan, V., A. Kendall, and R. Cipolla (2015) "SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation," `arXiv:1511.00561`.

[53] Seyedhosseini, M., M. Sajjadi, and T. Tasdizen (2013) "Image Segmentation with Cascaded Hierarchical Models and Logistic Disjunctive Normal Networks," in *The IEEE International Conference on Computer Vision (ICCV)*.

[54] Stollenga, M. F., W. Byeon, M. Liwicki, and J. Schmidhuber (2015) "Parallel Multi-Dimensional LSTM, With Application to Fast Biomedical Volumetric Image Segmentation," in *Advances in Neural Information Processing Systems 28* (C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, eds.), Curran Associates, Inc., pp. 2998–3006.

[55] Hochreiter, S. and J. Schmidhuber (1997) "Long Short-Term Memory," *Neural Computation*, **9**(8), pp. 1735–1780, `https://doi.org/10.1162/neco.1997.9.8.1735`.
URL `https://doi.org/10.1162/neco.1997.9.8.1735`

[56] Schmidt, U., M. Weigert, C. Broaddus, and G. Myers (2018) "Cell detection with star-convex polygons," in *International Conference on Medical Image Computing and Computer-Assisted Intervention*, Springer, pp. 265–273.

[57] Stringer, C., M. Michaelos, and M. Pachitariu (2020) "Cellpose: a generalist algorithm for cellular segmentation," *bioRxiv*.

[58] Redmon, J., S. Divvala, R. Girshick, and A. Farhadi (2016) "You only look once: Unified, real-time object detection," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 779–788.

[59] He, K., G. Gkioxari, P. Dollár, and R. B. Girshick (2017) "Mask R-CNN," *CoRR*, **abs/1703.06870**, 1703.06870.
URL http://arxiv.org/abs/1703.06870

[60] Nedumpully-Govindan, P., D. B. Jemec, and F. Ding (2016) "CSAR Benchmark of Flexible MedusaDock in Affinity Prediction and Nativelike Binding Pose Selection," *J. Chem. Inf. Model.*, **56**(6), pp. 1042–1052.
URL http://pubs.acs.org/doi/10.1021/acs.jcim.5b00303

[61] Whitley, D. (1994) "A genetic algorithm tutorial," *Statistics and Computing*, **4**(2), pp. 65–85.

[62] Kramer, B., M. Rarey, and T. Lengauer (1999) "Evaluation of the FLEXX incremental construction algorithm for protein¨Cligand docking," *Proteins: Struct., Funct., Bioinf.*, **37**(2), pp. 228–241.

[63] Pellecchia, M. (2009) "Fragment-based drug discovery takes a virtual turn," *Nat. Chem. Biol.*, **5**(5), pp. 274–275.
URL http://www.nature.com/doifinder/10.1038/nchembio0509-274

[64] Li, J., A. Fu, and L. Zhang (2019) "An overview of scoring functions used for protein–ligand interactions in molecular docking," *Interdiscip. Sci.: Comput. Life Sci.*, pp. 1–9.

[65] Yang, Y., F. C. Lightstone, and S. E. Wong (2013) "Approaches to efficiently estimate solvation and explicit water energetics in ligand binding: the use of WaterMap," *Expert Opin. Drug Discovery*, **8**(3), pp. 277–287.

[66] Michel, J., J. Tirado-Rives, and W. L. Jorgensen (2009) "Prediction of the water content in protein binding sites," *J. Phys. Chem. B*, **113**(40), pp. 13337–13346.

[67] Ross, G. A., G. M. Morris, and P. C. Biggin (2012) "Rapid and accurate prediction and scoring of water molecules in protein binding sites," *PloS One*, **7**(3).

[68] Uehara, S. and S. Tanaka (2016) "AutoDock-GIST: Incorporating thermodynamics of active-site water into scoring function for accurate protein-ligand docking," *Molecules*, **21**(11), p. 1604.

[69] Kumar, A. and K. Y. Zhang (2013) "Investigation on the effect of key water molecules on docking performance in CSARdock exercise," *J. Chem. Inf. Model.*, **53**(8), pp. 1880–1892.

[70] Sun, H., Y. Li, D. Li, and T. Hou (2013) "Insight into crizotinib resistance mechanisms caused by three mutations in ALK tyrosine kinase using free energy calculation approaches," *J. Chem. Inf. Model.*, **53**(9), pp. 2376–2389.

[71] Chaskar, P., V. Zoete, and U. F. Rohrig (2017) "On-the-fly QM/MM docking with attracting cavities," *J. Chem. Inf. Model.*, **57**(1), pp. 73–84.

[72] Muegge, I. and Y. C. Martin (1999) "A general and fast scoring function for protein- ligand interactions: a simplified potential approach," *J. Med. Chem.*, **42**(5), pp. 791–804.

[73] Gohlke, H., M. Hendlich, and G. Klebe (2000) "Knowledge-based scoring function to predict protein-ligand interactions," *J. Mol. Biol.*, **295**(2), pp. 337–356.

[74] Velec, H. F., H. Gohlke, and G. Klebe (2005) "DrugScoreCSD knowledge-based scoring function derived from small molecule crystal data with superior recognition rate of near-native ligand poses and better affinity prediction," *J. Med. Chem.*, **48**(20), pp. 6296–6303.

[75] Neudert, G. and G. Klebe (2011) "DSX: a knowledge-based scoring function for the assessment of protein–ligand complexes," *J. Chem. Inf. Model.*, **51**(10), pp. 2731–2745.

[76] Yang, C.-Y., R. Wang, and S. Wang (2006) "M-score: a knowledge-based potential scoring function accounting for protein atom mobility," *J. Med. Chem.*, **49**(20), pp. 5903–5911.

[77] Ganesan, K., J. San Miguel, and N. Enright Jerger (2019) "The What's Next Intermittent Computing Architecture," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 211–223.

[78] Bertsekas, D. P. and D. Castañon (1999) "Rollout algorithms for stochastic scheduling problems," in *Journal of Heuristics*.

[79] Yeh, T.-Y. and Y. N. Patt (1992) "Alternative implementations of two-level adaptive branch prediction," in *Proc. of International symposium on Computer architecture (ISCA)*, ACM, New York, NY, USA, pp. 124–134.

[80] Pan, S.-T., K. So, and J. T. Rahmeh (1992) "Improving the accuracy of dynamic branch prediction using branch correlation," in *Proc. of International conference on Architectural support for programming languages and operating systems (ASPLOS)*.

[81] Jiménez, D. A. and C. Lin (2002) "Dynamic Branch Prediction with Perceptrons," in *Proc. of International Symposium on High Performance Computer Architecture (HPCA)*.

[82] Chrysos, G. Z. and J. S. Emer (1998) "Memory dependence prediction using store sets," in *Proc. of International symposium on Computer architecture (ISCA)*.

[83] Moshovos, A. and G. S. Sohi (1999) "Read-After-Read Memory Dependence Prediction," in *Proc. of International Symposium on Microarchitecture (MICRO)*.

[84] Subramaniam, S. and G. H. Loh (2006) "Store Vectors for Scalable Memory Dependence Prediction and Scheduling," in *Proc. of International Symposium on High Performance Computer Architecture (HPCA)*.

[85] Uht, A. K., V. Sindagi, and K. Hall (1995) "Disjoint eager execution: an optimal form of speculative execution," in *Proc. of International symposium on Microarchitecture (MICRO)*.

[86] Klauser, A., A. Paithankar, and D. Grunwald (1998) "Selective eager execution on the PolyPath architecture," in *Proc. of International symposium on Computer architecture (ISCA)*.

[87] August, D. I., D. A. Connors, S. A. Mahlke, J. W. Sias, K. M. Crozier, B.-C. Cheng, P. R. Eaton, Q. B. Olaniran, and W.-m. W. Hwu (1998) "Integrated Predicated and Speculative Execution in the IMPACT EPIC Architecture," in *Proceedings of the 25th Annual International Symposium on Computer Architecture*.

[88] Lim, C. L., A. Moffat, and A. Wirth (2014) "Lazy and Eager Approaches for the Set Cover Problem," in *Proceedings of the Thirty-Seventh Australasian Computer Science Conference - Volume 147*.

[89] Williams, S., A. Waterman, and D. Patterson (2009) "Roofline: an insightful visual performance model for multicore architectures," *Communications of the ACM*, **52**(4), pp. 65–76.

[90] Hong, S. and H. Kim (2010) "An integrated GPU power and performance model," in *Proceedings of the 37th annual international symposium on Computer architecture*, pp. 280–289.

[91] Lv, M., W. Yi, N. Guan, and G. Yu (2010) "Combining abstract interpretation with model checking for timing analysis of multicore software," in *2010 31st IEEE Real-Time Systems Symposium*, IEEE, pp. 339–349.

[92] Stengel, H., J. Treibig, G. Hager, and G. Wellein (2015) "Quantifying performance bottlenecks of stencil computations using the execution-cache-memory model," in *ICS*, pp. 207–216.

[93] Hoefler, T., W. Gropp, W. Kramer, and M. Snir (2011) "Performance modeling for systematic performance tuning," in *SC*.

[94] Konstantinidis, E. and Y. Cotronis (2015) "A practical performance model for compute and memory bound GPU kernels," in *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, IEEE, pp. 651–658.

[95] Gast, N. and G. Bruno (2010) "A mean field model of work stealing in large-scale systems," *ACM SIGMETRICS Performance Evaluation Review*, **38**(1), pp. 13–24.

[96] Yoo, W., K. Larson, L. Baugh, S. Kim, and R. H. Campbell (2012) "Adp: Automated diagnosis of performance pathologies using hardware events," *Sigmetrics*, **40**(1), pp. 283–294.

[97] Yoo, W., A. Sim, and K. W (2016) "Machine Learning Based Job Status Prediction in Scientific Clusters," in *SAI Computing Conference*.

[98] Kaufman, S. J., P. M. Phothilimthana, Y. Zhou, C. Mendis, S. Roy, A. Sabne, and M. Burrows (2020) "A Learned Performance Model for Tensor Processing Units," *arXiv preprint arXiv:2008.01040*.

[99] Ardalani, N., C. Lestourgeon, K. Sankaralingam, and X. Zhu (2015) "Cross-architecture performance prediction (XAPP) using CPU code to predict GPU performance," in *MICRO*.

[100] Jaggard, A. D., S. Kopparty, V. Ramachandran, and R. N. Wright (2013) "The design space of probing algorithms for network-performance measurement," *Sigmetrics*, **41**(1), pp. 105–116.

[101] Wang, Y., G.-Y. Wei, and D. Brooks (2020) "A Systematic Methodology for Analysis of Deep Learning Hardware and Software Platforms," in *Proceedings of Machine Learning and Systems*, vol. 2, pp. 30–43.

[102] Dev, K., X. Zhan, and S. Reda (2016) "Power-aware characterization and mapping of workloads on CPU-GPU processors," in *IISWC*.

[103] Krizhevsky, A., I. Sutskever, and G. E. Hinton (2012) "ImageNet Classification with Deep Convolutional Neural Networks," in *Advances in Neural Information Processing Systems 25*, Curran Associates, Inc., pp. 1097–1105.
URL `http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolu`
`pdf`

[104] Jia, Y. et al. (2014) "Caffe: Convolutional Architecture for Fast Feature Embedding," *arXiv preprint arXiv:1408.5093*.

[105] Phrma (2015) "Biopharmaceutical research & development: The process behind new medicines," *PhRMA*.

[106] Morgan, S., P. Grootendorst, J. Lexchin, C. Cunningham, and D. Greyson (2011) "The cost of drug development: a systematic review," *Health Policy*, **100**(1), pp. 4–17.

[107] Dickson, M. and J. P. Gagnon (2009) "The cost of new drug discovery and development," *Discovery Medicine*, **4**(22), pp. 172–179.

[108] Mullin, R. (2003) "Drug development costs about $1.7 billion," *Chem. Eng. News*, **81**(50), pp. 8–8.

[109] Hammersley, J. (2013) *Monte carlo methods*, Springer Science & Business Media.

[110] Trott, O. and A. J. Olson (2010) "AutoDock Vina: improving the speed and accuracy of docking with a new scoring function, efficient optimization, and multithreading," *J. Comput. Chem.*, **31**(2), pp. 455–461.

[111] Fourches, D., E. Muratov, F. Ding, N. V. Dokholyan, and A. Tropsha (2013) "Predicting binding affinity of CSAR ligands using both structure-based and ligand-based approaches," *J. Chem. Inf. Model.*, **53**(8), pp. 1915–1922.

[112] Proctor, E. A., S. Yin, A. Tropsha, and N. V. Dokholyan (2012) "Discrete molecular dynamics distinguishes nativelike binding poses from decoys in difficult targets," *Biophys. J.*, **102**(1), pp. 144–151.

[113] Hsieh, J.-H., S. Yin, X. S. Wang, S. Liu, N. V. Dokholyan, and A. Tropsha (2012) "Cheminformatics meets molecular mechanics: a combined application of knowledge-based pose scoring and physical force field-based hit scoring functions improves the accuracy of structure-based virtual screening," *J. Chem. Inf. Model.*, **52**(1), pp. 16–28.

[114] Hsieh, J.-H., S. Yin, S. Liu, A. Sedykh, N. V. Dokholyan, and A. Tropsha (2011) "Combined application of cheminformatics-and physical force field-based scoring functions improves binding affinity prediction for CSAR data sets," *J. Chem. Inf. Model.*, **51**(9), pp. 2027–2035.

[115] Feinberg, G. and J. Sucher (1970) "General theory of the van der Waals interaction: A model-independent approach," *Phys. Rev. A*, **2**(6), p. 2395.

[116] Wang, C. and Y. Zhang (2017) "Improving scoring-docking-screening powers of protein–ligand scoring functions using random forest," *J. Comput. Chem.*, **38**(3), pp. 169–177.

[117] Nguyen, D. D. and G.-W. Wei (2019) "AGL-score: algebraic graph learning score for protein–ligand binding scoring, ranking, docking, and screening," *J. Chem. Inf. Model.*, **59**(7), pp. 3291–3304.

[118] Sánchez-Cruz, N., J. L. Medina-Franco, J. Mestres, and X. Barril (2021) "Extended connectivity interaction features: improving binding affinity prediction through chemical description," *Bioinformatics*, **37**(10), pp. 1376–1382.

[119] Morrone, J. A., J. K. Weber, T. Huynh, H. Luo, and W. D. Cornell (2020) "Combining docking pose rank and structure with deep learning improves protein–ligand binding mode prediction over a baseline docking approach," *J. Chem. Inf. Model.*, **60**(9), pp. 4170–4179.

[120] Torng, W. and R. B. Altman (2019) "Graph convolutional neural networks for predicting drug-target interactions," *Journal of Chemical Information and Modeling*, **59**(10), pp. 4131–4149.

[121] Liu, Z., M. Su, L. Han, J. Liu, Q. Yang, Y. Li, and R. Wang (2017) "Forging the basis for developing protein–ligand interaction scoring functions," *Acc. Chem. Res.*, **50**(2), pp. 302–309.

[122] Huang, Y., B. Niu, Y. Gao, L. Fu, and W. Li (2010) "CD-HIT Suite: a web server for clustering and comparing biological sequences," *Bioinformatics*, **26**(5), pp. 680–682.

[123] Niu, B., L. Fu, S. Sun, and W. Li (2010) "Artificial and natural duplicates in pyrosequencing reads of metagenomic data," *BMC bioinformatics*, **11**(1), pp. 1–11.

[124] Wang, J. and N. V. Dokholyan (2019) "MedusaDock 2.0: Efficient and Accurate Protein–Ligand Docking With Constraints," *Journal of Chemical Information and Modeling*, **59**(6), pp. 2509–2515.

[125] Shi, Y., Z. Huang, W. Wang, H. Zhong, S. Feng, and Y. Sun (2020) "Masked Label Prediction: Unified Message Passing Model for Semi-Supervised Classification," *arXiv preprint arXiv:2009.03509*.

[126] Shi, W. and R. Rajkumar (2020) "Point-gnn: Graph neural network for 3d object detection in a point cloud," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 1711–1719.

[127] Lin, K., L. Wang, and Z. Liu (2021) "End-to-end human pose and mesh reconstruction with transformers," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 1954–1963.

[128] Willmott, C. J. and K. Matsuura (2005) "Advantages of the mean absolute error (MAE) over the root mean square error (RMSE) in assessing average model performance," *Climate research*, **30**(1), pp. 79–82.

[129] Li, Y., L. Han, Z. Liu, and R. Wang (2014) "Comparative assessment of scoring functions on an updated benchmark: 2. Evaluation methods and general results," *J. Chem. Inf. Model.*, **54**(6), pp. 1717–1736.

[130] Li, Y., M. Su, Z. Liu, J. Li, J. Liu, L. Han, and R. Wang (2018) "Assessing protein–ligand interaction scoring functions with the CASF-2013 benchmark," *Nat. Protoc.*, **13**(4), pp. 666–680.

[131] Su, M., Q. Yang, Y. Du, G. Feng, Z. Liu, Y. Li, and R. Wang (2018) "Comparative assessment of scoring functions: the CASF-2016 update," *J. Chem. Inf. Model.*, **59**(2), pp. 895–913.

[132] Esmaeilzadeh, H., A. Sampson, L. Ceze, and D. Burger (2012) "Neural acceleration for general-purpose approximate programs," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, IEEE Computer Society, pp. 449–460.

[133] Yazdanbakhsh, A., D. Mahajan, H. Esmaeilzadeh, and P. Lotfi-Kamran (2016) "AxBench: A multiplatform benchmark suite for approximate computing," *IEEE Design & Test*, **34**(2), pp. 60–68.

[134] Shimbel, A. (1954) "Structure in communication nets," in *Proceedings of the symposium on information networks*, Polytechnic Institute of Brooklyn, pp. 119–203.

[135] Tang, X., A. Pattnaik, H. Jiang, O. Kayiran, A. Jog, S. Pai, M. Ibrahim, M. T. Kandemir, and C. R. Das (2017) "Controlled Kernel Launch for Dynamic Parallelism in GPUs," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 649–660.

[136] LeCun, Y., L. Bottou, Y. Bengio, and P. Haffner (1998) "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, **86**(11), pp. 2278–2324.

[137] Simonyan, K. and A. Zisserman (2014) "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*.

[138] Ding, F., S. Yin, and N. V. Dokholyan (2010) "Rapid flexible docking using a stochastic rotamer library of ligands," *Journal of chemical information and modeling*, **50**(9), pp. 1623–1632.

[139] Yazdanbakhsh, A., D. Mahajan, H. Esmaeilzadeh, and P. Lotfi-Kamran (2017) "AxBench: A multiplatform benchmark suite for approximate computing," *IEEE Design & Test*, **34**(2), pp. 60–68.

[140] Ulman, V., M. Maska, K. E. G. Magnusson, O. Ronneberger, C. Haubold, N. Harder, P. Matula, P. Matula, D. Svoboda, M. Radojevic, I. Smal, K. Rohr, J. Jaldén, H. M. Blau, O. Dzyubachyk, B. Lelieveldt, P. Xiao, Y. Li, S.-Y. Cho, A. C. Dufour, J.-C. Olivo-Marin, C. C. Reyes-Aldasoro, J. A. Solis-Lemus, R. Bensch, T. Brox, J. Stegmaier, R. Mikut, S. Wolf, F. A. Hamprecht, T. Esteves, P. Quelhas, Ö. Demirel, L. Malmström, F. Jug, P. Tomancak, E. Meijering, A. Muñoz-Barrutia, M. Kozubek, and C. Ortiz-de Solorzano (2017) "An objective comparison of cell-tracking algorithms," *Nature Methods*, pp. EP –.
URL http://dx.doi.org/10.1038/nmeth.4473

[141] Maška, M., V. Ulman, D. Svoboda, P. Matula, P. Matula, C. Ederra, A. Urbiola, T. España, S. Venkatesan, D. M. Balak, P. Karas, T. Bolcková, M. Štreitová, C. Carthel, S. Coraluppi, N. Harder, K. Rohr, K. E. G. Magnusson, J. Jaldén, H. M. Blau, O. Dzyubachyk, P. Křížek, G. M. Hagen, D. Pastor-Escuredo, D. Jimenez-Carretero, M. J. Ledesma-Carbayo, A. Muñoz-Barrutia, E. Meijering, M. Kozubek, and C. Ortiz-de Solorzano (2014) "A benchmark for comparison of cell tracking algorithms," *Bioinformatics*, **30**(11), pp. 1609–1617.
URL +http://dx.doi.org/10.1093/bioinformatics/btu080

[142] Wang, R., X. Fang, Y. Lu, C.-Y. Yang, and S. Wang (2005) "The PDBbind database: methodologies and updates," *Journal of medicinal chemistry*, **48**(12), pp. 4111–4119.

[143] Padmavathi, G., P. Subashini, and P. Lavanya (2009) "Performance evaluation of the various edge detectors and filters for the noisy IR images," in *Proceedings of the 2Nd WSEAS International Conference on Sensors, and Signals and Visualization, Imaging and Simulation and Materials Science*.

[144] Fanding, D. (1994) "A Faster Algorithm for Shortest-Path - SPFA," *Journal of Southwest Jiaotong University*, **2**.

[145] Kislal, O., P. Berman, and M. Kandemir (2012) "Improving the performance of k-means clustering through computation skipping and data locality optimizations," in *Proceedings of the 9th conference on Computing Frontiers*, ACM, pp. 273–276.

[146] Jiang, H., H. Zhang, X. Tang, V. Govindaraj, J. Sampson, M. T. Kandemir, and D. Zhang (2021) "Fluid: a framework for approximate concurrency via controlled dependency relaxation," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pp. 252–267.

Vita

Huaipan Jiang

Huaipan Jiang was born in Jiangsu, China in 1993. He received his Bachelor's degree in Computer Science and Technology in 2015, from University of Science and Technology of China. He entered the PhD program in Computer Science and Engineering at Penn State since January 2016, as a member of both High Performance Computing Laboratory (HPCL) and Microsystems Design Lab (MDL), where he closely worked with Prof. Mahmut T. Kandemir. His dissertation is broadly focused on maximizing performance and minimizing accuracy loss for existing applications with the help of the machine learning technique. Those applications widely from but not limited to image processing, biomedical and drug discovery. His research has been published in top-tier conferences and journals such as PLDI, MICRO, DATE, HPCA, and JCIM. During his PhD, he also worked as an intern at Google in 2018, 2019, 2021 and Facebook in 2020.