

UNITED STATES PATENT AND TRADEMARK OFFICE

BEFORE THE PATENT TRIAL AND APPEAL BOARD

HUGGING FACE, INC.,

Petitioner,

v.

FRIENDLIAI INC.,

Patent Owner.

IPR 2024-01234

U.S. Patent No. 11,442,775

DECLARATION OF GHAITH HAMMOURI, Ph.D.

TABLE OF CONTENTS

	Page No.
I. INTRODUCTION	3
A. Qualifications	3
II. MATERIALS REVIEWED	6
III. SUMMARY OF MY OPINIONS	8
IV. LEGAL PRINCIPLES.....	8
A. Understanding of Patent Law.....	8
B. Claim Construction.....	10
C. Level of Ordinary Skill in the Art	12
V. THE ‘775 PATENT.....	13
A. Priority of ‘775 Patent.....	13
B. Summary of the ‘775 Patent.....	14
VI. OVERVIEW OF THE PRIOR ART	18
A. Summary of Gao	26
B. Summary of Katharopoulos	32
VII. REASONS THE CHALLENGED CLAIMS OF THE ‘775 PATENT ARE UNPATENTABLE.....	34
A. GROUND 1: Claims 1-18 are rendered obvious under 35 U.S.C. § 103 by Gao in view of Katharopoulos.....	34
VIII. OATH	110

I, Dr. Ghaith Hammouri, declare as follows:

I. INTRODUCTION

1. Hugging Face, Inc., (collectively “Hugging Face”) has retained my services in connection with the above captioned *Inter Partes* Review (IPR) of U.S. Patent No. 11,442,775 (‘775 Patent). I have been asked to study and provide my opinions as an independent expert witness regarding technology described in the ‘775 Patent. I am being compensated at my usual and customary rate for my time. Such compensation, however, does not influence my opinion nor does the outcome of this proceeding impact my compensation.

A. Qualifications

2. My qualifications and professional experience are described in my curriculum vitae. I have been informed that a copy of my curriculum vitae will be submitted with my declaration. The following is a summary of my relevant qualifications and professional experience.

3. I received my PhD in Electrical and Computer Engineer from Worcester Polytechnique Institute (WPI) in 2009. Prior to that I received my M.S. in Electrical Engineering from the University of Hartford in 2004. I also received my B.S. in Electrical Engineering with a second major in Physics and a minor in Math from the University of Hartford in 2003.

4. My PhD work and thesis were focused on cryptography and the use of learning models for extracting unique and secure identifiers from physical hardware.

After my PhD, I worked as a post-Doctoral researcher at WPI where I continued pursuing my research. I invented a technique for fingerprinting CDs and other forms of optical media using statistical learning. During this time, I mentored junior graduate students who joined the (CRIS) lab and acted as their intern advisor where I coordinated with the head of the lab. My post-Doc and research were both funded through a National Science Foundation (NSF) academic grant.

5. In 2010, I co-founded Intryca (renamed Claveo) in order to commercialize my research. I served as the VP of Technology where I lead the research and development efforts of the company. My research focused on developing statistical learning methods for extracting hardware fingerprints from smart phones. Based on my research, I was awarded a Small Business Innovation Research (SBIR) grant from the (NSF). Further, this research and corresponding development resulted in 3 US patents.

6. In 2012, I became the CEO (and later a co-owner) of Simtix, a physical security company with customers across the Middle East. The company's products mainly revolved around automating physical security using RFID and Automatic Number Plate Recondition (ANPR) technology. At my role, I spent a significant portion of my time overseeing the technology while supervising the technical team and introducing them to new technologies.

7. In 2016, I left my full-time role at Simtix to completely focus my attention on Machine Learning. I co-founded Xr.AI, a New York based startup focused on utilizing Machine Learning and Natural Language Processing (NLP) in order to automate the analysis of legal contracts. I served as the Chief Scientific Officer of the company where my work focused on creating a new technology for clause-level vector embeddings using Recurrent Neural Networks (RNNs). For this work, I was awarded a Small Business Innovation Research (SBIR) grant from the National Science Foundation (NSF).

8. In 2020, I left corporate work to spend more time on my long-term research project of exploring deep connections between learning theory and cryptography. Over the past four years, I have served in a consulting capacity as the CEO of Simtix where I lead the development of Secure-Brain technology, a vision AI engine built on Multi-Modal Transformer technology to be used in security and defense applications. In this capacity, I mentored the development team and introduced them to modern machine learning concepts and tools.

9. I currently also serve as a consultant for several private companies advising them on the use and applications of AI technology including Large Language Models (LLMs) and Multi-Modal Transformer models.

10. Since 2023, I have also served as an affiliate research scientist at WPI pursuing research in areas relating to machine learning and security. Specifically, I

have been actively researching adversarial vulnerabilities in LLMs which are in general based on the machine learning Transformer model.

II. MATERIALS REVIEWED

11. In forming my opinions provided in this declaration, I have reviewed and considered at least the following. I have also considered the other background references I have cited in this declaration as well as my education, knowledge, and experience working in machine learning, software development, and consumer products.

U.S. Patent No. 11,442,775 (“the ’775 patent”)
Declaration of Ghaith Hammouri
C.V. of Ghaith Hammouri
Pin Gao, <i>et. al.</i> , <i>Low Latency RNN Inference with Cellular Batching</i> , Thirteenth EuroSys Conference 2018, April 23–26, 2018, Porto, Portugal, published by the Association for Computing Machinery (2018) (“Gao”)
Katharopoulos, <i>et. al.</i> , <i>Transformers are RNNs: Fast Autoregressive Transformers with Linear Attention</i> by Katharopoulos <i>et al.</i> , arXiv : 2006.16236v3 , Aug. 31 , 2020, presented at Proceedings of the 37th International Conference on Machine Learning, Online, PMLR 119, 2020 (“Katharopoulos”).
Original Complaint, FriendliAI Inc. v. Hugging Face, Inc., Case No. 23-816-MN, D. Del. (filed July 28, 2023)
First Amended Complaint, FriendliAI Inc. v. Hugging Face, Inc., Case No. 23-816-MN, D. Del. (filed January 8, 2024)
Gyeong-In Yu, <i>et. al.</i> , <i>Orca: A Distributed Serving System for Transformer-Based Generative Models</i> , Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation, July 11–13, 2022, Carlsbad, CA, USA (“Orca paper”)

A. Vaswani, <i>et al.</i> , <i>Attention is All you Need</i> , Advances in Neural Information Processing Systems, 2017 (“Vaswani”)
Prosecution history of application 17/542,193.
Plaintiff’s Disclosure of Proposed Claim Constructions, <i>FriendliAI Inc. v. Hugging Face, Inc.</i> , Case No. 23-816-MN, D. Del. (served on May 28, 2024)
Defendant’s Disclosure of Proposed Claim Construction, <i>FriendliAI Inc. v. Hugging Face, Inc.</i> , Case No. 23-816-MN, D. Del. (served on May 28, 2024)
Plaintiff’s Final Infringement Contentions of ’775 Patent, <i>FriendliAI Inc. v. Hugging Face, Inc.</i> , Case No. 23-816-MN, D. Del. (served on July 1, 2024)
Ilya Sutskever, <i>et al.</i> , <i>Sequence to Sequence Learning with Neural Networks</i> , arXiv : 1409.3215v3, Dec. 14, 2014, Proceedings of the 28th Conference on Neural Information Processing Systems 2014, December 8–13, 2014, Palais des congrès de Montréal, Canada, (“Sutskever”)
Dzmitry Bahdanau, <i>et al.</i> , <i>Neural Machine Translation by Jointly Learning to Align and Translate</i> , arXiv : 1409.0473v7, May 19, 2016, Proceedings of the International Conference on Learning Representations 2015, May 7–9, 2015, San Diego, California (“Bahdanau”)
Romain Paulus, <i>et al.</i> , <i>A Deep Reinforced Model for Abstractive Summarization</i> , arXiv : 1705.04304v3, November 13, 2017, Published online by Salesforce Research, Palo Alto, California (“Paulus”)
Peter J. Liu, <i>et al.</i> , <i>Generating Wikipedia by Summarizing Long Sequences</i> , arXiv : 1801.10198v1, January 30, 2018, Proceedings of the International Conference on Learning Representations 2018, April 30–May 3, 2018, Vancouver, Canada (“Liu”)
Alec Radford, <i>et al.</i> , <i>Improving Language Understanding by Generative Pre-Training</i> , https://api.semanticscholar.org/CorpusID:49313245 , June 11, 2018, Published online (“Radford”)
Yao-Hung Hubert Tsai, <i>et al.</i> , <i>Transformer Dissection: A Unified Understanding of Transformer’s Attention via the Lens of Kernel</i> , arXiv : 1908.11775v4, November 11, 2019, Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP) (pp. 4335–4344), November 3–7, 2019, Hong Kong, China (“Tsai”)
Ankit Singh Rawat, <i>et al.</i> , <i>Sampled Softmax with Random Fourier Features</i> , arXiv : 1907.10747v2, December 31, 2019, Proceedings of the Conference on Neural Information Processing Systems 2019, December 8–14, 2019, Vancouver, Canada (“Rawat”)

Guy Blanc, *et al.*, *Adaptive Sampled Softmax with Kernel Based Sampling*, arXiv : 1712.00527v2, August 1, 2018, Proceedings of the 35th International Conference on Machine Learning, PMLR Vol. 80 pp. 590–599, July 10–15, 2018, Stockholm, Sweden (“Blanc”)

Claim appendix for claims 1-18 of the '775 Patent

III. SUMMARY OF MY OPINIONS

12. I understand the following table lists the ground of rejection I have considered in this declaration:

Prior Art	Basis	Claims
Gao in view of Katharopoulos	Obviousness	1-18

13. After a review of the '775 Patent and the prior art, it is my opinion that the Challenged Claims are invalid under the proposed ground. My opinions, and the bases therefore, are detailed throughout this Declaration.

IV. LEGAL PRINCIPLES

A. Understanding of Patent Law

14. I am not an attorney and will not be offering legal conclusions. However, I have been informed of several principles concerning legal issues relevant to my analysis of the Challenges to the claims of the '775 Patent, and I relied on these principles to arrive at my conclusions.

15. I understand a claim is anticipated under 35 U.S.C. § 102 if all limitations are found in a single prior art reference, arranged as in the claim. The

identical invention must be shown in complete detail as is contained in the patent claim.

16. I understand a prior art reference can disclose an element not expressly identified in a reference if the element is “inherently present” in the reference. To be “inherent,” I understand the missing element must necessarily be present in the reference. An element is not “inherent” if the missing element is only probably present or if there is merely a possibility it is present.

17. I understand a claim is invalid as obvious under 35 U.S.C. § 103 if the differences between the subject matter sought to be patented and the prior art are such that the subject matter of the claim as a whole would have been obvious at the time of the patent’s filing date to a Person of Ordinary Skill in The Art (POSITA). It is my understanding that the following factors are used to determine whether or not the claimed subject matter would have been obvious: (i) the scope and content of the prior art; (ii) the differences between the prior art and the claimed invention; (iii) the level of ordinary skill in the field of the invention; and (iv) any relevant objective considerations of non-obviousness.

18. I understand a party asserting obviousness based on a combination of prior art references must demonstrate that one of ordinary skill in the art would have been motivated to combine the teachings of those references to achieve the claimed invention with a reasonable expectation of success. It is my understanding that it is

not sufficient to show that one of ordinary skill in the art *could* combine elements of multiple references. Instead, there must be a rational reason that would have prompted a POSITA to combine the elements in the way the claimed invention does; and the reason should be explained or articulated.

19. I understand a combination of references would not have been obvious if the alleged modification(s) to be made to the reference(s) are inconsistent with the stated goals of the reference(s). I understand a combination of references would not have been obvious if the modification of the reference(s) to derive what is claimed would render the reference(s) unsatisfactory or inoperable for their intended purpose. I further understand the party asserting obviousness must explain why a POSITA would have selected components for combination in the manner claimed.

20. It is my further understanding that an invention would not have necessarily been obvious simply because all the elements of the invention may have been known separately in the prior art; there must be a reason to combine the separately known elements. Obviousness cannot be based on a hindsight combination of components selectively picked from the art using the claims as guide.

B. Claim Construction

21. I understand that claim construction in an IPR proceeding is a legal question for the Patent Trial and Appeal Board (PTAB or Board) to decide. In

general, I understand that claim terms are to be given their ordinary and customary meaning to a POSITA in the context of the patent at the time the patent was filed.

22. I also understand that in construing claim terms, the Board asks what the claim terms would mean to a person of ordinary skill in the relevant art in view of the plain claim language and the disclosures of the patent and prosecution history. I understand that the Board may also consider other external evidence, such as dictionaries, however the disclosures in the patent and prosecution history carry more weight than external evidence.

23. As such, any claim term not construed should be given its ordinary and customary meaning as would be understood by one of ordinary skill in the art.

24. I understand that the best source for determining the meaning of a claim is intrinsic evidence—the claims themselves, the written description, and the prosecution history. I also understand that extrinsic evidence, which consists of all evidence external to the patent and prosecution history, may be considered to determine the meaning of a claim term.

25. In view of the principles described above and the materials I have reviewed, I do not believe any limitations in the claims addressed herein require a specific construction to support the opinions I provide in this declaration.

C. Level of Ordinary Skill in the Art

26. I understand that certain issues in an IPR, such as claim construction and whether a claim is invalid as obvious, are assessed from the view of a hypothetical person of ordinary skill in the relevant art at the time of the invention. I understand there are multiple factors relevant to determining the level of ordinary skill in the art, including: (1) the level of education and experience of persons working in the field at the time of the invention; (2) the sophistication of the technology; (3) the types of problems encountered in the field; and (4) the prior art solutions to those problems.

27. In order to determine the characteristics of a hypothetical POSITA at the time of the claimed invention, I have considered a variety of factors. I have considered the prior art (referred to in the “Materials Considered” section of this declaration) and the various approaches to address the batching of machine-learning tasks disclosed in those prior art documents, the types of problems encountered in the art and the solutions to those problems, the alleged problems encountered by the inventor as described in the ‘775 patent, the sophistication of the technology involved, and the educational background and experience of those actively working in the relevant field at the time of the invention.

28. Additionally, I considered the technology available in 2021, immediately before the filing of the patent application in December of 2021, and the

professionals with whom I worked during that time, including their levels of education, sophistication, and activities in professional associations. I am informed that such considerations are in accordance with factors identified in case law and typically considered to determine the level of skill in the art.

29. The field of “art” for the ‘775 patent is machine-learning.

30. In view of the above and based on my experience and knowledge, I believe a hypothetical Person of Ordinary Skill in The Art (POSITA) with regard to the ‘775 patent would have either: (1) a bachelor’s degree in electrical engineering, computer engineering, or computer science, with two to three years of work experience in machine learning; or (2) a master’s degree in electrical engineering, computer engineering, or computer science, with one year of work experience in machine learning.

31. Although I describe the POSITA as of December 2021, it is my further opinion that the fundamental qualifications, attributes, and skills of the person of ordinary skill in the art would have been the same for many years prior to December of 2021 and presently remain the same.

V. THE ‘775 PATENT

A. Priority of ‘775 Patent

32. The ‘775 patent was filed on December 3, 2021. *See* ‘775 Patent.

B. Summary of the ‘775 Patent

33. The ‘775 Patent relates to an inference system that applies a machine-learning transformer model to batches of input requests with variable input lengths. ‘775 Patent at Abstract. This dynamic batching allows the utilization of hardware accelerators’ parallel computation capabilities while avoiding unnecessary computations from forcing requests into uniform lengths. ‘775 Patent at Abstract.

34. The ‘775 Patent, in Figures 5A-5D, illustrates the claimed method for dynamic batching and processing of requests using a machine-learning transformer model. ‘775 Patent at Figures 5A-5D; 22:22-24:38.

35. As shown in Figures 5A-5D, the system includes a serving system 435, request processor 580, and scheduler 585 coupled to multiple execution engines 590A and 590B. ‘775 Patent at 22:22-57.

36. Figure 5A illustrates that a single request R1 (with a single input token) is scheduled to execute in execution engine 590A while execution engine 590B is scheduled to execute a batch of requests, R3 (having two input tokens), R4 (having three input tokens) and R5 (having two input tokens). ‘775 Patent at 22:47-51.

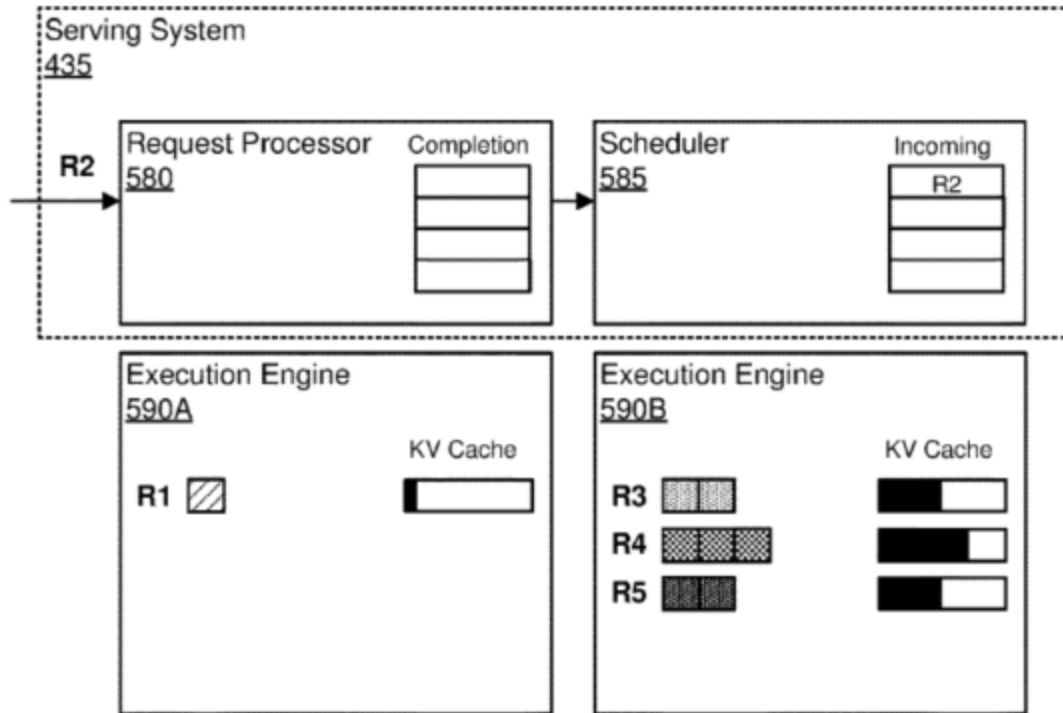


FIG. 5A

37. As new request R2 arrives at request processor 590, it is forwarded to scheduler 585 which monitors the cache memory for execution engines 590A and 590B to determine if memory is available for processing request R2. ‘775 Patent at 22:58-23:29.

38. Moving to Figure 5B, as first output token is generated for requests R1, R3, R4, and R5, execution engine 590A is now scheduled to execute updated batch R1 and R2 at a second iteration. ‘775 Patent at 23:30-50. Execution engine 590A is capable of performing both the encoding phase and decoding phase for the same batch of requests in conjunction with using a machine-learning transformer model

300. '775 Patent at 23:39-44; *see also* '775 Patent at 11:26-17:32 (describing processing methodology for a machine-learning transformer model).

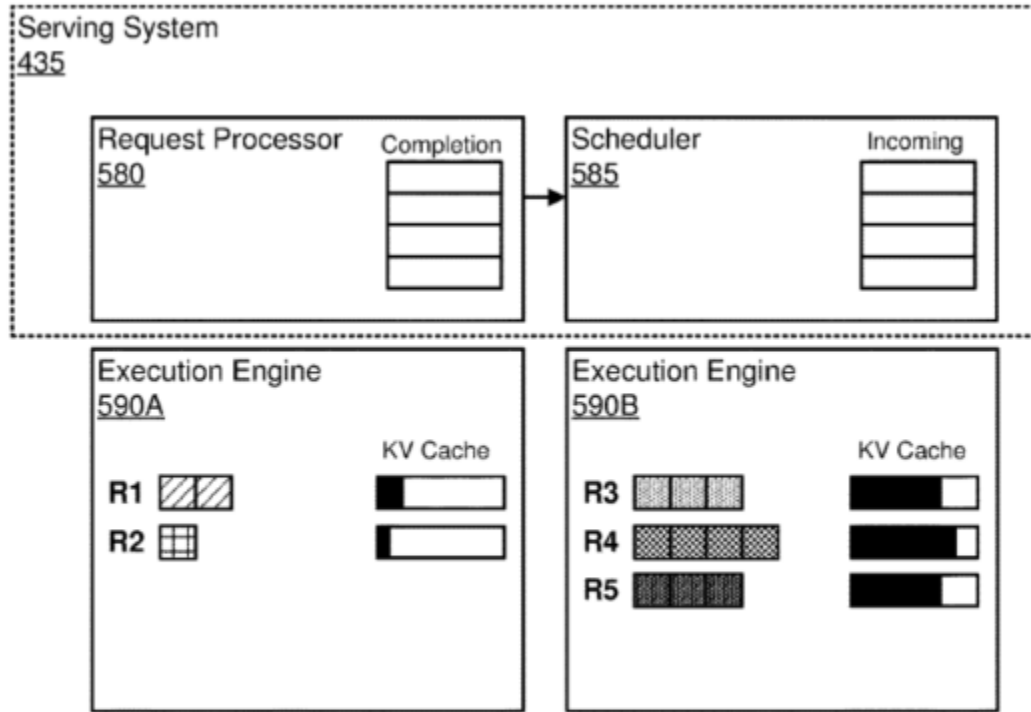


FIG. 5B

39. In Figure 5C, a second output token is generated for request R1, R3, R4, and R5 and a first output token generated for request R2 with an end token that moves the outputs for request R2 to the completion queue of the request processor 580. '775 Patent at 23:51-58. The execution engine 590A then frees the cache memory allocated to request R2. '775 Patent at 23:59-60. Similarly, the second output token for request R4 is sent to the completion queue with its end token and execution engine 590B frees its cache memory allocated to request R4. '775 Patent at 23:60-67.

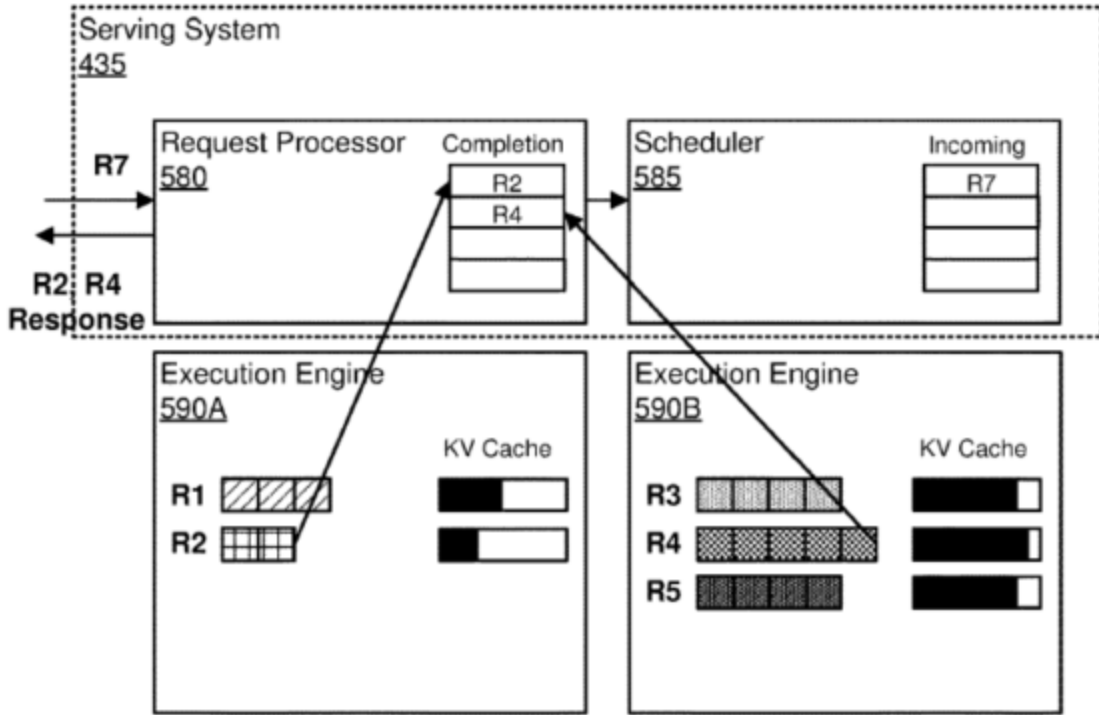


FIG. 5C

40. As another new request, R7 arrives, it is forwarded to the scheduler 585 where it is stored in the incoming queue. ‘775 Patent at 24:1-4. Since requests R2 and R4 are complete and cache is available in execution engine 590A, the scheduler 585 updates the batch for execution engine 590A to R1, R7 and execution engine 590B to R3, R5. ‘775 Patent at 24:4-11. Accordingly, by having dynamic batches for each iteration, “completed requests can be provided to the client device 110 as soon as processing is complete, and the scheduler 585 can schedule new requests. ‘775 Patent at 24:10-16.

41. Moving to Figure 5D, a third output token is generated for R1, R3, and R5 and execution 590A is ready to execute an updated batch of requests R1, R7

(with two tokens), and execution engine 590B is ready to execute an updated batch of requests R3 and R5. ‘775 Patent at 24:13-27.

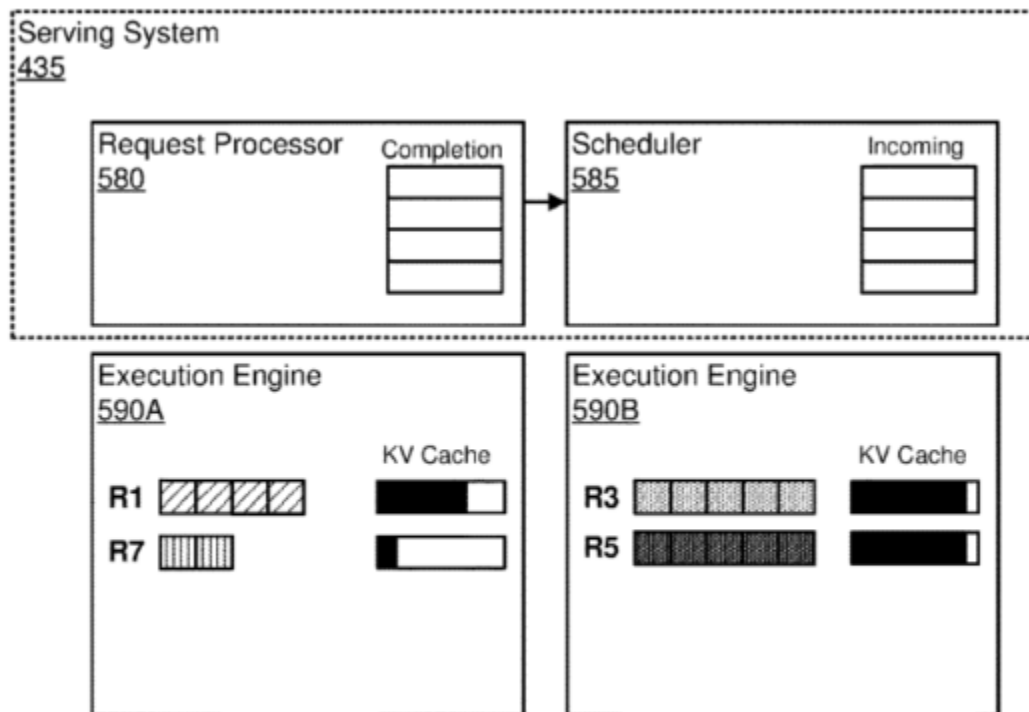


FIG. 5D

VI. OVERVIEW OF THE PRIOR ART

42. Before providing a detailed analysis of how the prior art discloses or teaches the limitations of the challenged claims, I provide a brief summary of state of the art and the individual prior art references.

Machine Learning and RNNs

43. Machine Learning in general is a field of study focused on using statistical techniques and algorithms in order to generalize data observations and predict the behavior over unseen data. One of the oldest techniques employed in

machine learning is that of a Neural Network (NN) model which was developed and introduced in the 1950's and 1960's. Early NNs were built using a single building cell, namely the perceptron. The perceptron is a simple circuit taking in a number of binary inputs each corresponding to some weight (a real number) which modulates the input, and finally produces a single decision output of True or False (1 or 0) based on some activation value (also a real number). The idea behind the perceptron was to capture the functionality that takes place in a human neuron.

44. In order to achieve higher levels of generalization (perceived intelligence), a NN utilizes many preceptors arranged in the form of a layer where multiple inputs are mapped into multiple outputs. Moreover, while a simple NN is made of a single layer, a more complex NN can be made of many interconnected layers (sometimes referred to as Multi-Layer Perceptron (MLP)). Typically, each layer is made of a number of preceptors whose output is connected to the next layer and so forth. The first layer is connected to the input while the last layer produces the output of the circuit leading to hidden layers in the middle. Once the network is created, the goal becomes to figure out the correct weights used in each perceptron in order to generate the correct output. This is the process of learning.

45. Learning in general is carried out over a set of desired inputs and outputs (training samples). The goal is to feed the input representation into the NN model and to find a way to modify the weights of the NN until the correct output is

produced. This process is carried out for all the training samples over many iterations until the NN model is finally capable of correctly producing the correct output for the training input.

46. The types of algorithms used to train a NN is outside of our scope here. However, it suffices to say that these training algorithms depend on the architecture and components of the underlying NN and typically require considerable computational resources to complete. A successful NN architecture would lend itself to training in a way that obtains higher levels of accuracy when tested to predict the output of the training samples. More importantly, the NN is expected to obtain high levels of accuracy when exposed to new data that the NN never saw during the training phase.

47. Over the years, many different types of NNs were introduced. Most known today are that of Recurrent Neural Networks (RNNs) and Convolutional Neural Networks (CNNs) both of which are considered to be Deep Neural Networks (DNN). Here we are interested in RNNs as they relate to the subject at hand.

48. RNNs were first introduced in the 1980s and only became practical to use during the late 1990 with the introduction of Long Short-Term Memory (LSTM). Since then, RNNs have evolved to encompass many types and variations of their underlying cell and neural networks. In general, RNNs were designed to process sequential data. That is, data that changes over time. Whereas NNs expect an input

of fixed size in order to produce an output, an RNN is designed to be able to process data in a sequence of varying length such as text or speech.

49. In its simplest form, an RNN can be viewed as a functional cell that takes in an external input along with a value of an internal hidden state, and correspondingly updates the values of the internally stored hidden state. This RNN function depends on a number of parameters that are learned during training. When processing an input sequence, say a list of words, the hidden state is initialized to some value before the RNN moves to process the first word in the sequence as input. Correspondingly, the RNN updates the value of the internal hidden state in a way that depends on the first word processed. Once the RNN moves to the next word in the list, it repeats the same computation only this time the hidden state used in the function has been updated in a way that depends on the first word. Continuing in this fashion, the RNN can process any number of input words until there are no more words to process. At such point, the internal state of the RNN holds a value that has been continuously updated corresponding to every word in the input. As a result, the internal state value can now be thought of as a representation of the input words and as such can be used to carry out a classification or prediction task relating to the processed input.

50. Due to the success of RNNs in many applications (e.g. Natural Language Processing (NLP), speech recognition, image classification) a large

number of RNN variations can be found in the literature. These various types of RNNs differ in their functionality, components, number of inputs or outputs, and so on. Examples of used RNN models include, LSTM, Gated Recurrent Units (GRU), Sequence to Sequence (Seq2Seq), and further include many other types of architectures and building blocks. Regardless of this variation, all RNNs share the same underlying property of retaining a hidden state that is updated according to the changing input in order to affect the final output of the circuit.

Transformers

51. In their groundbreaking work, Vaswani introduced Transformers, a new Deep Neural Network (DNN) model that completely relies on the attention mechanism to draw global dependencies between input and output. Simply put, the attention mechanism takes in an input encoded as a vector and maps it to an output that is also a vector.

52. By first mapping a sequence of encoded inputs to a corresponding set of Queries, Keys, and Values, (QKV) the output of the attention mechanism is formed as a weighted sum of input Values, where the weight assigned to each Value is computed as a compatibility function between a Query and a Key.

53. Although the Attention mechanism already existed in the literature (e.g. Bahdanau et al.), the main contribution of Vaswani was to rely only on the attention

mechanism to extract any interdependencies between the elements making up the input sequence.

54. Accordingly, one can easily see that all the building blocks making up a transformer model (except for the attention block) are element-wise operations that do not observe interdependency between input elements. Typically, transformers operate in an auto-regressive fashion where the predicted next element in a sequence of input elements is concatenated to the input and fed again to the system until a final special output element (e.g. <eos>) is generated. This mode of operation is very similar to that used in Seq2Seq RNN-models. (Sutskever et al.)

55. The original Transformer proposed by Vaswani was made up of two main component-stacks, an encoder-stack followed by a decoder-stack. While both components have slightly different building blocks and connections, their attention mechanism differs in one main way.

56. In an encoder, attention is computed between all input elements regardless of their position within the input sequence. On the other hand, the decoder uses masked attention (or causal attention), which only allows the attention to be computed between an input element and previous elements up to itself within the same input sequence. This difference ensures that the decoder, mainly tasked with predicting the next element in the input sequence, does not break the chain of

causality. That is, an element can only be influenced by itself and previous elements but not future elements.

57. Many different Transformer architectures have been proposed, but perhaps one of the most popular was the Decoder-Only-Transformer (Liu et al.), which is most known to have been used in GPT (Generative-Pretrained Transformer). *See* Radford et al.

58. In the DOT architecture, the Transformer is made of a number of decoders stacked in multiple consecutive layers, with their final output used to predict the next element of an input sequence. This architecture was preferred for having a simpler architecture along with efficient-implementation features, such as the masked attention, which allows the reuse of the Key and Value elements from previous iterations over the inputted and generated elements without requiring dependency on future element keys and values.

Batching in Machine Learning

59. Batching (sometimes called batch processing) is a known computer method which refers to the idea of combining multiple inputs to be processed in parallel as a batch. In the context of ML, batching has been widely used for many years in order to expedite the time required to process inputs. In the training phase of ML, batching is intimately related to the training algorithms used (e.g. minibatch-based Stochastic Gradient Descent (SGD)). Without going into the details, in this

phase the training dataset is typically broken into smaller batches in order reduce the amount of memory used (relative to what is required for the entire dataset) while expediting the processing time (relative to what is required for the entire dataset) by computing the training function in parallel.

60. In the inference phase of ML, the model is treated as a function that takes in an input and produces an output in real-time using the underlying trained network. To expedite this process, batching is used to process multiple inputs at the same time, thus maximizing utility of the computational resources such as memory and processors and increasing the output throughput.

61. Batching is particularly useful in the context of Graphics Processing Units (GPUs) since they are optimized to process high dimensional arrays. GPUs have multiple computing cores which gives them the ability to process large amounts of data in parallel especially high dimensional data (e.g. matrices and vectors). This is one of the reasons why most ML models greatly benefit from the usage of GPUs compared to a CPUs. By batching inference inputs when using a GPU, the amount of computation is increased by saturating the computing cores while masking the overhead required for off-chip memory access.

62. To this end, many ML batching techniques for inference have been introduced and used over the years. Different batching techniques typically take advantage of the ML model underlying architecture and the processing platform

used (e.g. CPU, GPU, ...). In fact, most ML software libraries and frameworks will incorporate some batching technique in order to expedite the ML inference when processing multiple inputs.

Final Remarks

63. A POSITA would have been well aware of these concepts as of December 3, 2021 and they would have formed part of a POSITA's background knowledge.

A. Summary of Gao

64. *Low Latency RNN Inference with Cellular Batching* by Gao et al., published in 2018 proposes a technique called “cellular batching” to improve the latency and throughput of Recurrent Neural Network (RNN) inference.

65. Deep learning consists of two main phases. A training phase, where “parameter weights are computed based on a training set;” and an inference phase wherein “the pre-trained model is used to process live application request using the computed weights.” Gao at p. 1. To provide a user with a good experience, inference requires low latency and good throughput – often requiring near real-time responses to request. Gao at p 1.

66. An RNN is a Deep Neural Network (DNN) architecture designed to process sequential data of variable length – often used for language processing applications. Gao at p. 2. RNNs help solve tasks such as predicting which words

would follow in a sentence. Gao at p. 2. For example, Gao provides figure 1 (reproduced below), which shows an unfolded dataflow graph for an input where the RNN is used for predicting the word most likely to follow the following input text: “system research is.” Gao at p. 2; Gao at Fig. 1. “At each time step, one input position is consumed and the calculated value of the hidden unit is then passed to the successor cell in the next time step.” Gao at p. 2. Once all three steps are

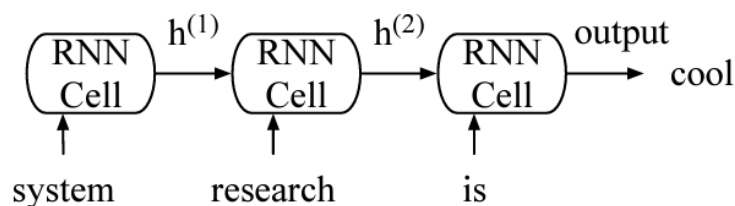


Figure 1: An unfolded chain-structured RNN. All RNN Cells in the chain share the same parameter weights.

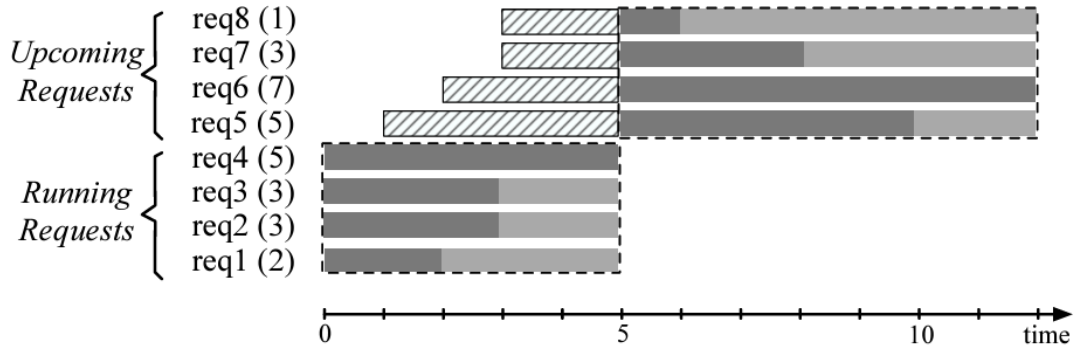
completed, the full context is known and the input sentence can be used to predict the next word. Gao at p. 2.

67. While the above example is a chain-like structure, RNNs are not so limited. Gao at p. 2.

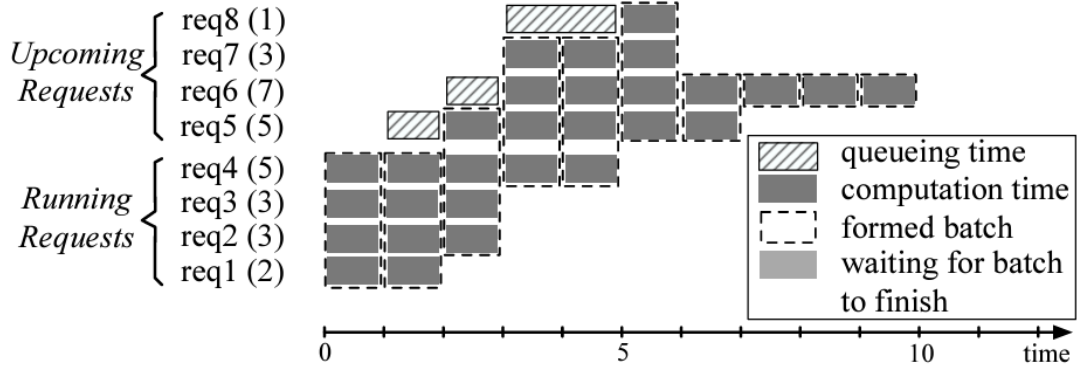
68. Traditional approaches to improving throughput have focused on the training phase. Gao at p. 1. One such example is graph batching, which “collects a batch of inputs, combines their dataflow graphs into a single graph whose operators represent batch execution of corresponding operators in the original graphs, and submits the combined graph to the backend for execution.” Gao at p. 1.

69. Graph batching, however, is not appropriate for inference. Gao at p. 1 and 4. For inference, “inputs [] arrive at different times,” and graph batching waits for an outgoing batch of requests to finish their execution before adding the new batch. Gao at p. 1. Additionally, when the batched input has different sizes, not all operators in a graph can be batched fully, reducing throughput. Gao at p. 1-2.

70. Figure 5 illustrates the difference between graph batching and cellular batching when processing the same 8 requests. Gao at p. 6; Gao at Fig. 5.



(a) Graph Batching



(b) Cellular Batching

Figure 5: The timeline of graph batching and Cellular Batching when processing 8 requests from req1 to req8. The number shown in parenthesis is the request’s sequence length, e.g. req1(2) means req1 has a sequence length of 2. Each row marks the lifetime of a request starting from its arrival time. Req1-4 are *Running Requests* as they arrive at time 0 and have started execution. Req5-8 are *Upcoming Requests* that arrive after the Req1-4.

71. Gao describes the difference as follows:

In the beginning of time ($t=0$), the first 4 requests (req1-4) arrive. Under graph batching, these 4 requests form a batch and their corresponding dataflow graphs are fused together and submitted to the backend for execution. The system does not finish executing the fused graph until time $t=5$, as the longest request in the batch (req4) has a length of 5. In

the meanwhile, newly arrived requests (req5-8) are being queued up and form the next batch. The system starts executing the next batch at $t=5$ and finishes at $t=12$. Under cellular batching, among the first 4 requests, the system forms two fully batched tasks, each performing the execution of a single (4way batched) RNN cell. At $t=2$, the second task finishes, causing req1 to complete and leave the system. Since a new request (req5) has already arrived, the system forms its third fully batched task containing req2-5 at $t=2$. After finishing this task, another two existing requests (req2, req3) depart and two new ones are added (req6, req7) to form the fourth task.

Gao at p. 6.

72. To address these issues, Gao et al. (2018) introduce the concept of cellular batching, a technique that improves both latency and throughput of RNN inference. Gao at p. 2. The authors called their cellular batching system “BatchMaker.” Gao at p. 2. BatchMaker consists of a manager and worker threads, handling request processing and GPU execution, respectively. Gao at Fig. 6.

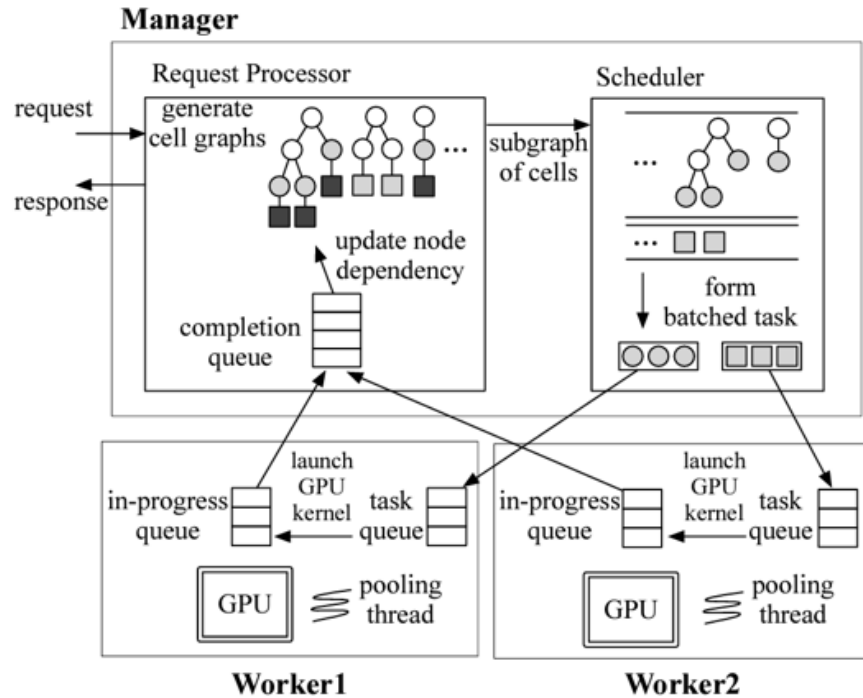


Figure 6: The system architecture of BatchMaker. In the cell graph, black means computed nodes, grey means nodes whose input is ready, and white means input dependency is not satisfied.

73. The manager, which consists of a request processor and scheduler, processes arriving requests and submits batch computation tasks to worker for execution. Gao at p. 6. The request processor, “generates the corresponding cell graph for the request” and tracks and updates the dependencies of each node. Gao at p. 5. “Once the dependencies have been satisfied (aka its inputs are ready” the scheduler “forms batched tasks among ready nodes of the same cell type.” Gao at p. 5.

74. Once the desired batch size is reached, the task is passed to the workers. Gao at p. 6-7. “Workers execute tasks on GPUs and notify the manager when its tasks complete.” Gao at p. 5.

75. In the BatchMaker system, as implemented by Gao, the authors saw BatchMaker “reduce latency by 17.5-90.6% and improves throughput by 25-80% compared with state-of-the-art systems including TensorFlow, MXNet, TensorFlow Fold, and DyNet.” Gao at p. 14.

B. Summary of Katharopoulos

76. *Transformers are RNNs: Fast Autoregressive Transformers with Linear Attention* by Katharopoulos et al., (“Katharopoulos”) published in 2020 proposes a technique called “linear transformers” that perform similarly to vanilla transformers but are “up to 4000x faster on autoregressive prediction of very long sequences.” Katharopoulos at p. 1.

77. Katharopoulos observes that while transformer models, originally introduced by Vaswani et al., (2017), “have demonstrated impressive results” when dealing with natural language, audio, and images, “these benefits often come with a very high computational and memory costs.” Katharopoulos at p. 1. The reason for these high computational and memory costs are attributed to “self-attention, which processes contexts of N inputs with a quadratic memory and time complexity $O(N^2)$.” Katharopoulos at p. 1.

78. Accordingly, Katharopoulos introduces “the *linear transformer* model that significantly reduces the memory footprint and scales linearly with respect to the context length.” Katharopoulos at p. 1.

79. In their paper, Katharopoulos describes the basic transformer architecture introduced by Vaswani in § 3.1 of his paper and proposes its *linear transformer* in § 3.2 and § 3.3 before rewriting the transformer as a recurrent neural network in § 3.4. Katharopoulos at p. 2-5.

80. As observed by Katharopoulos:

transformer models are considered to be fundamentally different approach to recurrent neural networks. However, from the causal masking formulation in § 3.3 ...it becomes evident that any transformer layer with causal masking can be written as a model that, given an input, modifies an internal state and then predicts an output, namely a Recurrent Neural Network (RNN).”

Katharopoulos at p. 5.

81. Katharopoulos formalizes the transformer layer for transformers as an RNN in the following equation where x_i denotes the i -th input and y_i , the i -th output for a specific transformer layer:

$$s_0 = 0, \quad (16)$$

$$z_0 = 0, \quad (17)$$

$$s_i = s_{i-1} + \phi(x_i W_K) (x_i W_V)^T, \quad (18)$$

$$z_i = z_{i-1} + \phi(x_i W_K), \quad (19)$$

$$y_i = f_l \left(\frac{\phi(x_i W_Q)^T s_i}{\phi(x_i W_Q)^T z_i} + x_i \right). \quad (20)$$

82. Katharopoulos at p. 5. Importantly, Katharopoulos recognized that the “formulation does not impose any constrain on the feature function and it can be used for representing *any transformer* model, in theory even the ones using softmax attention.” Katharopoulos at p. 5.

Katharopoulos ran several experiments utilizing their *linear transformer* and observed that their model achieved competitive performance with respect to the state-of-the-art transformer architectures, while requiring significantly less GPU memory and computation. Katharopoulos at p. 5; *see also* Katharopoulos at p. 5-8 (describing the performance of the *linear transformer* across real world applications of image generation and speech recognition.)

VII. REASONS THE CHALLENGED CLAIMS OF THE ‘775 PATENT ARE UNPATENTABLE

A. GROUND 1: Claims 1-18 are rendered obvious under 35 U.S.C. § 103 by Gao in view of Katharopoulos

1. Summary of the Combination and Motivation to Combine

83. It is straight forward for a POSITA to leverage the paradigm of cellular batching presented by Gao to achieve an efficient scheduling scheme for at least certain classes of transformer models. Whereas the work of Gao shows the efficacy of cellular batching on a Recurrent Neural Network (RNN), Katharopoulos teaches a transformer model that operates like an RNN. As such, a POSITA understands the methodology of Gao can be leveraged to batch requests in the context of a machine learned transformer model.

84. This combination leverages two key observations that would be known to a POSITA: (1) whenever a Decoder-Only-Transformer (DOT) uses an attention mechanism with a linearized attention-function, it can be formulated as an RNN

circuit as taught by Katharopoulos, and (2) transformers operate in a sequence-to-sequence fashion and thus, when DOTs are formulated as an RNN circuit they behave in a way that is similar to the Seq2Seq RNN model. *See generally* Sutskever et al.

85. With these two observations, a POSITA would be motivated to leverage Gao’s batching scheme to achieve efficient batching in a DOT with a linearized attention function as taught by Katharopoulos.

Gao’s Cellular Batching:

86. In their work, Gao presents a batching scheme (cellular batching) that operates on the functional cell level of an RNN rather than on the whole data graph as typically done in earlier graph batching schemes. In essence, the key insight of Gao is that: “RNN has the unique feature that it contains many identical computational units connected with each other. Cellular batching exploits this feature to 1) batch at the level of RNN cells instead of whole dataflow graphs, and 2) let new requests join the execution of current requests and let requests return to the user as soon as they finish.” Gao p. 4. This insight allows one to batch any number of requests (depending on the underlying processing and memory capacity) and to execute these batched requests in parallel. In addition, with cellular batching, requests of different sizes can be batched together without having to incur the cost of extra padding, another shortcoming of standard batching techniques.

87. Finally, using cellular batching, newly arriving requests can join the execution of ongoing batches, while other requests can be returned to the user as soon as they finish, without the need to wait for the entire batch to finish simultaneously.

88. Cellular batching has been shown to provide a significant improvement in terms of latency and throughput when running inference on requests of variable lengths. Gao built an inference system based on cellular batching that is called BatchMaker.

Katharopoulos Linearized Attention and Transformers as RNNs

89. As discussed earlier, the attention mechanism has been used in many different works. In the work of Vaswani, the attention mechanism uses an attention function that computes the weight assigned to each Value using the exponential of the scaled dot product between Queries and Keys, which we refer to as a Softmax attention function.

90. Many papers have discussed specifics and features of different attention-functions (e.g., Tsai et al.); however, most relevant to our discussion here is the class of attention-functions introduced in Katharopoulos.

91. In their work, Katharopoulos introduces linearized attention-functions which are defined as a linear dot-product of kernel feature maps. In this type of attention-function, the attention mechanism itself still relies on the Query, Key and

Value extracted in the same way defined in Vaswani. However, rather than using an exponential function directly on the product of the Query and the Key, a kernel feature map is first applied to the Query and Key values before their dot product is computed.

92. To this end, Katharopoulos uses this class of attention-function to reduce the complexity of Attention computation and dramatically accelerate the autoregressive nature of Transformers especially during inference. Moreover, this type of attention-function can help in increasing the context length without losing on efficiency. In fact, Katharopoulos used linearized attention-functions to show that any transformer layer that uses masked attention can be written as an RNN cell.

93. More precisely, using a linearized attention-function, Katharopoulos shows that a Transformer's Decoder layer can be viewed as an RNN computational cell (Decoder-Cell) that takes in a single encoded input from a sequence of inputs, modifies an internal state accordingly, and then produces an output.

94. When processing a sequence of inputs, the cell's internal state is initialized to zero when processing the first element in the sequence, and for every consecutive sequence element, the internal state is initialized using the internal state of the previous cell. Here, all the cells processing the input sequence are identical, as is the case in RNNs, and each cell has an output that can be passed to a different

Decoder-Cell corresponding to a higher layer of the Decoder stack or can be processed to predict the next element in the sequence.

95. The work of Katharopoulos presents a radical improvement in the cost of computing attention (the most expensive component within a Transformer). For a context window of length N , a softmax attention-function requires a time and memory complexity of $O(N^2)$ to compute, whereas using a linearized attention-function the same can be computed using a time and memory complexity of $O(N)$.

96. Moreover, Katharopoulos argues that, in theory, even a softmax attention-function could be cast as an RNN cell, given that masked Attention is used “it becomes evident that any transformer layer with causal masking can be written as a model that, given an input, modifies an internal state and then predicts an output, namely a Recurrent Neural Network (RNN).” Katharopoulos p. 5. In fact, the softmax attention-function itself has been approximated using a linear dot product of feature maps (Blanc et al. and Rawat et al.) which is why Katharopoulos concludes that it could be possible to use, even a softmax attention-function, with their introduced work “approximating the RBF kernel with random Fourier features could allow us to use models pretrained with softmax attention.” Katharopoulos p. 8.

Combining Gao’s Cellular Batching with Katharopoulos’s Transformer

97. Given the significant speed ups presented by cellular batching and linearized attention-functions, a POSITA can merge these two works into a fast-batching scheme for the well-known Decoder-Only-Transformer (“DOT”) architecture. Specifically, the link between these two works becomes clear when casting a Transformers’ Decoder into an RNN cell. In this view, the computational graph of a DOT becomes similar to that of a Seq2Seq RNN model, and thus, cellular batching can be immediately leveraged to batch inputs into the same cell-types as detailed in Gao.

98. To elaborate, I first explain how a DOT operates. A typical DOT circuit is made of a stack of L layers of Decoders. Each Decoder layer has different parameters (learned during training) and takes in a request in the form of a sequence of n (up to a maximum of N) input elements, each encoded as a vector of dimension d , and outputs a set of output elements with the same sequence length and vector dimensionality. So, a Decoder layer maps a $(d \times n)$ matrix into a $(d \times n)$ matrix. Stacking L layers of Decoders does not impact the mapping space as it will simply iterate different functional mappings over the $(d \times n)$ space until the final Decoder layer outputs the final output of size $(d \times n)$. To this end, a Transformer’s final output is expected to be a prediction of the next element in the sequence of inputs, just as is the case in Seq2Seq RNNs. Thus, the final output of the Decoder stack is passed

through a prediction-component (typically used in sequence transduction models like Seq2Seq RNNs) which is made of a linear projection to the possible output space followed by a softmax function (this is the output-softmax and is not to be confused with the attention-softmax discussed earlier) to produce a probability distribution over the possible output space.

99. Finally, a sampler is used to choose the predicted output element using the produced probability distribution. Once the output is produced it is added to the n input elements that will now be encoded as a $(d \times (n+1))$ matrix and passed through the Decoder stack in order to predict the $n+2$ elements in the sequence.

100. This process is repeated until the final output generated is a special output element (e.g. $\langle \text{eos} \rangle$). At this point, the sampler stops sampling and can return the response to the input request in the form of the newly generated elements. This is, in essence, how a standard DOT circuit operates.

101. In the work of Katharopoulos, each Decoder layer is viewed as a single RNN cell (Decoder-Cell). The Decoder-Cell only inputs a single element from the sequence of input elements, where the input elements are also encoded as a single vector of dimension d (one column from the matrix discussed in the previous paragraphs). In response, the Decoder-Cell outputs a vector of the same size as the input and updates the state-vector reflecting the internal state of the Decoder-Cell at the time-step of the input. The output of a Decoder-Cell within the first layer is then

passed to a different Decoder-Cell at the next layer (e.g. layer 2) while the updated state-vector is used for processing the next input element within the input sequence.

102. Accordingly, an input of n elements passing through a DOT with a linearized attention-function has a computational graph which can be viewed as a matrix of Decoder-Cells with size $(L \times n)$. Each layer (row in the computational graph) comprises identical cells that process the different encoded input elements and only differ in the state-vector passed from the previous (identical) Decoder-Cell within the same computational row.

103. The first Decoder-Cells in all layers (the first column in the computational graph) is initialized with a zero state, as shown in Katharopoulos (Eq. 16 and 17). The final layer of Decoder-Cells does not pass their output anywhere and only update the state of the next Decoder-Cell in the same final layer (computational row). The only exception here is the final Decoder-Cell of the final layer (the top right Decoder-Cell of position (L,d) in the computational graph) which passes its output through the prediction-component (can be viewed as a different computational cell) made of the final linear projection and the output-softmax function in order to produce the predicted next token using a sampler as previously explained.

104. Just as done in Seq2Seq RNNs, the predicted output element is now passed through another column made of L Decoder-Cells identical to the previous

Decoder-Cells used in each column of the computational graph. Moreover, the Decoder-Cells in the new column get their states initial value from the previous column of Decoder-Cells. That is, the computational graph can now be seen as being made of $(L \times (n+1))$ decoder cells. The new Decoder-Cell column behaves exactly as the last Decoder-Cell column in that it passes its final output through a prediction-component cell to predict the next element. And just as done in the standard DOT architecture (or Seq2Seq RNNs), this process continues until the special element (e.g. $\langle \text{eos} \rangle$) is generated, at which point the computational graph is terminated.

105. As can be seen from the description here, this process is essentially identical to what is done in a Seq2Seq RNN model which can also use multi-levels of RNN cells and employs two types of cells, encoder-cells and decoder-cells. The only difference here is that the decoding process starts at the n th time-step rather than the $n+1$ time-step while only using a single type of cells. The rest is essentially identical in process to the Seq2Seq RNN model. The first n Cells in the first layer take in the encoded input elements (similar to encoder-cells in Seq2Seq), while for each newly generated element in the sequence the output element is encoded and passed as an input to the newly added Decoder-Cells (similar to the decoder cells in Seq2Seq). Thus, using the work of Katharopoulos, a POSITA can immediately appreciate a DOT with linearized attention function as a Seq2Seq RNN circuit.

106. Once casted in the language of RNNs, a POSITA would understand that a DOT with linearized attention can immediately leverage cellular-batching exactly as presented in Gao. In essence, one can use Gao’s BatchMaker system exactly as-is and without any changes to produce a batching system for DOT inference requests. The reason this application is immediate and straightforward to a POSITA is the ability to cast DOTs with linearized attention as an RNN with a computational graph that reuses the same cells for different steps of the computation. Any internal dependency between the Decoder-Cells, along with handling of variable-length requests, are directly handled by the BatchMaker system.

107. A POSITA would fully appreciate the effect of applying BatchMaker to DOTs by observing the improvements that BatchMaker produces when applied to Seq2Seq RNNs. For example, in dynamic RNN models, the length of the computational path is not known beforehand, but rather is dependent on the output of the circuit itself, as is the case with Seq2Seq and DOTs. This feature presents a challenge to batching since even requests with identical input sizes might have an early or late finish depending on how the model responds to each request. In contrast, BatchMaker demonstrates how effective it is in these cases since cellular batching operates on the cell level which allows different requests to join or leave the execution at any time step.

108. Finally, it is worth noting here that a POSITA applying Gao's Cellular batching to other classes of Transformer circuits, that are different from RNNs, may require further modifications. However, this is not the case for DOTs, which have a forward-moving dependency stemming from the use of Causal masking. With Causal masking, the direction of dependency is very similar to the forward passing state used in RNNs. Moreover, the attention mechanism, which is the defining characteristic of Transformers, requires input elements to, at least in the case of DOTs, have access to all the Keys and Values of all the previous input elements.

109. While in other transformer models the forward dependency must grow with the order of the input element (within a sequence of elements) in a way that requires an increasing number of computational cells to handle; however, this would only be the case for transformer models that use specific types of attention-functions that cannot linearize.

110. The attention mechanism relies on computing a weighted sum over the Values, with weights computed using a compatibility function between the Queries and the Keys. *See generally* Vaswani. Therefore, an intelligent choice of how compatibility is computed between the Queries and the Keys, as done in Katharopoulos, can easily allow the attention mechanism to be efficiently computed without the need for growth in the number of used computational cells.

Conclusion:

111. A POSITA would have been motivated to and had a reasonable chance of success to combine cellular batching, as taught by Gao, and DOT with linearized attention functions, as taught by in Katharopoulos. For the reasons discussed above, a POSITA would understand the link between these two works and appreciate the significant speedup that would be gained by applying Gao’s BatchMaker system to any Decoder-Only-Transformer DOT, specifically ones using a linearized attention function as taught by Katharopoulos.

2. Detailed Claim Analysis

Claim

1.

[PRE] A method of dynamically executing batches of requests on one or more execution engines running a machine-learning transformer model, comprising:

112. Gao teaches a method of dynamically executing batches through the introduction of their system called BatchMaker, which employs “the technique of cellular batching, which improves both the latency and throughput of RNN inference.” Gao at p. 1. By making batching decisions at the granularity of the cell level, BatchMaker teaches a system that “dynamically assembles a batched cell for execution as requests join and leave the system.” *See e.g.*, Gao at p. 1.

113. In particular, “[a]s each input arrives, BatchMaker breaks its computation graph into a graph of cells and dynamically decides the set of common cells that should be batched together for execution.” Gao at p. 2.

114. To perform the cellular batching, BatchMaker proposes a system architecture of a “Manager and Worker. The manager processes arriving requests and submits batched computation tasks to workers for execution.” Gao at p. 5. A copy of the system architecture for BatchMaker is as follows:

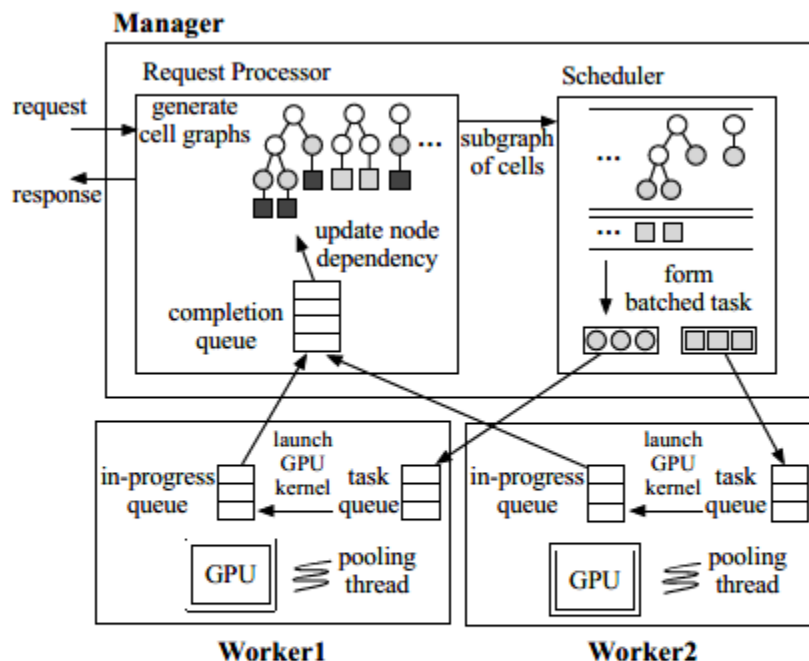


Figure 6: The system architecture of BatchMaker. In the cell graph, black means computed nodes, grey means nodes whose input is ready, and white means input dependency is not satisfied.

115. As can be seen from Figure 6, BatchMaker includes multiple “execution engines” (e.g., workers) that “execute tasks on GPUs”. Gao at p. 6.

116. While Gao relates to RNN, and not a machine-learning transformer model, its dynamically batching cells would improve the latency associated with inference in a machine-learning transformer model such as the *linear transformer* model of Katharopoulos. See **Summary of the Combination and Motivation**

to **Combine Supra** (describing the implementation of a linear transformer as disclosed by Katharopoulos with the cellular batching disclosed by Gao).

117. Katharopoulos introduces a machine-learning transformer model representing the transformer model as a Recurrent Neural Network (RNN). *See e.g.*, Katharopoulos at p. 5 (discussing that “from the causal masking formulation in 3.3 and the discussion in the previous section, it becomes evident that any transformer layer with causal masking can be written as a model that, given an input, modifies an internal state then predicts an output, namely a Recurrent Neural Network (RNN).”)

118. Katharopoulos formalizes the transformer layer for transformers with causal masking as an RNN in the following equation where x_i denotes the i -th input and y_i , the i -th output for a specific transformer layer:

$$s_0 = 0, \quad (16)$$

$$z_0 = 0, \quad (17)$$

$$s_i = s_{i-1} + \phi(x_i W_K) (x_i W_V)^T, \quad (18)$$

$$z_i = z_{i-1} + \phi(x_i W_K), \quad (19)$$

$$y_i = f_l \left(\frac{\phi(x_i W_Q)^T s_i}{\phi(x_i W_Q)^T z_i} + x_i \right). \quad (20)$$

Katharopoulos at p. 5.

119. Importantly, Katharopoulos recognized that the “formulation does not impose any constrain on the feature function, and it can be used for representing *any transformer* model, in theory even the ones using softmax attention.” Katharopoulos at p. 5. *See also* Summary of the Combination and Motivation to Combine

Supra (discussing the application of cellular batching to DOT with a linearized attention function).

1[A] receiving, by a serving system, one or more requests for execution, the serving system including a scheduler and one or more execution engines each coupled to access a machine-learning transformer model including at least a set of decoders;

120. As discussed above, Gao proposes a system architecture of a “Manager and Worker. The manager processes arriving requests and submits batched computation tasks to workers for execution.” Gao at p. 5. A copy of the system architecture for BatchMaker is as follows:

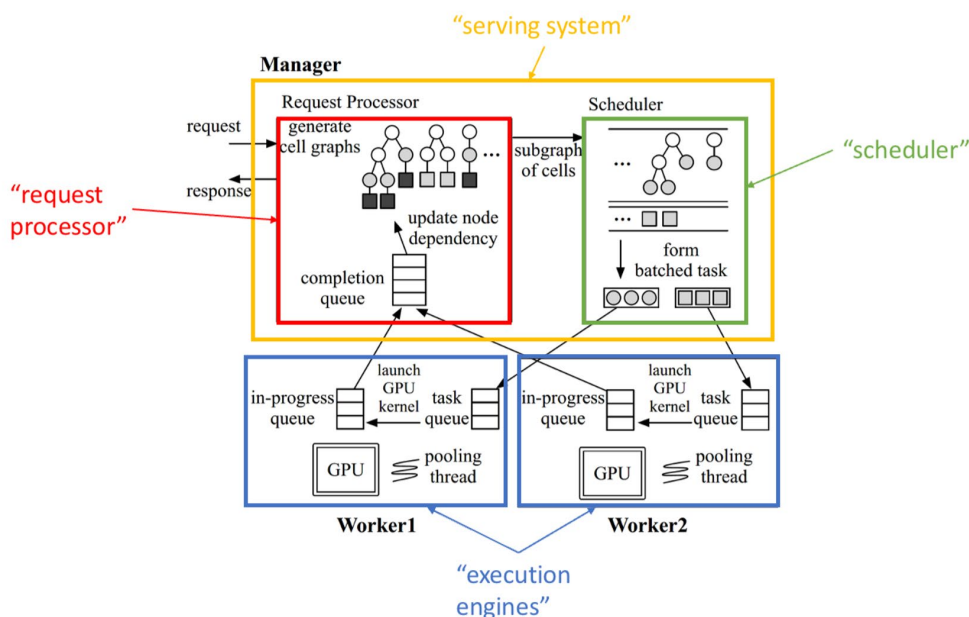


Figure 6: The system architecture of BatchMaker. In the cell graph, black means computed nodes, grey means nodes whose input is ready, and white means input dependency is not satisfied.

121. As can be seen from Figure 6, BatchMaker includes a manager as a serving system that “processes arriving requests and submits batched computation

tasks to workers for execution.” Gao at p. 5. The manager in BatchMaker has two components, a “request processor” for tracking the progress of execution for the requests, and a scheduler for “determining which cells from different requests would form a batched task, and selects a worker to execute the task.” Gao at p. 5.

122. BatchMaker also includes an execution engine, namely workers, that use GPUs to “move[]a task from the task queue to the in-process queue once the task’s corresponding GPU kernel has been issued.” Gao at p. 6. Using a pooling mechanism, the worker continues to move “finished tasks from the in-process queue and pushes it into the completion queue in the request processor.” Gao at p. 6.

123. As discussed above, Gao relates to RNN and not a machine-learning transformer model including at least a set of decoders. However, its dynamically batching cells would improve the latency associated with inference in a machine-learning transformer model such as Katharopoulos's linear transformer model which applies to a set of decoders since decoders are known to use causal masking. Motivation to Combine Discussion *Supra*.

124. Katharopoulos introduces a machine-learning transformer model that represents the transformer model as a Recurrent Neural Network (RNN). *See e.g.*, Katharopoulos at p. 5 (discussing that “from the causal masking formulation in 3.3 and the discussion in the previous section, it becomes evident that any transformer layer with causal masking can be written as a model that, given an input, modifies

an internal state then predicts an output, namely a Recurrent Neural Network (RNN).”).

125. Katharopoulos formalizes the transformer layer for transformers using causal masking for decoder-only-transformer as an RNN in the following equation where x_i denotes the i -th input and y_i , the i -th output for a specific transformer layer:

$$s_0 = 0, \quad (16)$$

$$z_0 = 0, \quad (17)$$

$$s_i = s_{i-1} + \phi(x_i W_K) (x_i W_V)^T, \quad (18)$$

$$z_i = z_{i-1} + \phi(x_i W_K), \quad (19)$$

$$y_i = f_l \left(\frac{\phi(x_i W_Q)^T s_i}{\phi(x_i W_Q)^T z_i} + x_i \right). \quad (20)$$

Katharopoulos at p. 5.

126. Importantly, Katharopoulos recognized that the “formulation does not impose any constrain on the feature function, and it can be used for representing *any transformer* model, in theory even the ones using softmax attention.” Katharopoulos at p. 5.

127. In view of the above, combining Gao and Katharopoulos would teach a system where Gao’s Batchmaker has access to a machine-learning transformer model including at least a set of decoders. *See also* Summary of the Combination and Motivation to Combine *Supra* (discussing the application of cellular batching to Decoder-Only-Transformer architectures).

1[B] scheduling, by the scheduler, a batch of requests including the one or more requests for execution on an execution engine;

128. As discussed above, BatchMaker proposes a system architecture of a “Manager and Worker. The manager processes arriving requests and submits batched computation tasks to workers for execution.” Gao at p. 5. A copy of the system architecture for BatchMaker is as follows:

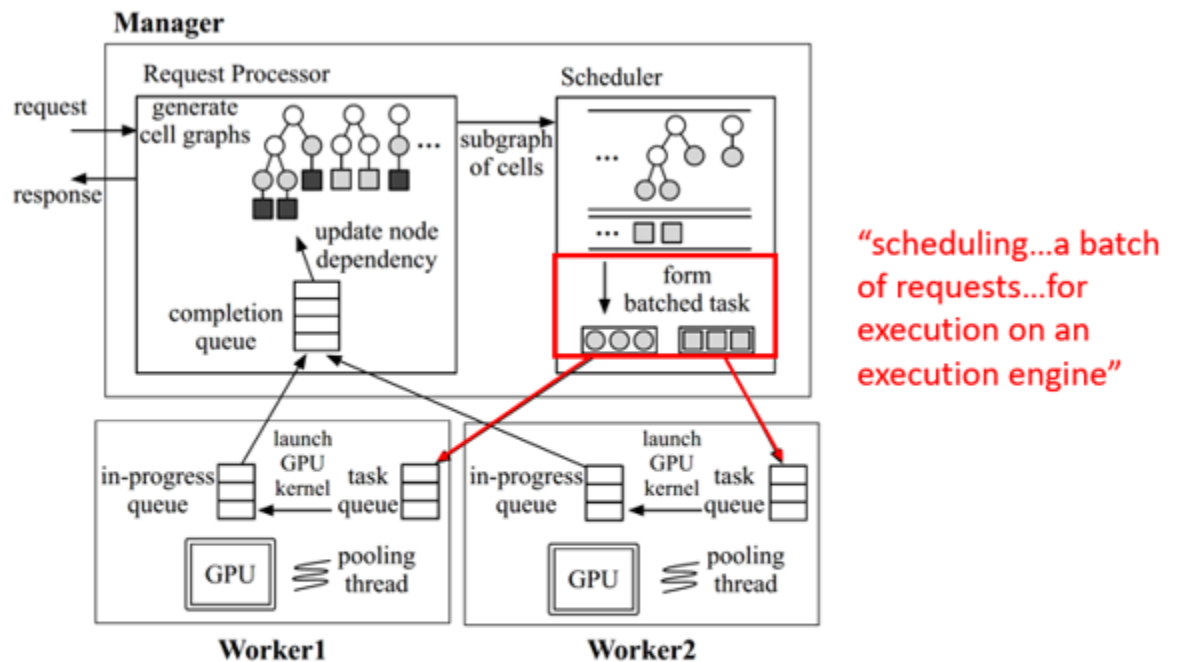


Figure 6: The system architecture of BatchMaker. In the cell graph, black means computed nodes, grey means nodes whose input is ready, and white means input dependency is not satisfied.

129. As can be seen from Figure 6, BatchMaker includes a manager that “processes arriving requests and submits batched computation tasks to workers for execution.” Gao at p. 5. The manager in BatchMaker has two components, a “request processor” for tracking the progress of execution for the requests, and a

scheduler for “determining which cells from different requests would form a batched task, and selects a worker to execute the task.” Gao at p. 5.

130. Gao provides that “[t]he scheduler needs to make decisions on what nodes should be batched together to form a task and what tasks to be pushed to which workers.” Gao at p. 6. Gao goes on to provide detail for the aspects the scheduler must account for, namely “locality, priority, and utilization of multiple GPUs, which are often in conflict with each other.” Gao at 6. “Locality refers to the preference that 1) the same set of requests should be batched together if they are to execute the same sequence of nodes, and the execution of the sequence of nodes should stick to the same GPU.” Gao at p. 6. “Priority refers to the ability to prefer the execution of one type of cell over another.” Gao at p. 6.

131. Gao also provides the basic algorithm employed by BatchMaker for scheduling and creation of batches. Gao at p. 6.

Algorithm 1: Scheduling and Batching Algorithm

```
1 Bsizes: a set of supported batch sizes.
2 CellTypes: a set of cell types, each associated with a priority.
3 MaxTasksToSubmit: the maximum number of tasks that can
  be submitted to a worker.
4 def Schedule(worker):
5   S ← {ct ∈ CellTypes | ct.NumReadyNodes() ≥
     Bsizes.Max()};
6   if S.Size() = 0 :
7     S ← {ct ∈ CellTypes | ct.NumRunningTasks() = 0 and
       ct.NumReadyNodes() > 0};
8   if S.Size() = 0 :
9     S ← { ct ∈ CellTypes | ct.NumReadyNodes() > 0};
10  ct ← GetCellTypeWithMaxPriority(S);
11  Batch(ct, worker);
12 def Batch(ct, worker):
13  num_tasks ← 0;
14  while num_tasks < MaxTasksToSubmit :
15    batch ← FormBatchedTask(ct, worker);
16    if batch.Size() ≥ Bsizes.Min() or num_tasks = 0 :
17      SubmitBatchedTask(batch, worker);
18      UpdateNodesDependency(batch);
19      num_tasks++;
20      for subgraph ∈ batch :
21        subgraph.pinned ← worker.id;
22        # pinned is unset once subgraph has no
23        task running
24    else:
25      break;
26 def FormBatchedTask(ct, worker):
27  batch ← {};
28  for subgraph ∈ ct.subgraphs :
29    if subgraph.pinned ∈ {None, worker.id} :
30      for node ∈ subgraph's ready nodes :
31        batch ← batch ∪ {node};
32        if batch.Size() = Bsizes.Max() :
33          return batch;
34  return batch;
```

1[C] generating, by the execution engine, a first set of output tokens by applying the transformer model to a first set of inputs for the batch of requests, wherein applying the transformer model comprises applying at least one batch operation to one or more input tensors associated with the batch of requests;

132. As discussed above, Gao discloses an execution engine, namely workers, that use GPUs to “move[]a task from the task queue to the in-process queue once the task’s corresponding GPU kernel has been issued.” Gao at p. 6. Using a

pooling mechanism, the worker continues to move “finished tasks from the in-process queue and pushes it into the completion queue in the request processor.” Gao at p. 6; *see also* Gao at Fig. 6.

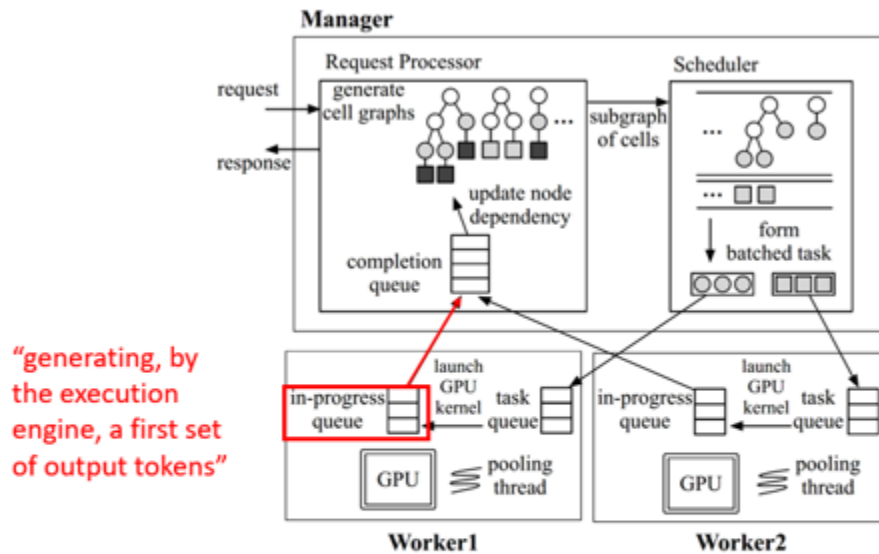
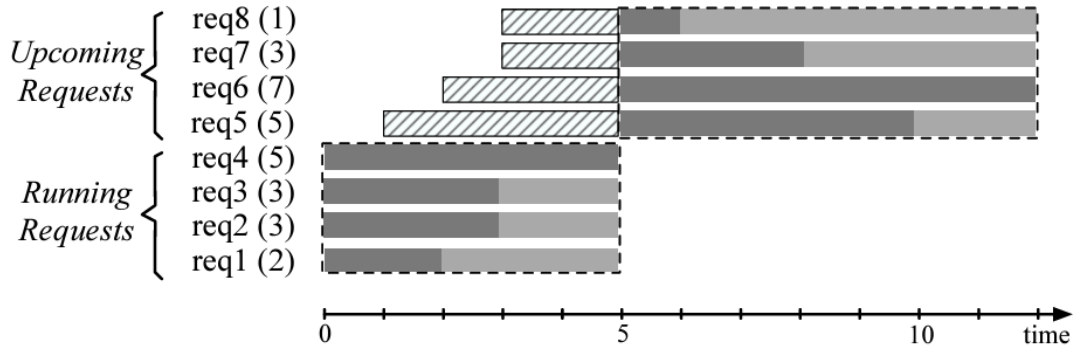
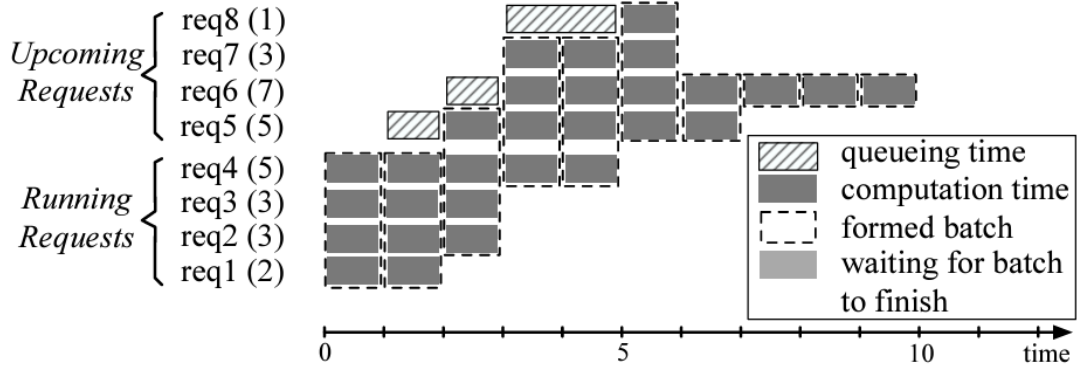


Figure 6: The system architecture of BatchMaker. In the cell graph, black means computed nodes, grey means nodes whose input is ready, and white means input dependency is not satisfied.

133. The behavior of Gao’s cellular batching when being processed is illustrative Figure 5. Gao at p. 6; Gao at Fig. 5.



(a) Graph Batching



(b) Cellular Batching

Figure 5: The timeline of graph batching and Cellular Batching when processing 8 requests from req1 to req8. The number shown in parenthesis is the request's sequence length, e.g. req1(2) means req1 has a sequence length of 2. Each row marks the lifetime of a request starting from its arrival time. Req1-4 are *Running Requests* as they arrive at time 0 and have started execution. Req5-8 are *Upcoming Requests* that arrive after the Req1-4.

134. Gao describes the process as follows:

In the beginning of time ($t=0$), the first 4 requests (req1-4) arrive... Under cellular batching, among the first 4 requests, the system forms two fully batched tasks, each performing the execution of a single (4way batched) RNN cell. At $t=2$, the second task finishes, causing req1 to complete and leave the system. Since a new request (req5) has

already arrived, the system forms its third fully batched task containing req2-5 at $t=2$. After finishing this task, another two existing requests (req2, req3) depart and two new ones are added (req6, req7) to form the fourth task.

Gao at p. 6. (emphasis added).

135. Similarly, and as discussed above, Gao provides figure 1, which shows an unfolded dataflow graph for an input where the RNN is used for predicting the word most likely to follow the following input text: “system research is.” Gao at p. 2; Gao at Fig. 1. “At each time step, one input position is consumed and the calculated value of the hidden unit is then passed to the successor cell in the next time step.” Gao at p. 2. Once all three steps are completed, the full context is known and the input sentence can be used to predict the next word. Gao at p. 2.

136. Moreover, Gao provides requests include a sequence of input tokens (e.g., $X(1)$, $X(2)$, $X(r)$) to predict an output token (e.g., “the most likely next word”):

“in its simplest form, we can view RNNs as operating on an input sequence, $X = [x(1), x(2), \dots, x(\tau)]$, where $x(i)$ represents the input at the i -th position (or timestep). For language processing, the input X would be a sentence, and $x(i)$ would be the vector embedding of the i -th word in the sentence... For example, we can use an RNN to process the input sentence “system research is” and to derive the most likely next word from the RNN’s output. Figure 1 shows the unfolded dataflow graph for this input.”

137. Similarly, I note that a POSITA would have understood that input requests include input “tokens” and replies include “output tokens.” For example, the seminal 2017 article, “Attention Is All You Need,” introduced transformer models and disclosed input requests include “input tokens.” *See* Vaswani, 5 (3.4

Embeddings and Softmax) (“Similarly to other sequence transduction models, we use learned embeddings to convert the input tokens and output tokens to vectors of dimension d_{model} .”); 6 (5.1 Training Data and Batching) (“Sentences were encoded using byte-pair encoding [3], which has a shared source-target vocabulary of about 37000 tokens.”).

138. To demonstrate the performance of BatchMaker, Gao provides an example of latency and performance improvements using Sequence to Sequence (“Seq2Seq”), a widely used RNN model for machine translation.

139. Gao provides, that:

“A basic Seq2Seq model contains two types of RNN cells: encoder and decoder, as depicted in Figure 12. The encoder takes in a sequence of words as input. In each step, the encoder and decoder will convert a word to a vector by doing an embedding lookup, then feed this vector to an RNN Cell. The first decoder cell takes in the output state of the encoder and a $\langle \text{go} \rangle$ symbol as input and computes states, which are passed to the succeeding decoder cell. In addition to the state, the decoder cell outputs a word as well, which is obtained by applying a linear transformation and an argmax. The output word is also fed to the next step as the input. When the decoder outputs the $\langle \text{eos} \rangle$ symbol, it means the decoder has finished, and there will be any more decoder steps.”

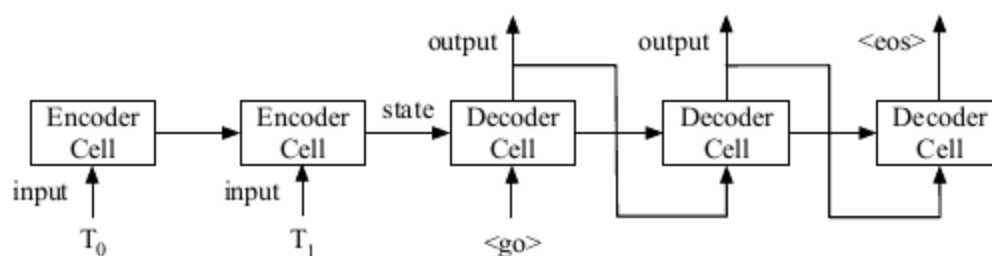


Figure 12: Seq2Seq with “feed previous” decoder

140. Using the BatchMaker technique, experiments show 17.5-90.5% latency improvement and 25-60% throughput performance. Gao at 2.

141. Katharopoulos introduces a machine-learning transformer model representing the transformer model as a Recurrent Neural Network (RNN). *See e.g.*, Katharopoulos at p. 5 (discussing that “from the causal masking formulation in 3.3 and the discussion in the previous section, it becomes evident that any transformer layer with causal masking can be written as a model that, given an input, modifies an internal state then predicts an output, namely a Recurrent Neural Network (RNN).”)

142. Katharopoulos formalizes the transformer layer for transformers (using causal masking) as an RNN in the following equation where x_i denotes the i -th input and y_i , the i -th output for a specific transformer layer:

$$s_0 = 0, \quad (16)$$

$$z_0 = 0, \quad (17)$$

$$s_i = s_{i-1} + \phi(x_i W_K) (x_i W_V)^T, \quad (18)$$

$$z_i = z_{i-1} + \phi(x_i W_K), \quad (19)$$

$$y_i = f_l \left(\frac{\phi(x_i W_Q)^T s_i}{\phi(x_i W_Q)^T z_i} + x_i \right). \quad (20)$$

143. Katharopoulos at p. 5. Importantly, Katharopoulos recognized that the “formulation does not impose any constrain on the feature function, and it can be used for representing *any transformer* model, in theory even the ones using softmax attention.” Katharopoulos at p. 5. *See also* Summary of the Combination and Motivation to Combine *Supra* (discussing the application of cellular batching to DOT with a linearized attention function).

144. Accordingly, Katharopoulos teaches that a first set of output tokens (e.g., “ y_i , the i -th output”) by applying the transformer model (e.g., equation 20) to a first set of inputs (e.g., “ x_i denotes the i -th input”) for the batch of requests, wherein applying the transformer model comprises applying at least one batch operation, as disclosed by Gao, to one or more input tensors associated with the batch of requests. “A simple cell contains a few tensor operators... Two cells are of the same type if they have identical sub-graphs, share the same parameter weights, and expect the same number of identically-shaped input tensors. Cells with the same type can be batched together if there is no data dependency between them.” Gao at p. 4. “For a

cell to be considered batchable, the first dimension of each of its input tensors should be the batch dimension.” Gao at p. 5. *See also* Summary of the Combination and Motivation to Combine *Supra*.

1[D] receiving, by a request processor, a new request from a client device, the new request including a sequence of input tokens;

145. As discussed above, BatchMaker proposes a system architecture of a “Manager and Worker. The manager processes arriving requests and submits batched computation tasks to workers for execution.” Gao at p. 5. A copy of the system architecture for BatchMaker is as follows:

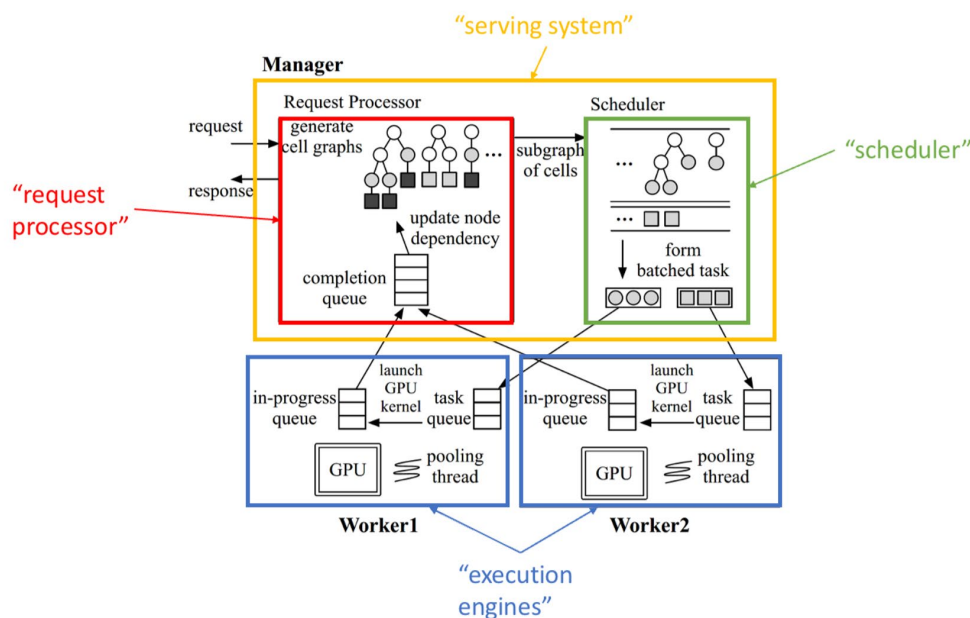


Figure 6: The system architecture of BatchMaker. In the cell graph, black means computed nodes, grey means nodes whose input is ready, and white means input dependency is not satisfied.

146. As can be seen from Figure 6, BatchMaker includes a manager that “processes arriving requests and submits batched computation tasks to workers for execution.” Gao at p. 5. The manager in BatchMaker has two components, a “request processor” for tracking the progress of execution for the requests, and a scheduler for “determining which cells from different requests would form a batched task, and selects a worker to execute the task.” Gao at p. 5.

147. As also discussed above, Gao shows the behavior of cellular batching when being processed in Figure 5. Gao at p. 6; Gao at Fig. 5.

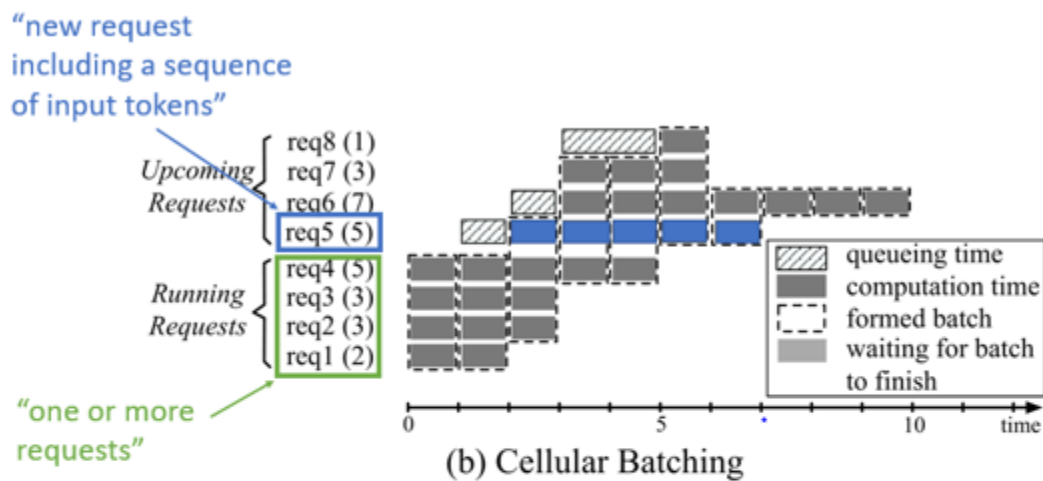


Figure 5: The timeline of graph batching and Cellular Batching when processing 8 requests from req1 to req8. The number shown in parenthesis is the request’s sequence length, e.g. req1(2) means req1 has a sequence length of 2. Each row marks the lifetime of a request starting from its arrival time. Req1-4 are *Running Requests* as they arrive at time 0 and have started execution. Req5-8 are *Upcoming Requests* that arrive after the Req1-4.

148. Gao describes the difference as follows:

In the beginning of time ($t=0$), the first 4 requests (req1-4) arrive... Under cellular batching, among the first 4 requests, the system forms two fully batched tasks, each performing the execution of a single (4way batched) RNN cell. At $t=2$, the second task finishes, causing req1 to complete and leave the system. Since a new request (req5) has already arrived, the system forms its third fully batched task containing req2-5 at $t=2$. After finishing this task, another two existing requests (req2, req3) depart and two new ones are added (req6, req7) to form the fourth task.

Gao at p. 6. (emphasis added).

149. As can be seen in the above, a new request (e.g., request 5) includes five cells, which is a different length than requests 1, 2, and 3.

150. Gao also provides that requests include a sequence of input tokens (e.g., $X(1), X(2), X(r)$) to predict an output token (e.g., “the most likely next word”):

“in its simplest form, we can view RNNs as operating on an input sequence, $X = [x(1), x(2), \dots, x(r)]$, where $x(i)$ represents the input at the i -th position (or timestep). For language processing, the input X would be a sentence, and $x(i)$ would be the vector embedding of the i -th word in the sentence... For example, we can use an RNN to process the input sentence “system research is” and to derive the most likely next word from the RNN’s output. Figure 1 shows the unfolded dataflow graph for this input.” Gao at p. 2.

151. Similarly, I note that a POSITA would have understood that input requests include input “tokens” and replies include “output tokens.” For example, the seminal 2017 article, “Attention Is All You Need,” introduced transformer models and disclosed input requests include “input tokens.” See Vaswani, 5 (3.4 Embeddings and Softmax) (“Similarly to other sequence transduction models, we use learned embeddings to convert the input tokens and output tokens to vectors of

dimension d_{model} .”); 6 (5.1 Training Data and Batching) (“Sentences were encoded using byte-pair encoding [3], which has a shared source-target vocabulary of about 37000 tokens.”).

1[E] scheduling, by the scheduler, a second batch of requests additionally including the new request for execution on the execution engine, the second batch of requests scheduled responsive to determining that the execution engine has memory available to execute the second batch of requests, wherein in a second set of inputs for the second batch of requests, a length of the sequence of input tokens for the new request is different from a length of an input for at least one request other than the new request; and

152. As discussed above, BatchMaker proposes a system architecture of a “Manager and Worker. The manager processes arriving requests and submits batched computation tasks to workers for execution.” Gao at p. 5. A copy of the system architecture for BatchMaker is as follows:

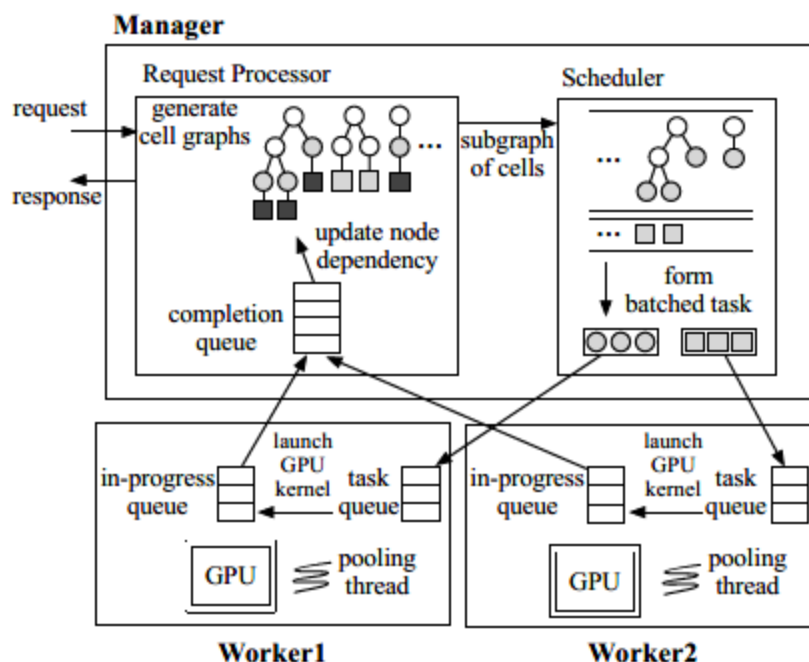


Figure 6: The system architecture of BatchMaker. In the cell graph, black means computed nodes, grey means nodes whose input is ready, and white means input dependency is not satisfied.

153. As can be seen from Figure 6, BatchMaker includes a manager that “processes arriving requests and submits batched computation tasks to workers for execution.” Gao at p. 5. The manager in BatchMaker has two components, a “request processor” for tracking the progress of execution for the requests, and a scheduler for “determining which cells from different requests would form a batched task, and selects a worker to execute the task.” Gao at p. 5.

154. Gao provides that “[t]he scheduler needs to make decisions on what nodes should be batched together to form a task and what tasks to be pushed to which workers.” Gao at p. 6. Gao goes on to provide detail for the aspects the scheduler must account for, namely “locality, priority, and utilization of multiple GPUs, which

are often in conflict with each other.” Gao at 6. “Locality refers to the preference that 1) the same set of requests should be batched together if they are to execute the same sequence of nodes, and the execution of the sequence of nodes should stick to the same GPU.” Gao at p. 6. “Priority refers to the ability to prefer the execution of one type of cell over another.” Gao at p. 6.

155. Gao also provides the basic algorithm employed by BatchMaker for scheduling and creation of batches. Gao at p. 6.

Algorithm 1: Scheduling and Batching Algorithm

```
1 Bsizes: a set of supported batch sizes.
2 CellTypes: a set of cell types, each associated with a priority.
3 MaxTasksToSubmit: the maximum number of tasks that can
  be submitted to a worker.
4 def Schedule(worker):
5   S ← {ct ∈ CellTypes | ct.NumReadyNodes() ≥
     Bsizes.Max()};
6   if S.Size() = 0 :
7     S ← {ct ∈ CellTypes | ct.NumRunningTasks() = 0 and
       ct.NumReadyNodes() > 0};
8   if S.Size() = 0 :
9     S ← {ct ∈ CellTypes | ct.NumReadyNodes() > 0};
10  ct ← GetCellTypeWithMaxPriority(S);
11  Batch(ct, worker);
12 def Batch(ct, worker):
13  num_tasks ← 0;
14  while num_tasks < MaxTasksToSubmit :
15    batch ← FormBatchedTask(ct, worker);
16    if batch.Size() ≥ Bsizes.Min() or num_tasks = 0 :
17      SubmitBatchedTask(batch, worker);
18      UpdateNodesDependency(batch);
19      num_tasks++;
20      for subgraph ∈ batch :
21        subgraph.pinned ← worker.id;
22        # pinned is unset once subgraph has no
23        task running
24    else:
25      break;
26 def FormBatchedTask(ct, worker):
27  batch ← {};
28  for subgraph ∈ ct.subgraphs :
29    if subgraph.pinned ∈ {None, worker.id} :
30      for node ∈ subgraph's ready nodes :
31        batch ← batch ∪ {node};
32        if batch.Size() = Bsizes.Max() :
33          return batch;
34  return batch;
```

156. As also discussed above, Gao shows the behavior of cellular batching when being processed in Figure 5. Gao at p. 6; Gao at Fig. 5.

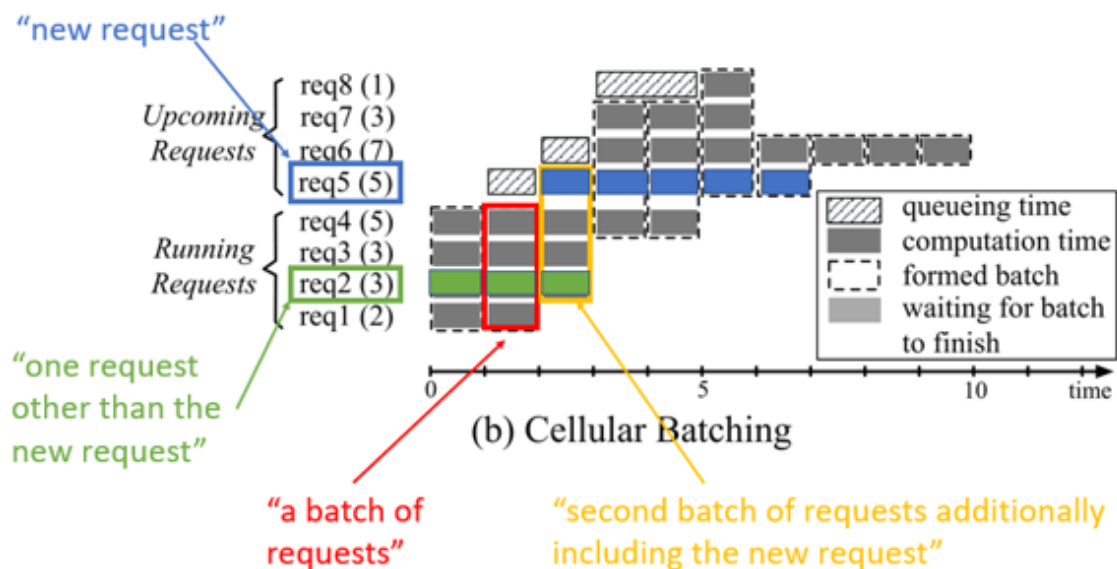


Figure 5: The timeline of graph batching and Cellular Batching when processing 8 requests from req1 to req8. The number shown in parenthesis is the request's sequence length, e.g. req1(2) means req1 has a sequence length of 2. Each row marks the lifetime of a request starting from its arrival time. Req1-4 are *Running Requests* as they arrive at time 0 and have started execution. Req5-8 are *Upcoming Requests* that arrive after the Req1-4.

157. Gao describes the process as follows:

In the beginning of time ($t=0$), the first 4 requests (req1-4) arrive... Under cellular batching, among the first 4 requests, the system forms two fully batched tasks, each performing the execution of a single (4way batched) RNN cell. At $t=2$, the second task finishes, causing req1 to complete and leave the system. Since a new request (req5) has already arrived, the system forms its third fully batched task containing req2-5 at $t=2$. After finishing this task, another two existing requests (req2, req3) depart and two new ones are added (req6, req7) to form the fourth task.

Gao at p. 6. (emphasis added).

158. When reviewing the above, Gao shows that as new requests are received, they are scheduled and moved to the execution engine. Herein, we note

that when executing a request in an execution engine it would be obvious to a POSITA to first ensure that memory is available before execution. Additionally, the second batch of requests (e.g., $t=3$) includes the new request for execution on the execution engine (e.g., request 5 with a sequence length of 5). Accordingly, the above illustrates that the new request and associated batch have a length that is different from the length of other requests received by BatchMaker.

159. Similarly, Gao makes clear that the system is configured to account for requests, of varying lengths, are processed efficiently – regardless of their arrival time:

Cellular batching is highly flexible, as the set of batched cells may come from requests arriving at different times or even from the same request. As a result, a newly arrived request can immediately join the ongoing execution of existing requests, without needing to waiting for them to finish. Long requests also do not decrease the amount of batching when they are batched together with short ones: each request can return to the user as soon as its last cell finishes and a long request effectively hitches a ride with multiple short requests over its execution lifetime.

Gao at p. 2.

160. Additionally, Gao teaches that batches have allocated memory that depends on the batch size. Gao at p. 8. Specifically, Gao teaches that “BatchMaker aims to materialize all the cells for each supported batch size on every available GPU device. Such materialization requires knowledge of the type and shape information of each operator’s input/output tensors in order to allocate GPU memory and perform other compiler-level optimizations.” Gao at p. 8.

161. Since batches have allocated memory depending on their size, the number of requests in a batch is a direct reflection of allocated memory availability. Moreover, Gao teaches a scheduler that responds to determining the availability of locations in a batch to decide to add more requests to a batch:

- “for node $\in$ subgraph’s ready nodes: ” (Gao Algorithm 1, Line 28)
- “ batch $\leftarrow$ batch $\cup$ {node}; ” (Gao Algorithm 1, Line 39)
- “ if batch.Size() = Bsizes.Max(): ” (Gao Algorithm 1, Line 30)
- “ return batch; ” (Gao Algorithm 1, Line 31)

162. Since a batch has allocated GPU memory, determining the availability of locations in a batch is reflective of memory availability, and since Gao teaches that requests correspond to a subgraph made of cells (nodes), as such, the scheduler taught by Gao is responsive to determining that the execution engine has memory availability to execute a request (whether new or old).

1[F] generating, by the execution engine, a second set of output tokens by applying the transformer model to the second set of inputs for the second batch.

163. As discussed above, Gao discloses an execution engine, namely workers, that use GPUs to “move[] a task from the task queue to the in-process queue once the task’s corresponding GPU kernel has been issued.” Gao at p. 6. Using a pooling mechanism, the worker continues to move “finished tasks from the in-process queue and pushes it into the completion queue in the request processor.” Gao at p. 6; *see also* Gao at Fig. 6.

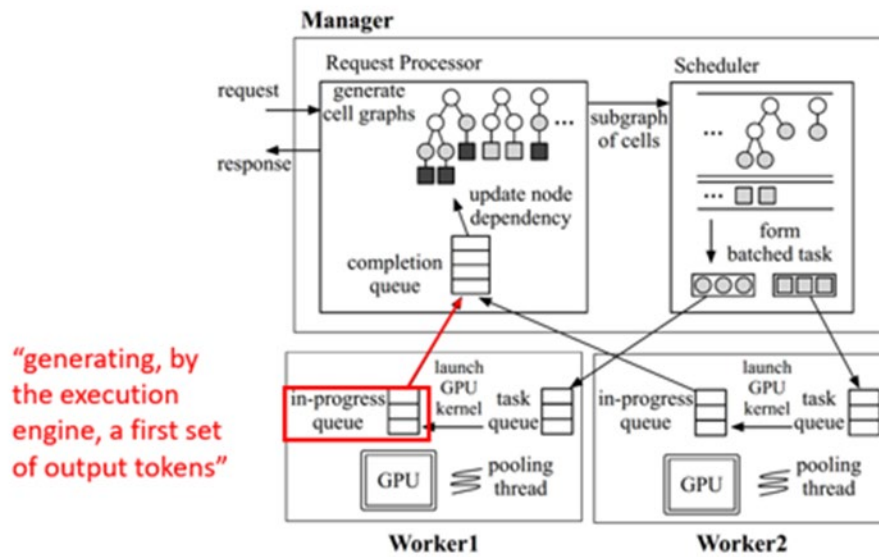
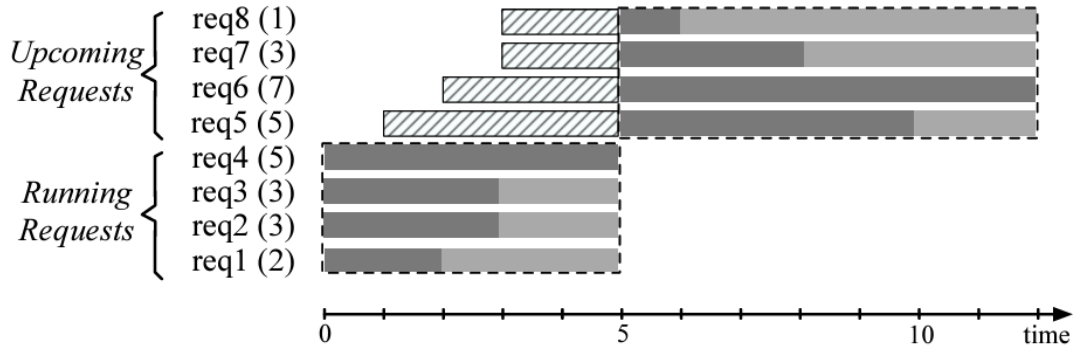
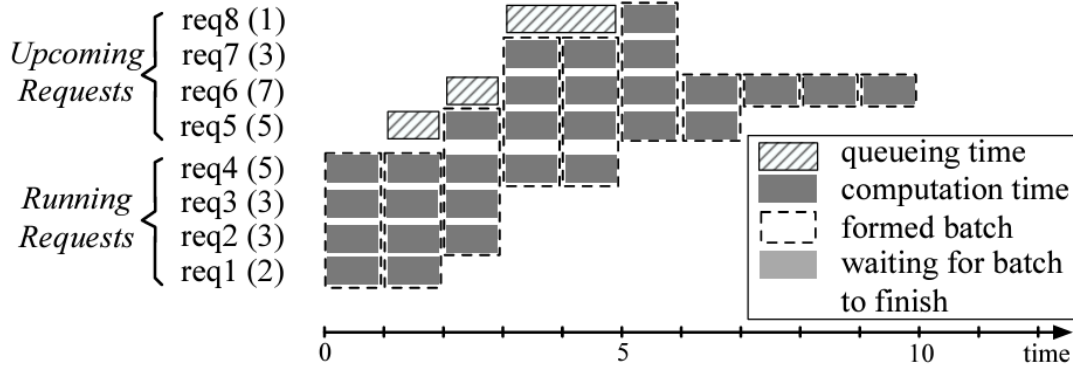


Figure 6: The system architecture of BatchMaker. In the cell graph, black means computed nodes, grey means nodes whose input is ready, and white means input dependency is not satisfied.

164. The behavior of cellular batching when being processed is illustrative
 Figure 5. Gao at p. 6; Gao at Fig. 5.



(a) Graph Batching



(b) Cellular Batching

Figure 5: The timeline of graph batching and Cellular Batching when processing 8 requests from req1 to req8. The number shown in parenthesis is the request's sequence length, e.g. req1(2) means req1 has a sequence length of 2. Each row marks the lifetime of a request starting from its arrival time. Req1-4 are *Running Requests* as they arrive at time 0 and have started execution. Req5-8 are *Upcoming Requests* that arrive after the Req1-4.

165. Gao describes the process as follows:

In the beginning of time ($t=0$), the first 4 requests (req1-4) arrive... Under cellular batching, among the first 4 requests, the system forms two fully batched tasks, each performing the execution of a single (4way batched) RNN cell. At $t=2$, the second task finishes, causing req1 to complete and leave the system. Since a new request (req5) has

already arrived, the system forms its third fully batched task containing req2-5 at $t=2$. After finishing this task, another two existing requests (req2, req3) depart and two new ones are added (req6, req7) to form the fourth task.

Gao at p. 6. (emphasis added).

166. To demonstrate the performance of BatchMaker, Gao provides an example of latency and performance improvements using Sequence to Sequence (“Seq2Seq”), a widely used RNN model for machine translation.

167. Gao provides, that

A basic Seq2Seq model contains two types of RNN cells: encoder and decoder, as depicted in Figure 12. The encoder takes in a sequence of words as input. In each step, the encoder and decoder will convert a word to a vector by doing an embedding lookup, then feed this vector to an RNN Cell. The first decoder cell takes in the output state of the encoder and a $\langle \text{go} \rangle$ symbol as input and computes states, which are passed to the succeeding decoder cell. In addition to the state, the decoder cell outputs a word as well, which is obtained by applying a linear transformation and an argmax. The output word is also fed to the next step as the input. When the decoder outputs the $\langle \text{eos} \rangle$ symbol, it means the decoder has finished, and there will be any more decoder steps.”

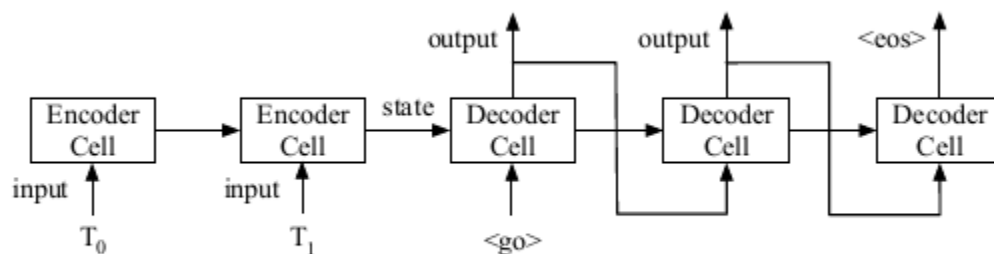


Figure 12: Seq2Seq with “feed previous” decoder

Gao at Fig. 12.

168. Moreover, Gao also provides that requests include a sequence of input tokens (e.g., $X(1), X(2), X(r)$) to predict an output token (e.g., “the most likely next word”):

“in its simplest form, we can view RNNs as operating on an input sequence, $X = [x(1), x(2), \dots, x(r)]$, where $x(i)$ represents the input at the i -th position (or timestep). For language processing, the input X would be a sentence, and $x(i)$ would be the vector embedding of the i -th word in the sentence... For example, we can use an RNN to process the input sentence “system research is” and to derive the most likely next word from the RNN’s output. Figure 1 shows the unfolded dataflow graph for this input.” Gao at p. 2.

169. Similarly, I note that a POSITA would have understood that input requests include input “tokens” and replies include “output tokens.” For example, the seminal 2017 article, “Attention Is All You Need,” introduced transformer models and disclosed input requests include “input tokens.” *See* Vaswani, 5 (3.4 Embeddings and Softmax) (“Similarly to other sequence transduction models, we use learned embeddings to convert the input tokens and output tokens to vectors of dimension d_{model} .”); 6 (5.1 Training Data and Batching) (“Sentences were encoded using byte-pair encoding [3], which has a shared source-target vocabulary of about 37000 tokens.”).

170. Using the BatchMaker technique, experiments show 17.5-90.5% latency improvement and 25-60% throughput performance. Gao at 2.

171. Katharopoulos introduces a machine-learning transformer model representing the transformer model as a Recurrent Neural Network (RNN). *See e.g.,*

Katharopoulos at p. 5 (discussing that “from the causal masking formulation in 3.3 and the discussion in the previous section, it becomes evident that any transformer layer with causal masking can be written as a model that, given an input, modifies an internal state then predicts an output, namely a Recurrent Neural Network (RNN).”)

172. Katharopoulos formalizes the transformer layer for transformers as an RNN in the following equation where x_i denotes the i -th input and y_i , the i -th output for a specific transformer layer:

$$s_0 = 0, \quad (16)$$

$$z_0 = 0, \quad (17)$$

$$s_i = s_{i-1} + \phi(x_i W_K)(x_i W_V)^T, \quad (18)$$

$$z_i = z_{i-1} + \phi(x_i W_K), \quad (19)$$

$$y_i = f_l \left(\frac{\phi(x_i W_Q)^T s_i}{\phi(x_i W_Q)^T z_i} + x_i \right). \quad (20)$$

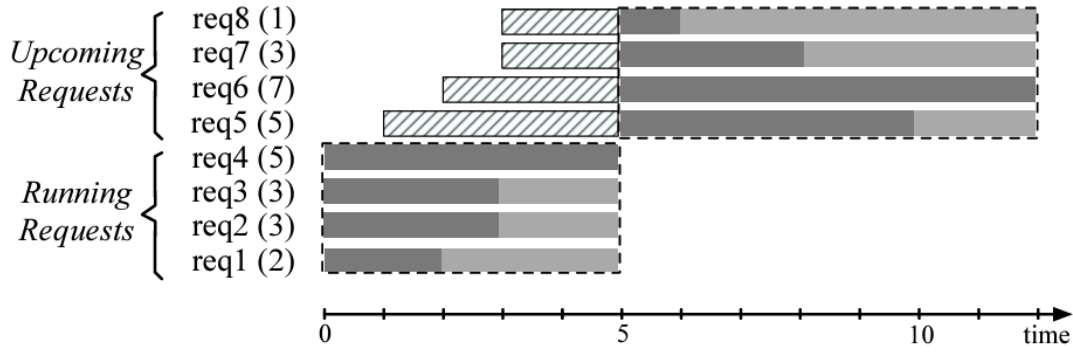
173. Katharopoulos at p. 5. Importantly, Katharopoulos recognized that the “formulation does not impose any constrain on the feature function, and it can be used for representing *any transformer* model, in theory even the ones using softmax attention.” Katharopoulos at p. 5. *See also* Summary of the Combination and Motivation to Combine *Supra* (discussing the application of cellular batching to DOT with a linearized attention function).

174. Accordingly, Katharopoulos teaches that a first set of output tokens (e.g., “ y_i , the i -th output”) by applying the transformer model (e.g., equation 20) to a first set of inputs (e.g., “ x_i denotes the i -th input”). Gao teaches applying an RNN cell (e.g. equation 20 of Katharopoulos) for a batch of requests, wherein applying the RNN cell for transformer model would comprise at least one batch operation, to one or more input tensors associated with the batch of requests. Furthermore, the batched output generated can be used to generate a set of output tokens. *See also* Summary of the Combination and Motivation to Combine *Supra*.

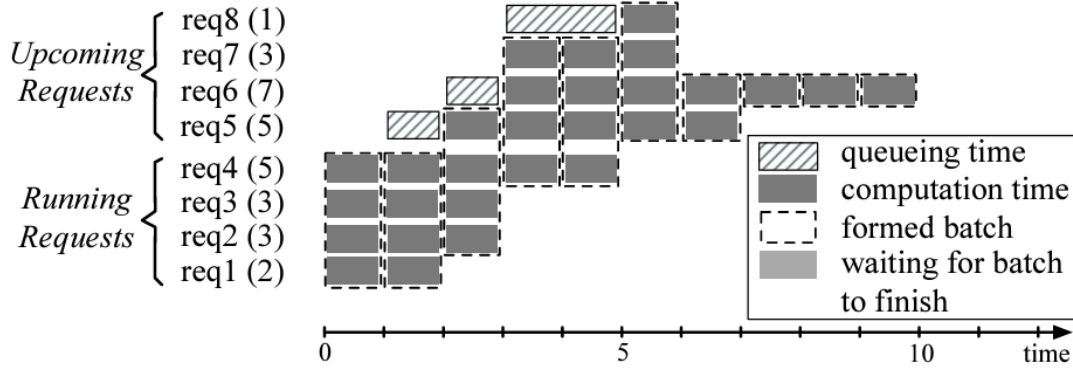
3. Claim 2. The method of claim 1, further comprising:

2[A] responsive to determining that a request in the first batch of requests has been completed, providing output tokens generated for the completed request to a client device as a response to the request, and

175. Gao describes the behavior of cellular batching when being processed as shown in Figure 5. Gao at p. 6; Gao at Fig. 5.



(a) Graph Batching



(b) Cellular Batching

Figure 5: The timeline of graph batching and Cellular Batching when processing 8 requests from req1 to req8. The number shown in parenthesis is the request's sequence length, e.g. req1(2) means req1 has a sequence length of 2. Each row marks the lifetime of a request starting from its arrival time. Req1-4 are *Running Requests* as they arrive at time 0 and have started execution. Req5-8 are *Upcoming Requests* that arrive after the Req1-4.

176. Gao describes the process as follows:

In the beginning of time ($t=0$), the first 4 requests (req1-4) arrive... Under cellular batching, among the first 4 requests, the system forms two fully batched tasks, each performing the execution of a single (4way batched) RNN cell. At $t=2$, the second task finishes, causing req1 to complete and leave the system. Since a new request (req5) has

already arrived, the system forms its third fully batched task containing req2-5 at $t=2$. After finishing this task, another two existing requests (req2, req3) depart and two new ones are added (req6, req7) to form the fourth task.

Gao at p. 6. (emphasis added).

177. When reviewing the above, Gao shows that when a task finishes with a request completing execution (e.g., $t=2$) it causes the completed request (e.g., request 1) to leave the system. Accordingly, the above illustrates that a completed request within a batch can leave the batch.

178. Similarly, Gao makes clear that the system is configured to account for finished requests to be returned to the user:

Cellular batching is highly flexible, as the set of batched cells may come from requests arriving at different times or even from the same request. As a result, a newly arrived request can immediately join the ongoing execution of existing requests, without needing to waiting for them to finish. Long requests also do not decrease the amount of batching when they are batched together with short ones: *each request can return to the user as soon as its last cell finishes* and a long request effectively hitches a ride with multiple short requests over its execution lifetime.

Gao p. 2. (emphasis added)

179. As discussed above, BatchMaker proposes a system architecture of a “Manager and Worker. The manager processes arriving requests and submits batched computation tasks to workers for execution.” Gao at p. 5. A copy of the system architecture for BatchMaker is as follows and shows a “response” returning to a user:

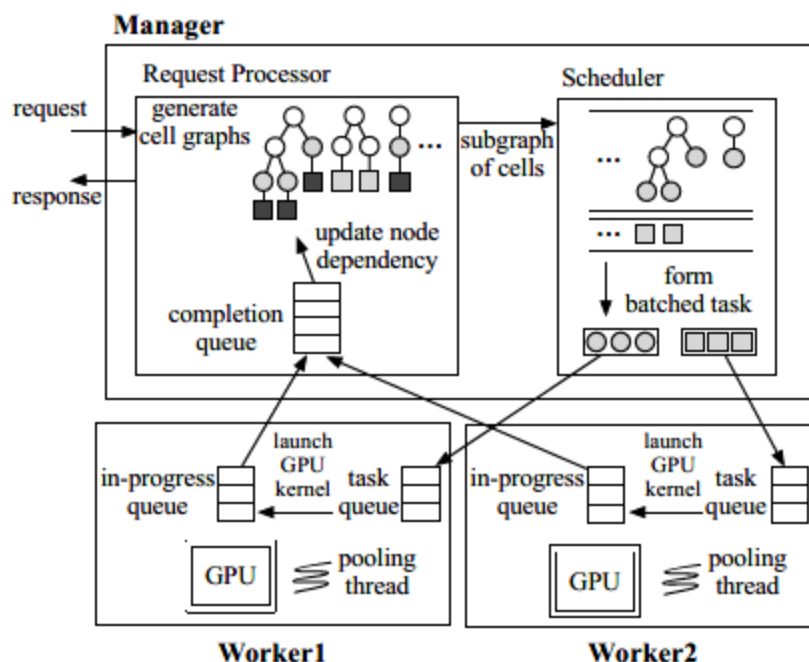


Figure 6: The system architecture of BatchMaker. In the cell graph, black means computed nodes, grey means nodes whose input is ready, and white means input dependency is not satisfied.

180. Moreover, and as discussed above, Gao also provides that requests include a sequence of input tokens (e.g., $X(1)$, $X(2)$, $X(r)$) to predict an output token (e.g., “the most likely next word”):

“in its simplest form, we can view RNNs as operating on an input sequence, $X = [x(1), x(2), \dots, x(r)]$, where $x(i)$ represents the input at the i -th position (or timestep). For language processing, the input X would be a sentence, and $x(i)$ would be the vector embedding of the i -th word in the sentence... For example, we can use an RNN to process the input sentence “system research is” and to derive the most likely next word from the RNN’s output. Figure 1 shows the unfolded dataflow graph for this input.” Gao at p. 2.

181. Similarly, I note that a POSITA would have understood that input requests include input “tokens” and replies include “output tokens.” For example,

the seminal 2017 article, “Attention Is All You Need,” introduced transformer models and disclosed input requests include “input tokens.” See Vaswani, 5 (3.4 Embeddings and Softmax) (“Similarly to other sequence transduction models, we use learned embeddings to convert the input tokens and output tokens to vectors of dimension d_{model} .”); 6 (5.1 Training Data and Batching) (“Sentences were encoded using byte-pair encoding [3], which has a shared source-target vocabulary of about 37000 tokens.”).

2[B] wherein the second batch of requests includes at least one of the remaining requests from the one or more requests and the new request.

182. As also discussed above, Gao shows the behavior of cellular batching when being processed in Figure 5. Gao at p. 6; Gao at Fig. 5.

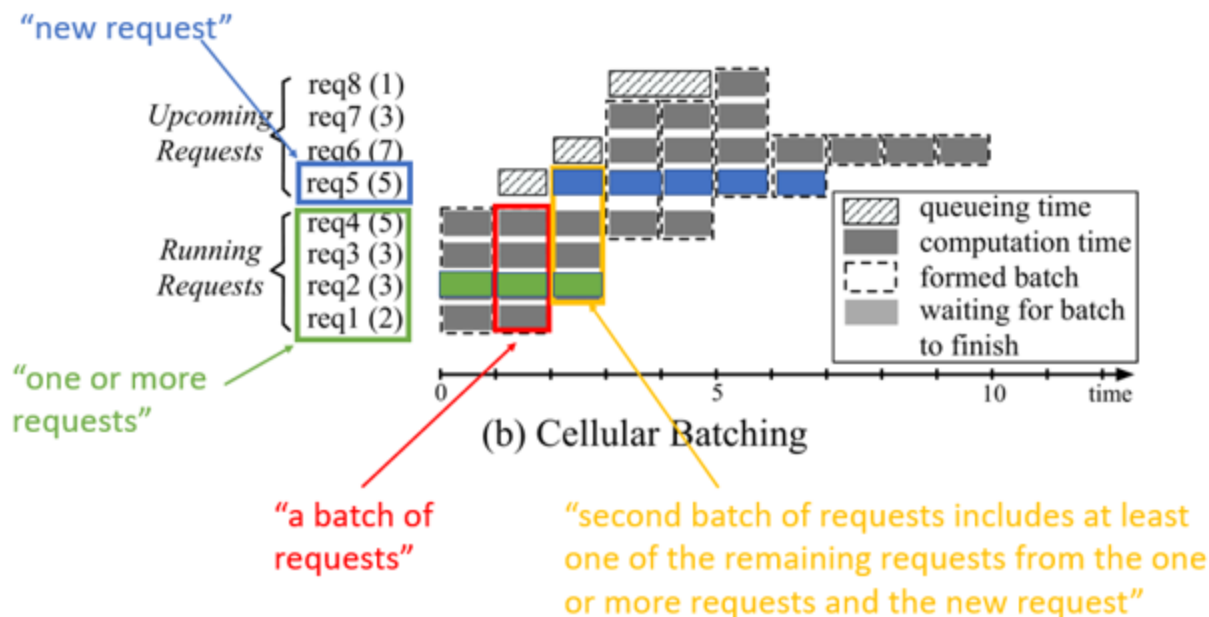


Figure 5: The timeline of graph batching and Cellular Batching when processing 8 requests from req1 to req8. The number shown in parenthesis is the request's sequence length, e.g. req1(2) means req1 has a sequence length of 2. Each row marks the lifetime of a request starting from its arrival time. Req1-4 are *Running Requests* as they arrive at time 0 and have started execution. Req5-8 are *Upcoming Requests* that arrive after the Req1-4.

183. Gao describes the process as follows:

In the beginning of time ($t=0$), the first 4 requests (req1-4) arrive... Under cellular batching, among the first 4 requests, the system forms two fully batched tasks, each performing the execution of a single (4way batched) RNN cell. At $t=2$, the second task finishes, causing req1 to complete and leave the system. Since a new request (req5) has already arrived, the system forms its third fully batched task containing req2-5 at $t=2$. After finishing this task, another two existing requests (req2, req3) depart and two new ones are added (req6, req7) to form the fourth task.

Gao at p. 6. (emphasis added).

184. When reviewing the above, Gao shows that when a task finishes with a request completing execution (e.g., $t=2$) it causes a new request (e.g., request 5) to

join the existing batch (e.g., $t=3$). Thus, Gao makes clear that a new request can join an existing batch which will contain the remaining requests and the new arriving request.

4. **Claim 3. The method of claim 2, wherein the request is associated with a cache memory in the execution engine dedicated for storing an internal state for the request, and responsive to determining that the request has been completed, freeing the dedicated cache memory for the request in the execution engine.**

185. Gao discloses a request processor that generates or builds a computational graph of dependent nodes or cells for the request, where the dependency between cell graphs (or subgraphs) corresponds to data dependency in the form of an internal state of the previous cell. For example, Gao discloses a request processor, the behavior of which in response to a new request is described as follows:

When a new request arrives, the request processor runs the user-code to unfold the recursion and generates the corresponding cell graph for the request. In this cell graph, each node represents a cell and is labeled with a unique node id as well as its cell type. Request processor will track and update the dependencies of each node. When a node's dependencies have been satisfied (aka its inputs are ready), the node is ready to be scheduled for execution (§4.3).

Gao at p. 5. (emphasis added)

186. When reviewing the above, Gao discloses a request is turned into a computational graph made of dependent nodes where each node is a cell. The

Request Processor, which retains information relating to each request in the form of a cell (node) graph where each node observes a list of dependencies and data that corresponds to the internal state of the previous node in the cell graph. This process is further disclosed in BatchMaker's scheduling and batching algorithm.

187. Moreover, the cell graph of a request is searched to find a sub-graph made of a number of connected cells such that all external dependencies, to other parts of the graph, are satisfied and that all cells within a subgraph are of the same type. *See* Gao at p. 6 (describing that “the request processor analyzes the cell graph of a request to find a subgraph to pass to the scheduler,” and that “[a] subgraph contains a single node or a number of connected nodes with the property that all external dependencies to other parts of the graph have been satisfied.”)

188. Gao provides the basic algorithm employed by BatchMaker for scheduling and creation of batches. Gao at p. 6.

Algorithm 1: Scheduling and Batching Algorithm

```
1 Bsizes: a set of supported batch sizes.
2 CellTypes: a set of cell types, each associated with a priority.
3 MaxTasksToSubmit: the maximum number of tasks that can
  be submitted to a worker.
4 def Schedule(worker):
5   S ← {ct ∈ CellTypes | ct.NumReadyNodes() ≥
     Bsizes.Max()};
6   if S.Size() = 0 :
7     S ← {ct ∈ CellTypes | ct.NumRunningTasks() = 0 and
       ct.NumReadyNodes() > 0};
8   if S.Size() = 0 :
9     S ← {ct ∈ CellTypes | ct.NumReadyNodes() > 0};
10  ct ← GetCellTypeWithMaxPriority(S);
11  Batch(ct, worker);
12 def Batch(ct, worker):
13  num_tasks ← 0;
14  while num_tasks < MaxTasksToSubmit :
15    batch ← FormBatchedTask(ct, worker);
16    if batch.Size() ≥ Bsizes.Min() or num_tasks = 0 :
17      SubmitBatchedTask(batch, worker);
18      UpdateNodesDependency(batch);
19      num_tasks++;
20      for subgraph ∈ batch :
21        subgraph.pinned ← worker.id;
22        # pinned is unset once subgraph has no
23        task running
24    else:
25      break;
26 def FormBatchedTask(ct, worker):
27  batch ← {};
28  for subgraph ∈ ct.subgraphs :
29    if subgraph.pinned ∈ {None, worker.id} :
30      for node ∈ subgraph's ready nodes :
31        batch ← batch ∪ {node};
32        if batch.Size() = Bsizes.Max() :
33          return batch;
34  return batch;
```

189. Gao explains the process of updating the internal dependency:

Once a batched task is submitted to the GPU worker, the scheduler updates the dependencies of those nodes in the batch (line 18) so a new set of cell nodes can be scheduled in subsequent batched tasks. Additionally, the scheduler pins those subgraphs to the same worker (line 20-21) to avoid scheduling the subsequent nodes in those subgraphs to a different worker (line 27). This is crucial to the correctness of the scheduling algorithm because tasks involving the same subgraph have data dependency.

Gao at p. 7. (emphasis added).

190. When reviewing the above, Gao details that the dependency between cell graphs (or subgraphs) corresponds to data dependency in the form of an internal state of the previous cell. As each request corresponds to a cell graph, the internal state continuously updated is associated with a specific request. Gao further provides for a request node graph to be pinned to the same worker to ensure the correctness of execution due to the data dependency stored within the worker. As such, Gao provides for a request's internal state to be stored within a worker.

191. Further, "Since the batched outputs of the execution are also stored in contiguous memory, there is no need for memory copy before each individual cell execution when executing the same set of requests on the same GPU." Gao at p. 6. Thus, we can conclude that Gao provides for a request internal state to be stored in the GPU's memory for further usage.

192. Gao also demonstrates that a subgraph corresponding to a request, and pinned to a worker to preserve the data-dependency between nodes, is made of cells of the same type which are RNN cells. Further, the data dependency between the cells of a subgraph corresponds to the internal cell state (or hidden unit) of the RNN which is also an output of the RNN cell:

In its simplest form, we can view RNNs as operating on an input sequence, $X = [x(1), x(2), \dots, x(\tau)]$, where $x(i)$ represents the input at the i -th position (or timestep). For language processing, the input X would be a sentence, and $x(i)$ would be the vector embedding of the i -th word

in the sentence. RNN's key advantage comes from parameter sharing when processing different positions. Specifically, *let f_θ be a function parameterized with θ , RNNs represent the recursive computation $h(t) = f_\theta(h(t-1), x(t))$, where $h(t)$ is viewed as the value of the hidden unit after processing the input sequence up to the t -th position. The function f_θ is commonly referred to as an RNN cell. An RNN cell can be as simple as a fully connected layer with an activation function, or the more sophisticated Long Short-Term Memory (LSTM) cell. The LSTM cell [24] contains internal cell state that store information and uses several gates to control what goes in or out of those cell state and whether to erase the stored information."*

Gao p. 2. (emphasis added)

193. Gao also teaches that the output of a cell, which can include the hidden internal state of the cell corresponding to the data dependency between nodes, is stored in the contiguous GPU memory. *See Gao at p. 6 (teaching that "the batched inputs of a cell must be laid out in contiguous GPU memory. Since the batched outputs of the execution are also stored in contiguous memory, there is no need for memory copy before each individual cell execution when executing the same set of requests on the same GPU.")*

194. Since batched requests have subgraphs pinned to a worker (executing on a GPU), and since the nodes in a request subgraph have outputs, which include the interdependent cell hidden state, that are stored in contiguous GPU memory, we can conclude that the batched requests cell-hidden-states (internal state) must be pinned to the GPU memory where it is stored to be used when the dependent cells

within the graph are executed. Gao also explains why not doing so is wasteful. Gao at p. 6.

195. According to Gao, and as detailed in Figure 5 (Gao at p. 5.) and Algorithm 1 (Gao at p. 6.), a subgraph of a request is executed one cell at a time as part of different batches to preserve the data dependency between cells.

“FormBatchedTask” function (line 24) scans the queue to select nodes whose dependencies are satisfied (line 28) for a batched task. Each invocation of “FormBatchedTask” forms at most one batched task. For better locality, the scheduler submits several tasks to the same worker for execution. The number of tasks submitted is limited by the configurable parameter “MaxTasksToSubmit” (line 14). Setting the limit to a small number (default is 5) avoids forming too many tasks belonging to one cell type, which gives other types of cell a chance to be scheduled, allows new requests to join execution, and avoids the decrease in effective batch size if one cell type does not have enough ready nodes.

Gao at p. 7

196. Batches, along with their input and output tensors, have allocated memory that depends on the batch size. *See* Gao at p. 8 (describing that “During initialization, BatchMaker aims to materialize all the cells for each supported batch size on every available GPU device. Such materialization requires knowledge of the type and shape information of each operator’s input/output tensors in order to allocate GPU memory and perform other compiler-level optimizations...”).

197. Thus, Gao allocates GPU cache memory in the execution engine for storing the hidden internal states for respective batched requests, where the amount

of GPU cache memory used for such storage must be allocated for each batch (depending on the batch size) during initializing.

198. Further, Gao also teaches that stored cell hidden state outputs corresponding to a batch of requests will correspond to a specific batch which will be updated when a task is completed. See Gao at p. 6 (showing that “[t]he request processor updates node dependencies based on the completed task, and checks whether a request is finished. If so, its result is immediately returned.”).

199. Finally, according to Gao, when the request is completed and there are no more nodes to execute for a request subgraph, Gao unpins the subgraph from a worker and thus frees the GPU cache memory allocated for storing the cell hidden internal states. Gao at p. 7.

5. **Claim 4. The method of claim 1, wherein the input for the at least one request is an output token from the first set of output tokens for the at least one request, and wherein a length of the sequence of input tokens for the new request is different from a length of the output token for the at least one request.**

200. As discussed above, Gao discloses an execution engine, namely workers, that use GPUs to “move[]a task from the task queue to the in-process queue once the task’s corresponding GPU kernel has been issued.” Gao at p. 6. Using a pooling mechanism, the worker continues to move “finished tasks from the in-process queue and pushes it into the completion queue in the request processor.” Gao at p. 6.

201. The behavior of cellular batching when being processed is illustrative

Figure 5. Gao at p. 6; Gao at Fig. 5.

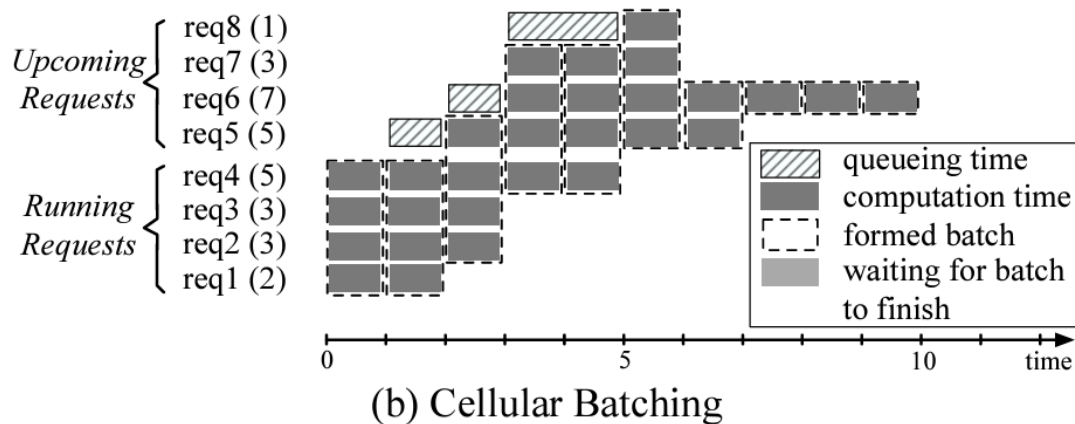
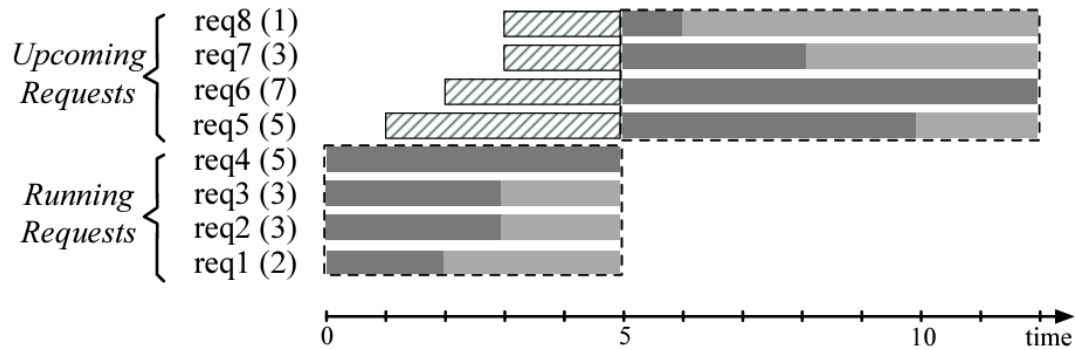


Figure 5: The timeline of graph batching and Cellular Batching when processing 8 requests from req1 to req8. The number shown in parenthesis is the request's sequence length, e.g. req1(2) means req1 has a sequence length of 2. Each row marks the lifetime of a request starting from its arrival time. Req1-4 are *Running Requests* as they arrive at time 0 and have started execution. Req5-8 are *Upcoming Requests* that arrive after the Req1-4.

202. Gao describes the difference as follows:

In the beginning of time ($t=0$), the first 4 requests (req1-4) arrive... Under cellular batching, among the first 4 requests, the system forms two fully batched tasks, each performing the execution of a single (4way batched) RNN cell. At $t=2$, the second task finishes, causing req1 to complete and leave the system. Since a new request (req5) has already arrived, the system forms its third fully batched task containing req2-5 at $t=2$. After finishing this task, another two existing requests (req2, req3) depart and two new ones are added (req6, req7) to form the fourth task.

Gao at p. 6. (emphasis added).

203. To demonstrate the performance of BatchMaker, Gao provides an example of latency and performance improvements using Sequence to Sequence (“Seq2Seq”), a widely used RNN model for machine translation.

204. Gao provides, that

A basic Seq2Seq model contains two types of RNN cells: encoder and decoder, as depicted in Figure 12. The encoder takes in a sequence of words as input. In each step, the encoder and decoder will convert a word to a vector by doing an embedding lookup, then feed this vector to an RNN Cell. The first decoder cell takes in the output state of the encoder and a < go > symbol as input and computes states, which are passed to the succeeding decoder cell. In addition to the state, the decoder cell outputs a word as well, which is obtained by applying a linear transformation and an argmax. The output word is also fed to the next step as the input. When the decoder outputs the < eos > symbol, it means the decoder has finished, and there will be any more decoder

steps.”

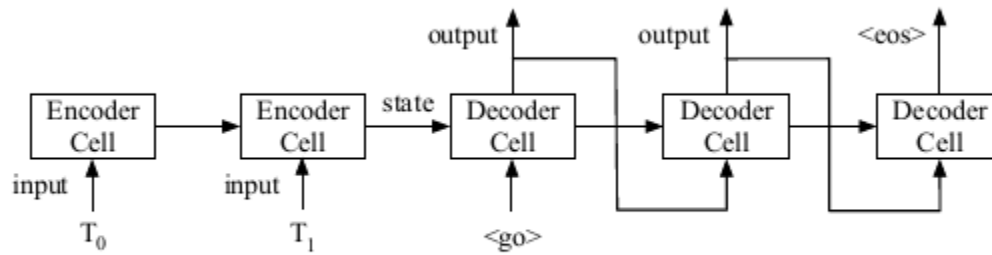


Figure 12: Seq2Seq with “feed previous” decoder

Gao at Fig. 12.

205. Using the BatchMaker technique, experiments show 17.5-90.5% latency improvement and 25-60% throughput performance. Gao at 2.

206. Accordingly, and as can be seen above, the input for the at least one request into the Decoder Cell is a token from the first set of output tokens for the at least one request. Moreover, because Gao teaches that the input tokens from requests can be different lengths, they are accordingly different from a length of “the output token” (singular) for the at least one request.

207. Moreover, Gao teaches that it was known for the output token length to be longer than the input token length. Gao at p. 11 (“When doing inference using Seq2Seq, the decoded sequence length is not known a priori. Deployed systems typically configure the maximum decoding length to be the input sequence length plus a fixed threshold of extra steps.”)

6. Claim 5.

5[A] The method of claim 4, wherein the execution engine includes a cache memory for maintaining a key cache tensor for storing keys and a value cache tensor for storing values for the at least one request, and

208. As I discuss above with respect to claim 3 and incorporate herein, Gao teaches the dependency between cell graphs (or subgraphs) corresponds to data dependency in the form of an internal state of the previous cell, and Gao teaches storing the internal state for a request in the GPU's cache memory. In other words, Gao caches information relating to each request in the form of a cell (node) graph where each node observes a list of dependencies and data that corresponds to the internal state of the previous node in the cell graph.

209. Gao further discloses a request processor the behavior of which in response to a new request is described as follows:

When a new request arrives, the request processor runs the user-code to unfold the recursion and generates the corresponding cell graph for the request. In this cell graph, each node represents a cell and is labeled with a unique node id as well as its cell type. Request processor will track and update the dependencies of each node. When a node's dependencies have been satisfied (aka its inputs are ready), the node is ready to be scheduled for execution (§4.3).

Gao at p. 5. (emphasis added)

210. When reviewing the above, Gao discloses a Request Processor, which retains information relating to each request in the form of a cell (node) graph where each node observes a list of dependencies and data that corresponds to the internal

state of the previous node in the cell graph. This process is further disclosed in BatchMaker's scheduling and batching algorithm.

211. Gao provides the basic algorithm employed by BatchMaker for scheduling and creation of batches. Gao at p. 6.

Algorithm 1: Scheduling and Batching Algorithm

```

1 Bsizes: a set of supported batch sizes.
2 CellTypes: a set of cell types, each associated with a priority.
3 MaxTasksToSubmit: the maximum number of tasks that can
  be submitted to a worker.
4 def Schedule(worker):
5   S ← {ct ∈ CellTypes | ct.NumReadyNodes() ≥
     Bsizes.Max()};
6   if S.Size() = 0:
7     S ← {ct ∈ CellTypes | ct.NumRunningTasks() = 0 and
        ct.NumReadyNodes() > 0};
8   if S.Size() = 0:
9     S ← {ct ∈ CellTypes | ct.NumReadyNodes() > 0};
10  ct ← GetCellTypeWithMaxPriority(S);
11  Batch(ct, worker);
12 def Batch(ct, worker):
13  num_tasks ← 0;
14  while num_tasks < MaxTasksToSubmit:
15    batch ← FormBatchedTask(ct, worker);
16    if batch.Size() ≥ Bsizes.Min() or num_tasks = 0:
17      SubmitBatchedTask(batch, worker);
18      UpdateNodesDependency(batch);
19      num_tasks++;
20      for subgraph ∈ batch:
21        subgraph.pinned ← worker.id;
22        # pinned is unset once subgraph has no
          task running
23    else:
24      break;
25 def FormBatchedTask(ct, worker):
26  batch ← {};
27  for subgraph ∈ ct.subgraphs:
28    if subgraph.pinned ∈ {None, worker.id}:
29      for node ∈ subgraph's ready nodes:
30        batch ← batch ∪ {node};
31        if batch.Size() = Bsizes.Max():
32          return batch;
33  return batch;

```

212. Gao explains the process of updating the internal dependency:

Once a batched task is submitted to the GPU worker, the scheduler updates the dependencies of those nodes in the batch (line 18) so a new set of cell nodes can be scheduled in subsequent batched tasks. Additionally, the scheduler pins those subgraphs to the same worker (line 20-21) to avoid scheduling the subsequent nodes in those subgraphs to a different worker (line 27). This is crucial to the correctness of the scheduling algorithm because tasks involving the same subgraph have data dependency.

Gao at p. 7. (emphasis added).

213. When reviewing the above, Gao details that the dependency between cell graphs (or subgraphs) corresponds to data dependency in the form of an internal state of the previous cell. As each request corresponds to a cell graph, the internal state continuously updated is associated with a specific request. Gao further provides for a request node graph to be pinned to the same worker to ensure correctness of execution due to the data dependency which would be stored within the worker. As such, Gao provides for a request's internal state to be stored within a worker. Further, "Since the batched outputs of the execution are also stored in contiguous memory, there is no need for memory copy before each individual cell execution when executing the same set of requests on the same GPU." Gao p. 6. Gao Thus, we can conclude that Gao teaches for a request internal state to be stored (cached) in the GPU's memory for further usage.

214. Katharopoulos further teaches the Transformer RNN internal state includes a Key (K_j) and Value (V_j) matrix or tensor, which provides a key cache tensor for storing keys and a value cache tensor for storing values for the at least one

request. For example, Katharopoulos explains it “simply store[s] the $\phi(K)V^T$ matrix as an internal state and update[s] it at every time step like a recurrent neural network. This results in inference thousands of times faster than other transformer models.” Katharopoulos at p. 4. Katharopoulos further teaches the Transformer RNN has two internal states, which includes attention memory and normalizer memory. Katharopoulos at p. 5 (“The resulting RNN has two hidden states, namely the attention memory s and the normalizer memory z . We use subscripts to denote the timestep in the recurrence.”). Katharopoulos further provides Equation 18 and Equation 19 to define the attention memory “ s ” and normalizer memory “ z .”

$$s_0 = 0, \quad (16)$$

$$z_0 = 0, \quad (17)$$

$$s_i = s_{i-1} + \phi(x_i W_K) (x_i W_V)^T, \quad (18)$$

$$z_i = z_{i-1} + \phi(x_i W_K), \quad (19)$$

$$y_i = f_l \left(\frac{\phi(x_i W_Q)^T s_i}{\phi(x_i W_Q)^T z_i} + x_i \right). \quad (20)$$

Katharopoulos at p. 5.

215. Regardless of the names used, a POSITA understands that the (s) internal state contains a computation of the Key and Value, whereas the (z) internal state only involves a computation of the Key. As such, Katharopoulos teaches a Transformer RNN with two internal states, one being a Value tensor (a matrix that includes value and key information), while the other is a Key tensor (a matrix that includes only key information).

216. Further, according to Gao:

“During initialization, BatchMaker aims to materialize all the cells for each supported batch size on every available GPU device. Such materialization requires knowledge of the type and shape information of each operator’s input/output tensors in order to allocate GPU memory and perform other compiler-level optimizations such as those done by NNVM [10], TensorFlow XLA [11].” (Gao p. 8.)

217. Gao’s memory cache, as discussed for claim 3, is used to store the cell internal state. Since the internal (hidden) state is an output of a cell operator, and since the memory allocated in a GPU is allocated to store the output tensors, a POSITA would understand that the memory cache for the Key is used to store Key tensors and that the memory cache for the Value (combined with Key) is used to store Value (combined with Key) tensors.

218. Thus, in light of Gao's teaching that a request's internal state is stored in cache memory for further usage, a POSITA would know that Katharopoulos’s Key and Value tensors (matrix) would be stored in the execution engine’s cache memory for the combined Gao and Katharopoulos system.

5[B] wherein after scheduling the second batch of requests, allocating, by the execution engine, a new cache memory dedicated to maintaining a key cache tensor and a value cache tensor for the new request.

219. According to Gao, different requests join different batches of different sizes. (Gao p. 6.). For example, batches can have:

- Different sizes: “Bsizes: a set of supported batch sizes.” Gao Algorithm 1, Line 1.
- A maximum size: “if batch.Size() = Bsizes.Max():” Gao Algorithm 1, Line 30.
- A minimum size: “if batch.Size() >= Bsizes.Min() or num_tasks = 0:” Gao Algorithm 1, Line 16.
- Different maximum sizes: “Each type of cell has a desired maximum batch size, which is determined through offline benchmarking.” Gao p. 5.
- A specific desirable size: “Once a task has reached a desired batch size, it is pushed into the task queue of one of the workers.” Gao p. 5. – p. 6.

220. Further, “During initialization, BatchMaker aims to materialize all the cells for each supported batch size on every available GPU device. Such materialization requires knowledge of the type and shape information of each operator’s input/output tensors in order to allocate GPU memory and perform other compiler-level optimizations such as those done by NNVM [10], TensorFlow XLA [11].” Gao p. 8.

221. As such, a new request as part of a new batch with a different batch size will have its own internal hidden state to be stored in a different contiguous GPU memory. Thus, a POSITA would understand that a new request can use a new

memory cache (different than the first) for Value and Key Tensors discussed previously.

7. **Claim 6. The method of claim 5, wherein after generating the second set of output tokens, a length of the key cache tensor for the at least one request is different from a length of a key cache tensor for the new request, and a length of the value cache tensor for the at least one request is different from a length of a value cache tensor for the new request.**

222. As discussed above, Gao provides a basic algorithm employed by BatchMaker for scheduling and creation of batches. Gao at p. 6.

Algorithm 1: Scheduling and Batching Algorithm

```
1 Bsizes: a set of supported batch sizes.
2 CellTypes: a set of cell types, each associated with a priority.
3 MaxTasksToSubmit: the maximum number of tasks that can
  be submitted to a worker.
4 def Schedule(worker):
5   S ← {ct ∈ CellTypes | ct.NumReadyNodes() ≥
     Bsizes.Max()};
6   if S.Size() = 0 :
7     S ← {ct ∈ CellTypes | ct.NumRunningTasks() = 0 and
       ct.NumReadyNodes() > 0};
8   if S.Size() = 0 :
9     S ← {ct ∈ CellTypes | ct.NumReadyNodes() > 0};
10  ct ← GetCellTypeWithMaxPriority(S);
11  Batch(ct, worker);
12 def Batch(ct, worker):
13  num_tasks ← 0;
14  while num_tasks < MaxTasksToSubmit :
15    batch ← FormBatchedTask(ct, worker);
16    if batch.Size() ≥ Bsizes.Min() or num_tasks = 0 :
17      SubmitBatchedTask(batch, worker);
18      UpdateNodesDependency(batch);
19      num_tasks++;
20      for subgraph ∈ batch :
21        subgraph.pinned ← worker.id;
22        # pinned is unset once subgraph has no
23        task running
24    else:
25      break;
26 def FormBatchedTask(ct, worker):
27  batch ← {};
28  for subgraph ∈ ct.subgraphs :
29    if subgraph.pinned ∈ {None, worker.id} :
30      for node ∈ subgraph's ready nodes :
31        batch ← batch ∪ {node};
32        if batch.Size() = Bsizes.Max() :
33          return batch;
34  return batch;
```

223. Specifically, Gao provides different batch sizes (“Bsizes: a set of supported batch sizes.” (Gao Algorithm 1, Line 1)); a maximum batch size (“if batch.Size() = Bsizes.Max():” (Gao Algorithm 1, Line 30)); and, a minimum batch size (“if batch.Size() ≥ Bsizes.Min() or num_tasks = 0:” (Gao Algorithm 1, Line 16)); *See also* Gao at p. 5 (discussing that “Each type of cell has a desired maximum batch size, which is determined through offline benchmarking.”); *See also* Gao at

pp. 5-6 (discussing that “Once a task has reached a desired batch size, it is pushed into the task queue of one of the workers.”)

224. Moreover, batches of different sizes, along with their input and output tensors, have allocated memory that depends on the batch size. Specifically, Gao provides that “*BatchMaker aims to materialize all the cells for each supported batch size on every available GPU device...*[which] requires knowledge of the type and shape information of each operator’s input/output tensors in order to allocate GPU memory and perform other compiler-level optimizations.” Gao at p. 8 (emphasis added).

225. As discussed above with respect to key cache tensor and value cache tensor, different batches of requests can have a different size memory. Since different requests can join different batches, the size (or length) of the cache memory for key cache tensor and value cache tensor discussed above can be different for different requests depending on which batches each request joins.

8. **Claim 7. The method of claim 1, after receiving the new request from the client device, determining, by the scheduler, that there is insufficient memory available to execute the second batch of requests on a second execution engine different from the execution engine, and responsive to the determination for the second execution engine, determining whether the execution engine has the memory available to execute the second batch of requests.**

226. Gao's BatchMaker contains multiple Workers and a Manager, where each Worker is associated with one GPU and when multiple GPUs are used, multiple Workers are used.

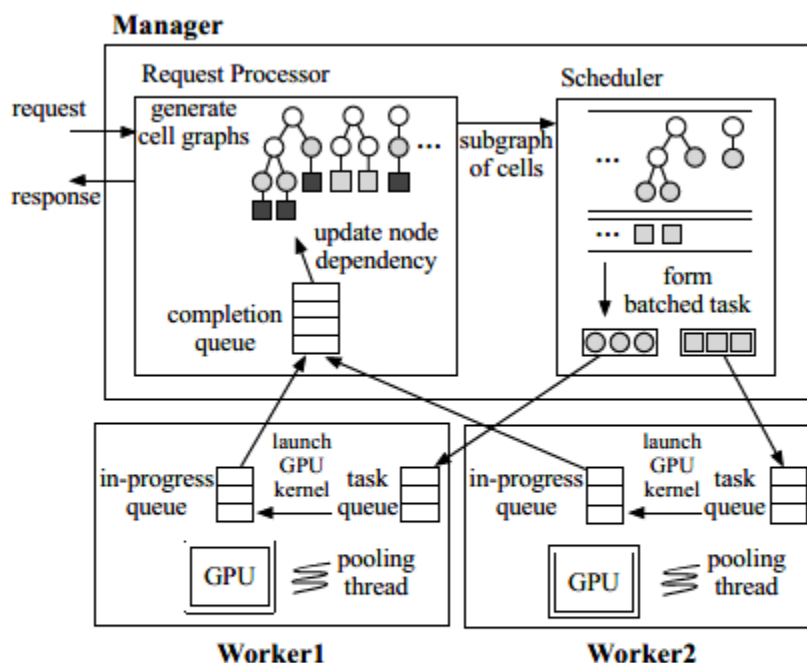


Figure 6: The system architecture of BatchMaker. In the cell graph, black means computed nodes, grey means nodes whose input is ready, and white means input dependency is not satisfied.

Gao at Fig. 6.

“BatchMaker has two main components: Manager and Worker. The manager processes arriving requests and submits batched computation tasks to workers for execution. *Depending on the number of GPU devices equipped, there may be multiple workers, each of which is associated with one GPU device. Workers execute tasks on GPUs and notify the manager when its tasks complete.*”

227. Moreover, the Manager, which “consists of two submodules, request processor and scheduler, invokes a worker whenever the worker becomes idle. *See*

Gao at p. 6 (providing “The scheduler’s “Schedule”” function (line 4) is invoked whenever some worker becomes idle...” As such, and in addition to the memory discussion in claim 1, a request subgraph node can be scheduled responsive to memory sufficiency (space within a batch).

228. A request subgraph node can be scheduled responsive to memory sufficiency (space within a batch). Thus, when multiple Workers are used, a new request can be executed through a first worker running on an execution engine responding to memory availability, otherwise, the new request will wait for a second worker to be invoked which will determine batching the new request also in response to memory availability. The new request can also be passed on by the second worker and the request would have to wait again for, potentially, the first (or another) worker which will also decide on batching the new request in response to memory availability.

9. Claim 8. The method of claim 1, wherein the execution engine is configured as a graphics processing unit (GPU) or a tensor processing unit (TPU).

229. As discussed above, BatchMaker proposes a system architecture of a “Manager and Worker. The manager processes arriving requests and submits batched computation tasks to workers for execution.” Gao at p. 5. A copy of the system architecture for BatchMaker is as follows:

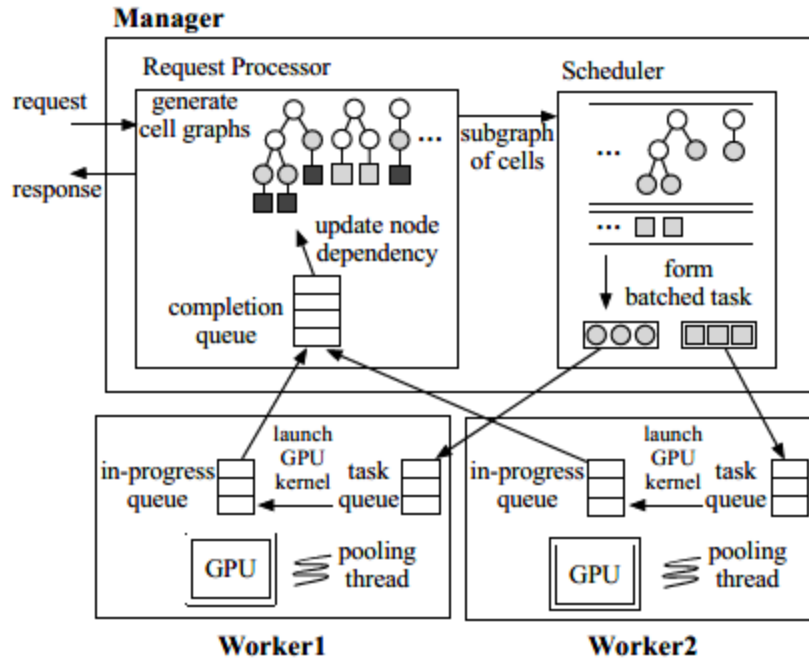


Figure 6: The system architecture of BatchMaker. In the cell graph, black means computed nodes, grey means nodes whose input is ready, and white means input dependency is not satisfied.

230. As can be seen from Figure 6, BatchMaker includes a Worker “Workers execute tasks on GPUs and notify the manager when its tasks complete.”

231. In light of the above, Gao discloses an execution engine configured as a GPU.

10. Claim 9. The method of claim 1, wherein each token in the sequence of input tokens represents a text unit.

232. An RNN is a Deep Neural Network (DNN) architecture designed to process sequential data of variable length – often used for language processing applications. Gao at p. 2. RNNs help solve tasks such as predicting which words would follow in a sentence. Gao at p. 2. For example, Gao provides figure 1 (reproduced below), which shows an unfolded dataflow graph for an input where the RNN is used for predicting the word most likely to follow the following input text: “system research is.” Gao at p. 2; Gao at Fig. 1. “At each time step, one input position is consumed and the calculated value of the hidden unit is then passed to the successor cell in the next time step.” Gao at p. 2. Once all three steps are completed, the full context is known and the input sentence can be used to predict the next word. Gao at p. 2.

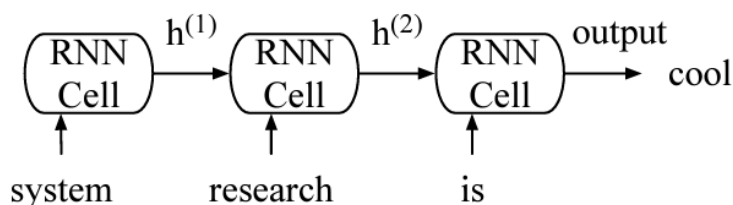


Figure 1: An unfolded chain-structured RNN. All RNN Cells in the chain share the same parameter weights.

See also Gao at p. 2:

In its simplest form, we can view RNNs as operating on an input sequence, $X = [x(1), x(2), \dots, x(\tau)]$, where $x(i)$ represents the input at the i -th position (or timestep). For language processing, the input X would be a sentence, and $x(i)$ would be the vector embedding of the i -th word in the sentence...RNNs can be used to model a natural language, solving tasks such as predicting the most likely word following an input sentence. For example, we can use an RNN to process the input sentence “system research is” and to derive the most likely next word from the RNN’s output.

233. Accordingly, Gao provides that each token in the sequence of input tokens represents a text unit. Moreover, accessing a machine learned transformer model as taught by Katharopoulos does not alter the inputs tokens that can be received by Gao. Katharopoulos also teaches that transformer models are operable with text such as natural language processing. Katharopoulos at p. 1.

11. Claim 10.

10[PRE] A non-transitory computer-readable storage medium storing computer program instructions executable to perform operations for dynamically executing batches of requests on one or more execution engines running a machine-learning transformer model, the operations comprising:

10[A] receiving, by a serving system, one or more requests for execution, the serving system including a scheduler and one or more execution engines each coupled to access a machine-learning transformer model including at least a set of decoders;

10[B] scheduling, by the scheduler, a batch of requests including the one or more requests for execution on an execution engine;

10[C] generating, by the execution engine, a first set of output tokens by applying the transformer model to a first set of

inputs for the batch of requests, wherein applying the transformer model comprises applying at least one batch operation to one or more input tensors associated with the batch of requests;

10[D] receiving, by a request processor, a new request from a client device, the new request including a sequence of input tokens;

10[E] scheduling, by the scheduler, a second batch of requests additionally including the new request for execution on the execution engine, the second batch of requests scheduled responsive to determining that the execution engine has memory available to execute the second batch of requests, wherein in a second set of inputs for the second batch of requests, a length of the sequence of input tokens for the new request is different from a length of an input for at least one request other than the new request; and

10[F] generating, by the execution engine, a second set of output tokens by applying the transformer model to the second set of inputs for the second batch.

234. Claim 10 is substantively the same as Claim 1 and I hereby incorporate by reference all of my analysis for claim 1 to explain why Claim 10 is taught by the combination of Gao and Katharopoulos.

235. The only difference in Claim 10 from Claim 1 is that Claim 10 recites the limitations are performed by “A non-transitory computer-readable storage medium storing computer program...” Both Gao and Katharopoulos disclose their systems as computer programs executed on non-transitory computer-readable storage medium. For example Gao “runs on a single machine [a computer] with potentially many GPU devices. Its overall system architecture is depicted in Figure 6” illustrating

numerous computer components. Gao at p. 5. In addition, both Gao and Katharopoulos disclose algorithms which is executable code that runs on computer-readable storage mediums. Gao at p. 6 and Katharopoulos at p. 5.

12. Claim 11.

11[A] The non-transitory computer-readable storage medium of claim 10, the operations further comprising: responsive to determining that a request in the first batch of requests has been completed, providing output tokens generated for the completed request to a client device as a response to the request, and

11[B] wherein the second batch of requests includes at least one of the remaining requests from the one or more requests and the new request.

236. Claim 11 is substantively the same as Claim 2 and I hereby incorporate by reference all of my analysis for claim 2 to explain why Claim 11 is taught by the combination of Gao and Katharopoulos.

13. Claim 12. The non-transitory computer-readable storage medium of claim 11, wherein the request is associated with a cache memory in the execution engine dedicated for storing an internal state for the request, and responsive to determining that the request has been completed, freeing the dedicated cache memory for the request in the execution engine.

237. Claim 12 is substantively the same as Claim 3 and I hereby incorporate by reference all of my analysis for claim 3 to explain why Claim 12 is taught by the combination of Gao and Katharopoulos.

14. Claim 13. The non-transitory computer-readable storage medium of claim 10, wherein the input for the at least one request is at least one output token from the first set of output tokens for the at least one request, and wherein a length of the sequence of input tokens for the new request is different from a length of the at least one output token for the at least one request.

238. Claim 13 is substantively the same as Claim 4 and I hereby incorporate by reference all of my analysis for claim 4 to explain why Claim 13 is taught by the combination of Gao and Katharopoulos.

15. Claim 14.

14[A] The non-transitory computer-readable storage medium of claim 13, wherein the execution engine includes a cache memory for maintaining a key cache tensor for storing keys and a value cache tensor for storing values for the at least one request, and

14[B] wherein after scheduling the second batch of requests, allocating, by the execution engine, a new cache memory dedicated to maintaining a key cache tensor and a value cache tensor for the new request.

239. Claim 14 is substantively the same as Claim 5 and I hereby incorporate by reference all of my analysis for claim 5 to explain why Claim 14 is taught by the combination of Gao and Katharopoulos.

16. Claim 15.

The non-transitory computer-readable storage medium of claim 14, wherein after generating the second set of

output tokens, a length of the key cache tensor for the at least one request is different from a length of a key cache tensor for the new request, and a length of the value cache tensor for the at least one request is different from a length of a value cache tensor for the new request.

240. Claim 15 is substantively the same as Claim 6 and I hereby incorporate by reference all of my analysis for claim 6 to explain why Claim 15 is taught by the combination of Gao and Katharopoulos.

17. Claim 16.

The non-transitory computer-readable storage medium of claim 10, after receiving the new request from the client device, determining, by the scheduler, that there is insufficient memory available to execute the second batch of requests on a second execution engine different from the execution engine, and responsive to the determination for the second execution engine, determining whether the execution engine has the memory available to execute the second batch of requests.

241. Claim 16 is substantively the same as Claim 7 and I hereby incorporate by reference all of my analysis for claim 7 to explain why Claim 16 is taught by the combination of Gao and Katharopoulos.

18. Claim 17.

The non-transitory computer-readable storage medium of claim 10, wherein the execution engine is configured as a graphics processing unit (GPU) or a tensor processing unit (TPU).

242. Claim 17 is substantively the same as Claim 8 and I hereby incorporate by reference all of my analysis for claim 8 to explain why Claim 17 is taught by the combination of Gao and Katharopoulos.

19. Claim 18.

The non-transitory computer-readable storage medium of claim 10, wherein each token in the sequence of input tokens represents a text unit.

243. Claim 18 is substantively the same as Claim 9 and I hereby incorporate by reference all of my analysis for claim 9 to explain why Claim 18 is taught by the combination of Gao and Katharopoulos.

VIII. OATH

I declare that all statements made herein of my own knowledge are true and that all statements made on information and belief are believed to be true; and further that these statements were made with the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under Section 1001 of Title 18 of the United States Code.

Dated: July 28th, 2024

By:  _____