

UNITED STATES PATENT AND TRADEMARK OFFICE

BEFORE THE PATENT TRIAL AND APPEAL BOARD

HUGGING FACE, INC.,
PETITIONER,

v.

FRIENDLIAI INC.,
PATENT OWNER.

Case IPR2024-01234
Patent 11,442,775

**PETITION FOR *INTER PARTES* REVIEW OF USPN 11,442,775
UNDER 35 U.S.C. §§ 311 *ET SEQ.* AND
37 C.F.R. § 42.100 *ET SEQ.***

TABLE OF CONTENTS

I.	Introduction.....	1
II.	Summary of the Argument	1
III.	MANDATORY NOTICES UNDER 37 C.F.R. § 42.8(a)(1).....	4
	A. Real Party In Interest Under 37 C.F.R. § 42.8(b)(1).....	4
	B. Related Matters Under 37 C.F.R. § 42.8(b)(2)	4
	C. Lead and Back-Up Counsel Under 37 C.F.R. § 42.8(b)(3)	4
	D. Service Information Under 37 C.F.R. § 42.8(b)(4).....	5
	E. Payment of Fees Under 37 C.F.R. § 42.15.....	5
	F. Certification of Word Count Under 37 C.F.R. § 42.24(d).....	6
IV.	GROUND FOR STANDING UNDER 37 C.F.R. § 42.104(a).....	6
V.	IDENTIFICATION OF GROUNDS FOR WHICH REVIEW IS REQUESTED UNDER 37 C.F.R. § 42.104(b)(1)	6
VI.	HOW THE CHALLENGED CLAIMS ARE TO BE CONSTRUED UNDER 37 C.F.R. § 42.104(b)(3)	7
VII.	OVERVIEW OF THE '775 PATENT	8
	A. Summary of the '775 Specification's Description of Claimed Matter	8
	B. Summary of the '775 Patent Prosecution History.....	11
VIII.	STATE OF TECHNOLOGY	13
	A. Machine Learning and RNNs.....	13

B.	Transformers	17
C.	Batching in Machine Learning	19
IX.	OVERVIEW OF THE PRIOR ART	21
A.	Summary of Gao	21
B.	Summary of Katharopoulos	25
X.	HOW THE CLAIMS ARE UNPATENTABLE UNDER 37 C.F.R. § 42.104(b)(4)	27
A.	Level of Skill in the Art.....	27
B.	Ground 1 – Gao in view of Katharopoulos renders obvious claims 1-18 under 35 U.S.C. § 102	27
XI.	DISCRETIONARY DENIAL IS NOT SUPPORTED	78
A.	35 U.S.C. § 325(d).....	78
B.	35 U.S.C. § 314(a).....	79
XII.	CONCLUSION.....	80

LIST OF EXHIBITS

EX	DESCRIPTION
1001	U.S. Patent No. 11,442,775
1002	Declaration of Ghaith Hammouri
1003	C.V. of Ghaith Hammouri
1004	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”)
1005	Katharopoulos, <i>et. al.</i> , <i>Transformers are RNNs: Fast Autoregressive Transformers with Linear Attention by Katharopoulos 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”)
1006	Original Complaint, <i>FriendliAI Inc. v. Hugging Face, Inc.</i> , Case No. 23-816-MN, D. Del. (filed July 28, 2023)
1007	First Amended Complaint, <i>FriendliAI Inc. v. Hugging Face, Inc.</i> , Case No. 23-816-MN, D. Del. (filed January 8, 2024)
1008	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”)
1009	A. Vaswani, <i>et. al.</i> , <i>Attention is All you Need</i> , Advances in Neural Information Processing Systems, 2017 (“Vaswani”)
1010	Prosecution history of application 17/542,193
1011	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)
1012	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)

EX	DESCRIPTION
1013	Plaintiff’s Final Infringement Contentions of ’775 Patent, FriendliAI Inc. v. Hugging Face, Inc., Case No. 23-816-MN, D. Del. (served on July 1, 2024)
1014	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”)
1015	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”)
1016	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”)
1017	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”)
1018	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”)
1019	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”)
1020	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”)

EX	DESCRIPTION
1021	Guy Blanc, <i>et al.</i> , <i>Adaptive Sampled Softmax with Kernel Based Sampling</i> , 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”)
1022	Claim appendix for claims 1-18 of the ’775 Patent

I. Introduction

Hugging Face, Inc. (“Petitioner”), hereby respectfully requests *Inter Partes* Review pursuant to 35 U.S.C. §§ 311 *et seq.* and 37 C.F.R. §§ 42.100 *et seq.*, of claims 1-18 (the “Challenged Claims”) of U.S. Patent No. 11,442,775 (“the ’775 Patent”) issued on Sept. 13, 2022 to Gyeong-In Yu *et. al.* See Ex-1001.

As explained in detail below, the Challenged Claims are rendered obvious in view of the prior art cited herein. Therefore, the Board should institute an *inter partes* review of claims 1–18 of the ’775 Patent.

II. Summary of the Argument

The ’775 Patent generally relates to an alleged improvement for machine learning transformer neural network models. Ex-1001, 1:7–9. In its complaint in the underlying litigation, Patent Owner states the ’775 Patent claims are directed to “methods used to process batches of transformer-based requests, which involves scheduling batches at an iteration-level.” Ex-1007, ¶¶ 23-27. According to Patent Owner, “iteration-level scheduling...allows for a finished request to be sent to a client, and for new requests to be sent to the execution engine, before all requests in a batch are completed.” *Id.*, ¶ 14.

However, after filing the ’775 Patent, the inventors acknowledged in their published “Orca” paper that “iteration-level scheduling” was a previously known method to optimize a recurrent neural network (RNN). Ex-1008, 532–533. Indeed,

when discussing the Gao reference (Ex-1004) cited in this IPR Petition, the inventors of the challenged '775 Patent stated:

We would like to highlight BatchMaker [Gao] as one of the most relevant previous works. BatchMaker is a serving system for RNNs that performs scheduling and batching at the granularity of RNN cells,...BatchMaker allows a newly arrived request for RNN to join (or a finished request to leave) the current executing batch without waiting for the batch to completely finish.

Ex-1008, 532-533 (emphasis added).¹ This description of Gao is the same as Patent Owner's description of the '775 claims. *Id.*, ¶ 14 ('775 claims "allows for a finished request to be sent to a client, and for new requests to be sent to the execution engine, before all requests in a batch are completed.").

Notably, Patent Owner alleges in its Complaint against Petitioner that the Orca paper embodies the '775 Patent. Ex-1007, ¶ 36 (Patent Owner stating that "The claimed advancements were described in a paper, titled "Orca:..." and ¶ 38 ("PeriFlow (aka Orca) practices the '775 patent"). Thus, the admissions about Gao in the Orca paper is strong evidence that Gao teaches scheduling batches at an iteration-level, the feature that is allegedly claimed in the '775 Patent.

The only difference in the claims of the '775 Patent and Gao is that the claims of the '775 Patent recite a "transformer model" while Gao discloses a RNN model. Yet, the application of Gao's scheduling technique for an RNN model to a

¹ Emphasis always added by Petitioner unless otherwise noted.

transformer model was not inventive, but rather a predictable and obvious implementation of a known improvement for one machine learning model applied to a different machine learning model. *KSR International Co. v. Teleflex Inc.*, 550 U.S. 398, 401 (2007).

Indeed, by the time of the '775 Patent, it was known that certain transformer models act as an RNN. For example, Katharopoulos is titled and teaches “Transformers are RNNs:...” thereby explicitly linking these two machine learning models. *See* Ex-1005, 5 (“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).”). As discussed in this IPR petition, the combination of Gao’s scheduling and batching with Katharopoulos’s transformer model was obvious and teaches all the claims of the '775 Patent.

Lastly, the Orca paper attempts to distinguish Gao by arguing that “BatchMaker [Gao] cannot make batches of cells for Transformer models” because of certain performance characteristics. *See* Ex-1008, 533. Yet, the Orca Paper provides nothing to support these conclusions and further, these conclusions are irrelevant to the broadly claimed transformer model in the '775 Patent. The Orca Paper also does not address applying Gao to Katharopoulos’s transformer model which uses linear attention and overcomes the Orca paper’s criticism of applying Gao to a transformer model. *See e.g.*, Ex-1005, 1 (“In this paper, we introduce the

linear transformer model that significantly reduces the memory footprint and scales linearly with respect to the context length.”).

III. MANDATORY NOTICES UNDER 37 C.F.R. § 42.8(a)(1)

Petitioner satisfies each requirement for *Inter Partes* Review of the ’775 patent pursuant to 37 C.F.R. § 42.8(a)(1).

A. Real Party In Interest Under 37 C.F.R. § 42.8(b)(1)

The Petitioner and real party in interest is Hugging Face, Inc. Petitioner is incorporated in Delaware with a principal business address of 20 Jay St, Suite 620, Brooklyn, NY 11201.

B. Related Matters Under 37 C.F.R. § 42.8(b)(2)

The ’775 Patent is presently asserted by the Patent Owner against Petitioner in *FriendlyAI Inc. v. Hugging Face, Inc.*, Case No. 1:23-cv-00816-MN, D. Del. (filed on July 28, 2023). *See* Exs. 1006 and 1007.

To the best of Petitioners’ knowledge, the ’775 Patent has not been at issue in any other litigation or PTAB proceeding before the instant petition was filed.

C. Lead and Back-Up Counsel Under 37 C.F.R. § 42.8(b)(3)

Petitioner is represented by the following counsel:

Lead Counsel	Backup Counsel
James P. Murphy Reg. No. 55,474 Polsinelli PC 1000 Louisiana Street Suite 6400 Houston, Texas 77002 Tel: (713) 374-1631 jpmurphy@polsinelli.com	Adam P. Daniels, Reg. No. 66,681 Polsinelli LLP 2049 Century Park E. Suite 2900 Los Angeles, CA 90067 Tel: (310) 556-6754 adaniels@polsinelli.com

Pursuant to 37 C.F.R. § 42.10(b), Powers of Attorney have been filed with this Petition.

D. Service Information Under 37 C.F.R. § 42.8(b)(4)

Physical mailing service information for lead and back-up counsel is as follows:

James Murphy
Polsinelli PC
1000 Louisiana Street
Suite 6400
Houston, Texas 77002

Petitioner also consents to service by e-mail at the above e-mail addresses provided above for lead and backup counsel.

E. Payment of Fees Under 37 C.F.R. § 42.15

All required fees have been paid with the filing of this Petition. Petitioner further authorizes the U.S. Patent & Trademark Office to charge Deposit Account

No. 50-1662 for any fees, including the fee set forth in 37 C.F.R. § 42.15(a) for this Petition.

F. Certification of Word Count Under 37 C.F.R. § 42.24(d)

Petitioner certifies that the word count in this Petition, including all footnotes, is 13,982 words as counted by the word-processing program (Microsoft Word for Office 365) used to generate this Petition, where such word count excludes the table of contents, mandatory notices, certificate of service, list of exhibits, and this certificate of word count. This Petition is in compliance with the 14,000 word limit set forth in 37 C.F.R. § 42.24(a)(1)(i).

IV. GROUNDS FOR STANDING UNDER 37 C.F.R. § 42.104(a)

Petitioner certifies that the '775 patent is available for *inter partes* review. Petitioner is not barred or estopped from requesting an *inter partes* review of the '775 patent claims on the grounds identified in this Petition. 37 C.F.R. § 42.104(a).

V. IDENTIFICATION OF GROUNDS FOR WHICH REVIEW IS REQUESTED UNDER 37 C.F.R. § 42.104(b)(1)

Petitioner asserts that claims 1-18 (the “Challenged Claims”) of the '775 patent are unpatentable based on the following ground:

Ground 1: Claims 1-18 are rendered obvious under 35 U.S.C. § 103 by Gao in view of Katharopoulos. *See* Ex-1002, ¶¶12-13.

VI. HOW THE CHALLENGED CLAIMS ARE TO BE CONSTRUED UNDER 37 C.F.R. § 42.104(b)(3)

In an IPR, claim terms are to be construed in accordance with the standard set forth in *Phillips*. *Phillips v. AWH Corp.*, 415 F.3d 1303, 1312 (Fed. Cir. 2005) (en banc). Further, claim terms need only be construed “to the extent necessary to resolve the controversy.” *Vivid Techs., Inc. v. Am Sci. & Eng’g, Inc.*, 200 F.3d 795, 803 (Fed. Cir. 1999). In the underlying litigation, the Patent Owner has asserted that all claim terms are to be given their plain and ordinary meaning without providing any further constructions. Ex-1011. Petitioner has proposed constructions for certain claim terms. Ex-1012, 2–3. Here, Petitioner does not believe that any term requires express construction to resolve the invalidity grounds presented in this Petition since the prior art renders claims 1–18 under any reasonable interpretation of the claims.

Claim construction negotiations are ongoing in the underlying litigation, and substantive briefing has not been filed. At present, Petitioner has proposed that certain limitations of the claims of the ’775 Patent are indefinite because Petitioner’s infringement positions require a claim scope that is unsupported and cannot be determined with reasonable certainty by a POSITA when reading the claims in light of the specification and prosecution history. Ex-1012, 4. To be clear, the uncertainty of the scope is an issue of infringement due to the unreasonable breadth in which Patent Owner is interpreting the claims to allege infringement. *See* Ex-1013. Yet, the arguments presented in this Petition do not rely on Patent Owner’s unreasonable

interpretations of the boundaries of the claim scope (rather the prior art reads on the claims under a narrower scope than Patent Owner is alleging). Thus, the Board does not need determine if the outer boundaries of the claims as asserted by Patent Owner is indefinite in order to determine that the prior art renders obvious claims 1-18 unpatentable. *See* Ex-1002, ¶25.

VII. OVERVIEW OF THE '775 PATENT

A. Summary of the '775 Specification's Description of Claimed Matter

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, Abstract. 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, Figures 5A–5D; 22:22–24:38; *see* Ex-1002, ¶33.

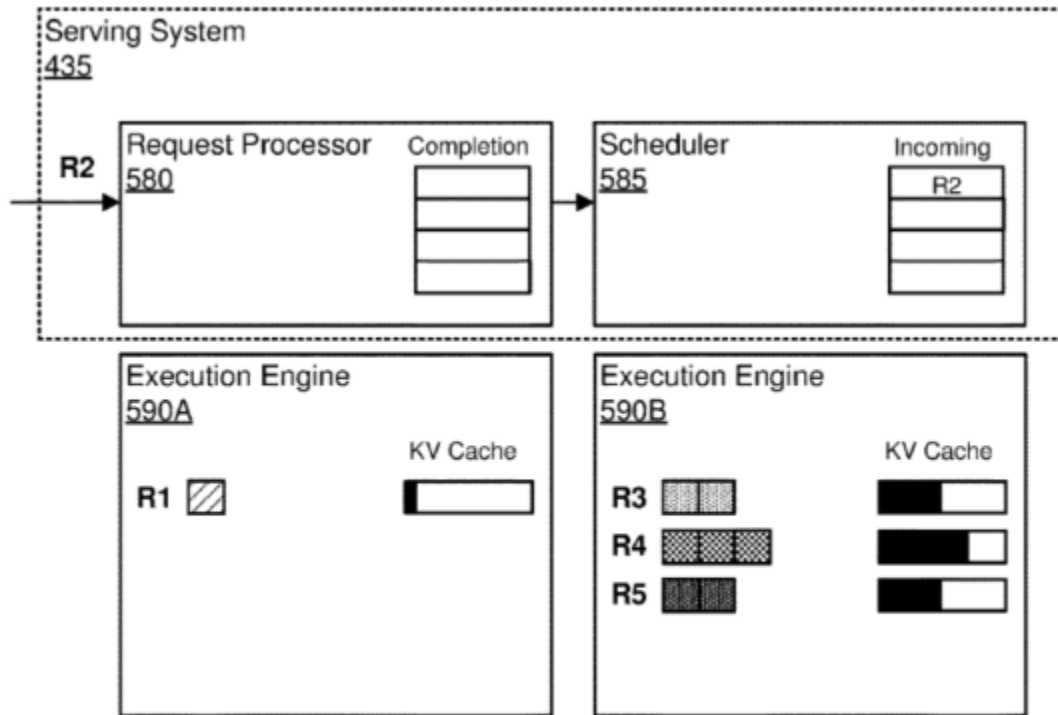


FIG. 5A

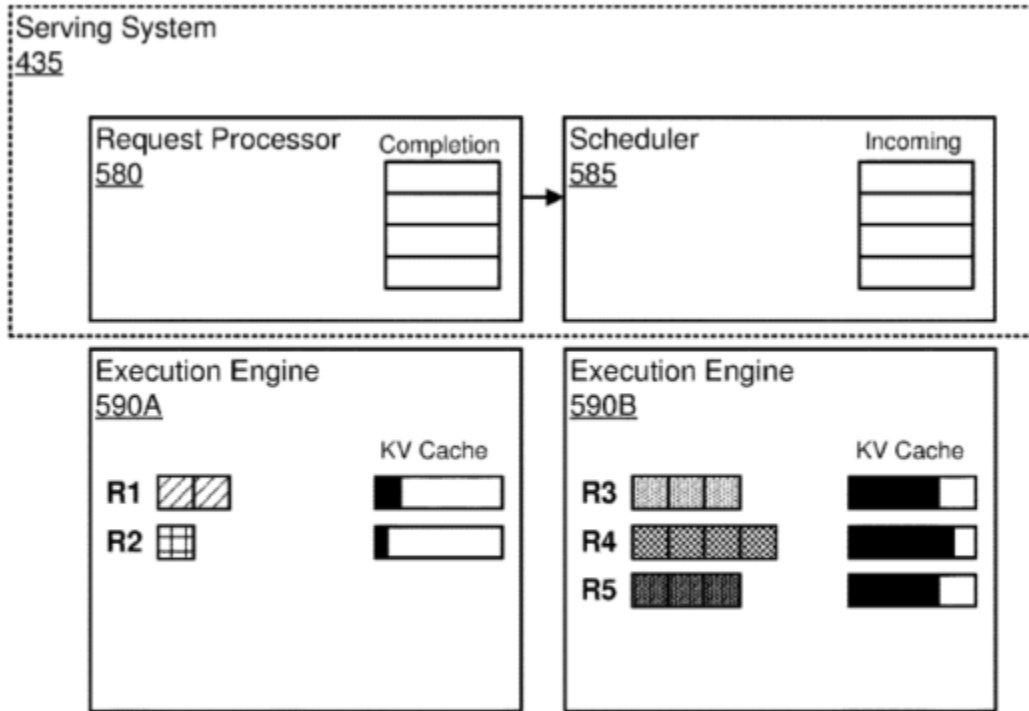


FIG. 5B

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, 22:58–23:29. 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, 23:30–50; *see* Ex-1002, ¶¶34-38.

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, 23:51–58; *see* Ex-1002, ¶39.

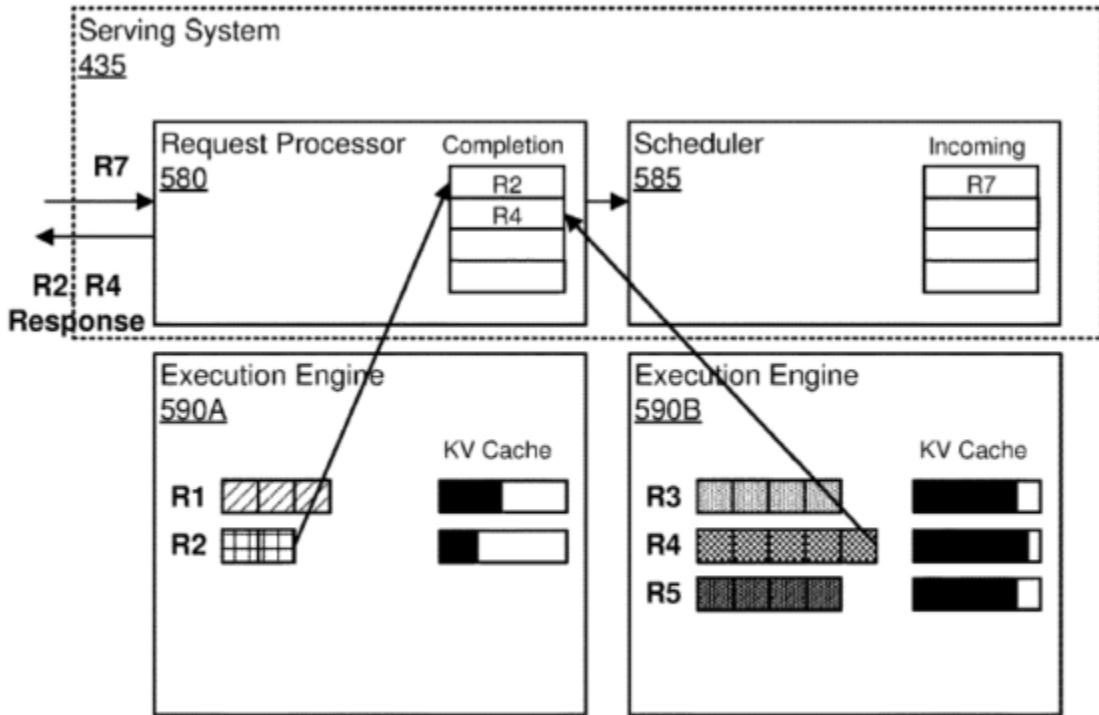


FIG. 5C

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, 24:10–16; *see* Ex-1002, ¶¶40-41.

B. Summary of the ’775 Patent Prosecution History

On December 7, 2021, applicants filed Application No. 17/542,193 (“the ’193 Application”) which issued as the ’775 Patent. *See* Ex-1010. During prosecution of the ’193 Application, the Examiner issued a single office action rejecting certain pending claims, including all independent claims, as being rendered obvious by U.S.

Patent Publication 2021/0192314 to Aarts *et. al.* (“Aarts”) in view of U.S. Patent No. 10,846,096 to Chung *et. al.* (“Chung”). Ex-1010, 120–124.

In response to the office action, the applicant amended both pending independent claims to further recite “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.” *Id.*, 181 and 184. Applicant argued that this amendment in conjunction with the scheduler limitation reciting “a second batch of requests additionally including the new request” distinguishes the prior art as emphasized in applicant’s response below:

- the scheduler schedules a second batch of requests additionally including the new request for execution on the execution engine, 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

Id., 189 (emphasis in original).

Applicant argues the independent claims as amended requires “the second batch of requests is modified to include a new request in addition to the one or more requests of the [first] batch of requests.” *Id.*, 189 (bracket in original); *see also id.* 191 (Applicant states “Claim 10 recites similar features as claim 1” and

distinguishable over the prior art for the same reasons as claim 1). According to the applicant, this distinguishes the claims from the prior art because:

In existing batching methods for transformer models, it is difficult to modify a batch of requests once it has started to process on an execution engine, since the length of the inputs or internal states are the same across the requests in the batch.

Id., 189.

Applicants then argue that neither Aarts nor Chung teaches batching in the context of a “machine-learned transformer model” and also that neither teaches “subsequently scheduling a second batch of requests that is modified to include a new request in addition to the first batch of requests, in which a length of the sequence of input tokens for a new request is different from a length of an input for at least one request other than the new request in the batch.” *Id.*, 190–191.

The Examiner then allowed all claims without any further comments. *Id.*, 199.

VIII. STATE OF TECHNOLOGY²

A. Machine Learning and RNNs

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

² Cited references not named in a ground of rejection are cited for the purpose of showing the state of the art and the background knowledge of a POSITA. *Randall Mfg. v. Rea*, 733 F.3d 1355, 1362-63 (Fed. Cir. 2013).

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. *See* Ex-1002, ¶43.

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. *See* Ex-1002, ¶44.

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. *See Ex-1002, ¶45.*

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. *See Ex-1002, ¶46.*

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. *See Ex-1002, ¶47.*

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. *See* Ex-1002, ¶48.

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 processes 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. *See* Ex-1002, ¶49.

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. *See* Ex-1002, ¶50.

B. Transformers

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. *See* Ex-1009. Simply put, the attention mechanism takes in an input encoded as a vector and maps it to an output that is also a vector. *See* Ex-1002, ¶51.

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. *See* Ex-1002, ¶52.

Although the Attention mechanism already existed in the literature (Ex-1015), 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. *See* Ex-1002, ¶53.

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. Ex-1014. *See* Ex-1002, ¶54.

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. *See* Ex-1002, ¶55.

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. *See* Ex-1002, ¶56.

Many different Transformer architectures have been proposed, but perhaps one of the most popular was the Decoder-Only-Transformer (Ex-1017), which is most known to have been used in GPT (Generative-Pretrained Transformer). *See* Ex-1018. *See* Ex-1002, ¶57.

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. *See* Ex-1002, ¶58.

C. Batching in Machine Learning

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. *See* Ex-1002, ¶59.

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. *See* Ex-1002, ¶60.

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. *See* Ex-1002, ¶61.

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. *See* Ex-1002, ¶62.

IX. OVERVIEW OF THE PRIOR ART

A. Summary of Gao

Low Latency RNN Inference with Cellular Batching by Gao et al., was published and presented at a conference in 2018 and is therefore prior art under 35 U.S.C. §103.

Gao proposes a technique called “cellular batching” to improve the latency and throughput of Recurrent Neural Network (RNN) inference. An RNN is a Deep Neural Network (DNN) architecture designed to process sequential data of variable length – often used for language processing applications. Ex-1004, 2. RNNs help solve tasks such as predicting which words would follow in a sentence. Ex-1004, 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.” Ex-1004, 2 and FIG.

1. Once all three steps are completed, the full context is known and the input sentence can be used to predict the next word. Ex-1004, 2. *See* Ex-1002, ¶¶64-66.

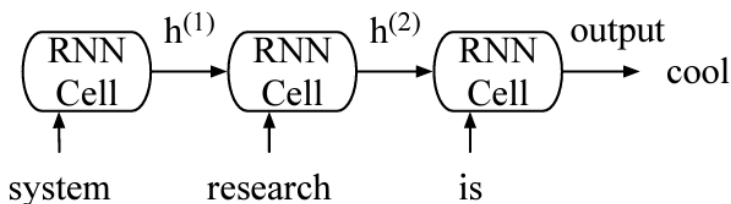
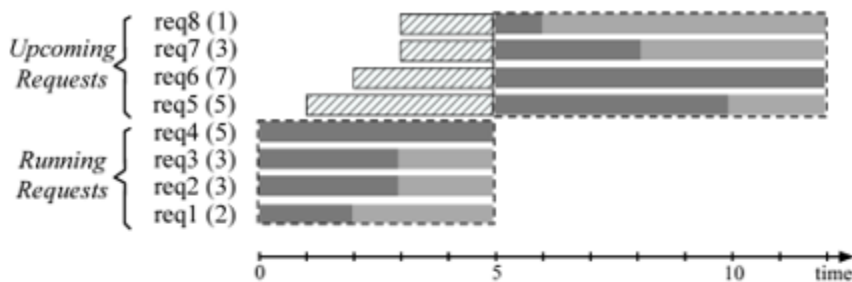


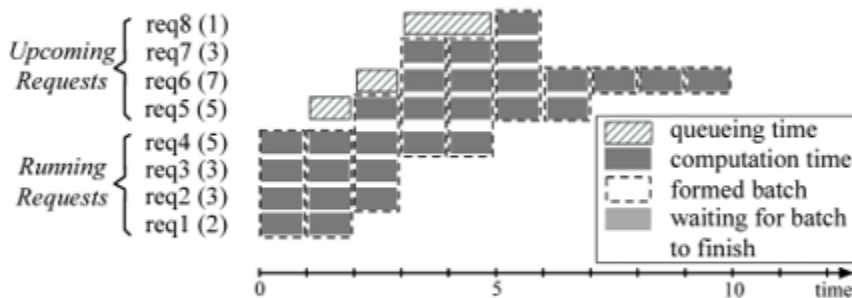
Figure 1: An unfolded chain-structured RNN. All RNN Cells in the chain share the same parameter weights.

While the above example is a chain-like structure, RNNs are not so limited. Ex-1004, 2. 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.” Ex-1004, 1. However, when the batched input has different sizes, not all operators in a graph can be batched fully, reducing throughput. Ex-1004, 1-2. *See* Ex-1002, ¶¶67-71.

To address these issues, Gao introduces the concept of cellular batching, a technique that improves both latency and throughput of RNN inference. Ex-1004, 2. Figure 5 illustrates the difference between graph batching and cellular batching when processing the same 8 requests. Ex-1004, 6 and FIG. 5. *See* Ex-1002, ¶72. Ex-1004, 6.



(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.

Ex-1004, 6.

The authors called their cellular batching system “BatchMaker.” Ex-1004, 2. BatchMaker consists of a manager and worker threads, handling request processing and GPU execution, respectively. Ex-1004, Fig. 6. See Ex-1002, ¶72.

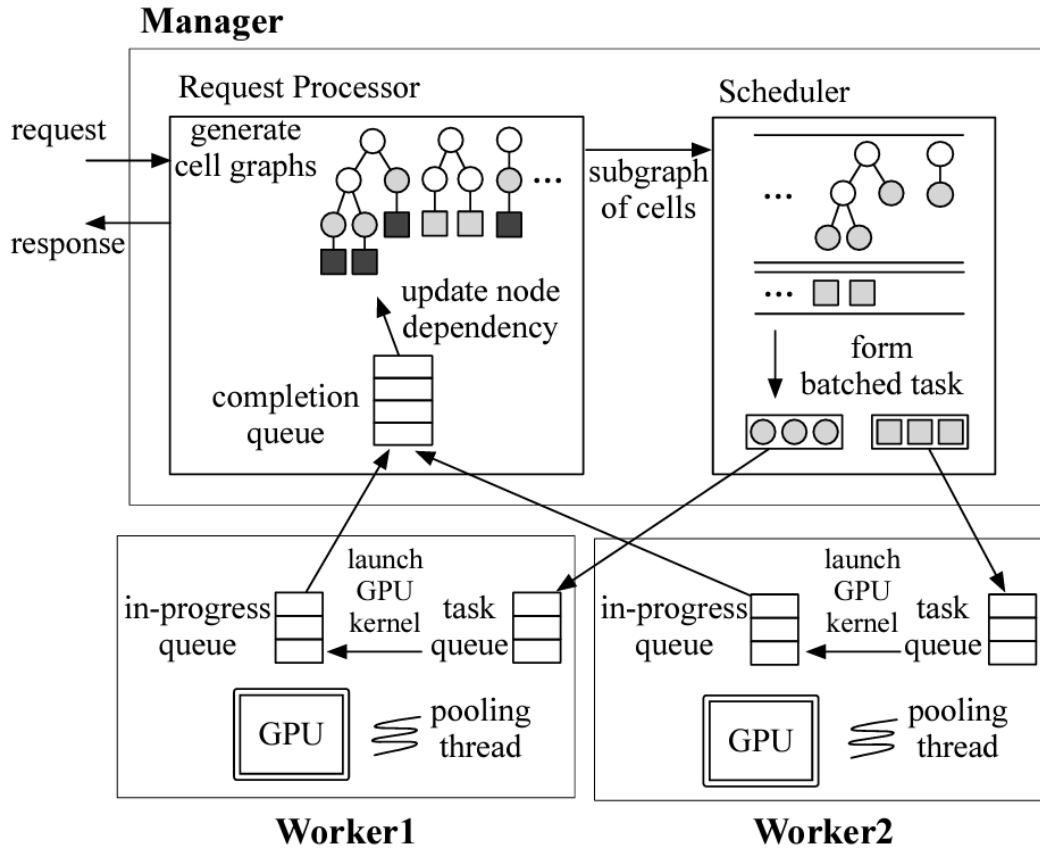


Figure 6: The system architecture of BatchMaker. In the cell graph, black means computed nodes, grey means nodes whose input is ready, and white means input dependency is not satisfied.

The manager, which consists of a request processor and scheduler, processes arriving requests and submits batch computation tasks to work for execution. Ex-1004, 6. The request processor, “generates the corresponding cell graph for the request” and tracks and updates the dependencies of each node. Ex-1004, 6. “Once the dependencies have been satisfied (aka its inputs are ready” the scheduler “forms

batched tasks among ready nodes of the same cell type.” Ex-1004, 6. *See* Ex-1002, ¶73.

Once the desired batch size is reached, the task is passed to the workers. Ex-1004, 6–7. “Workers execute tasks on GPUs and notify the manager when its tasks complete.” Ex-1004, 6. *See* Ex-1002, ¶¶74-75.

B. Summary of Katharopoulos

Transformers are RNNs: Fast Autoregressive Transformers with Linear Attention by Katharopoulos et al., (“Katharopoulos”) published and presented, a conference in 2020 and is therefore prior art under 35 U.S.C. §103.

Katharopoulos 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.” Ex-1005, 1. Katharopoulos observes that while transformer models, originally introduced by Vaswani (Ex-1009), “have demonstrated impressive results” when dealing with natural language, audio, and images, “the benefits often come with a very high computational and memory costs.” Ex-1005, 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 $O(N^2)$.” Ex-1005, 1. *See* Ex-1002, ¶¶76-77.

Accordingly, Katharopoulos introduces “the linear transformer model that significantly reduces the memory footprint and scales linearly with respect to the

context length.” Ex-1005, 1. 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. Ex-1005, 2–5. *See* Ex-1002, ¶¶78-79.

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).

Ex-1005, 5.

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:

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

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

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

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

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

Ex-1005, 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.” Ex-1005, 5.
See Ex-1002, ¶¶80-82.

X. HOW THE CLAIMS ARE UNPATENTABLE UNDER 37 C.F.R. § 42.104(b)(4)

A. Level of Skill in the Art

In view of the subject matter of the ’775 patent, a POSITA 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. *See* Ex-1002, ¶¶26-31.

B. Ground 1 – Gao in view of Katharopoulos renders obvious claims 1-18 under 35 U.S.C. § 102

1. Summary of the Combination and Motivation to Combine

It is straight forward to leverage the 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. *See* Ex-1002, ¶83.

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* Ex-1018. *See* Ex-1002, ¶84.

In view of the above, a POSITA would leverage Gao's batching scheme to achieve efficient batching in a DOT with a linearized attention function as taught by Katharopoulos. *See* Ex-1002, ¶85.

Gao's Cellular Batching

Gao disclose cellular batching that operates on the functional cell level of an RNN rather than on the whole data graph. In essence, the key insight of Gao is that, if batching is carried out on the cell level, then one can leverage the identical components in RNN cells in order to batch multiple requests and execute them all at the same time. Accordingly, Gao teaches batching any number of requests (depending on the underlying processing and memory capacity) and executing 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. *See* Ex-1002, ¶86.

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. *See* Ex-1002, ¶87.

Thus, Gao teaches cellular batching provides 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. *See* Ex-1002, ¶88.

Katharopoulos Linearized Attention and Transformers as RNNs

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. *See* Ex-1002, ¶89.

Many papers have discussed specifics and features of different attention-functions (*see* Ex-1019 and Ex-1021); however, most relevant to our discussion here is the class of attention-functions introduced in Katharopoulos. *See* Ex-1002, ¶90.

Katharopoulos discloses 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, Katharopoulos that 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. *See* Ex-1002, ¶91.

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. *See* Ex-1002, ¶92.

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. *See* Ex-1002, ¶93.

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 or the Decoder stack or can be processed to predict the next element in the sequence. *See* Ex-1002, ¶94.

Katharopoulos significantly improves 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 $O(N)$. *See* Ex-1002, ¶95.

Moreover, Katharopoulos teaches even a softmax attention-function could be cast as an RNN cell, given that masked Attention is used. In fact, the softmax attention-function itself has been approximated using a linear dot product of feature maps (*see* Ex-1021 and Ex-1020) which is why Katharopoulos predicts that it might be possible to use, even a softmax attention-function, with their introduced work. *See* Ex-1002, ¶96.

Combining Gao’s Cellular Batching with Katharopoulos’s Transformer

Given the significant speed ups presented by cellular batching and linearized attention-functions, it is natural to 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 only using multiple layers,

and thus, cellular batching can be immediately leveraged to batch inputs into the same cell-types as detailed in Gao. *See* Ex-1002, ¶97.

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 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. *See* Ex-1002, ¶98.

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. *See* Ex-1002, ¶99.

This process is repeated until the final output is a special output element (*e.g.*, $\langle \text{eos} \rangle$) is generated. 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. *See* Ex-1002, ¶100.

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 paragraph). 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. *See* Ex-1002, ¶101.

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. *See* Ex-1002, ¶102.

The first Decoder-Cells (in all layers) are initialized with a zero state, as shown in Katharopoulos (Eq. 16 and 17). The final layer of Decoder-Cells do 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. *See* Ex-1002, ¶103.

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. <eos>) is generated, at which point the computational graph is terminated. *See* Ex-1002, ¶104.

As can be seen from the description here, this process is essentially identical to what is done in a Seq2Seq RNN model which also uses 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, one can immediately see a DOT with linearized attention function as a Seq2Seq RNN circuit. *See* Ex-1002, ¶105.

Once casted in the language of a Seq2Seq RNNs, a DOT with linearized attention can immediately leverage cellular-batching exactly as presented in Gao. In essence, one can use their 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 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. *See* Ex-1002, ¶106.

A POSITA would fully appreciate the effect of applying BatchMaker to DOTs by observing the improvements that BatchMaker produces when applied to Seq2Seq RNNs. In this case, BatchMaker reports the ability to make interesting scheduling choices on multiple GPUs due to the presence of multiple Cell types. Moreover, 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. *See* Ex-1002, ¶107.

Finally, applying Gao's Cellular batching to other classes of Transformer circuits that are fundamentally different from RNNs might 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. *See* Ex-1002, ¶108.

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. *See* Ex-1002, ¶109.

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. *See* Ex-1002, ¶110.

Conclusion

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. *See* Ex-1002, ¶111.

2. Claim 1

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

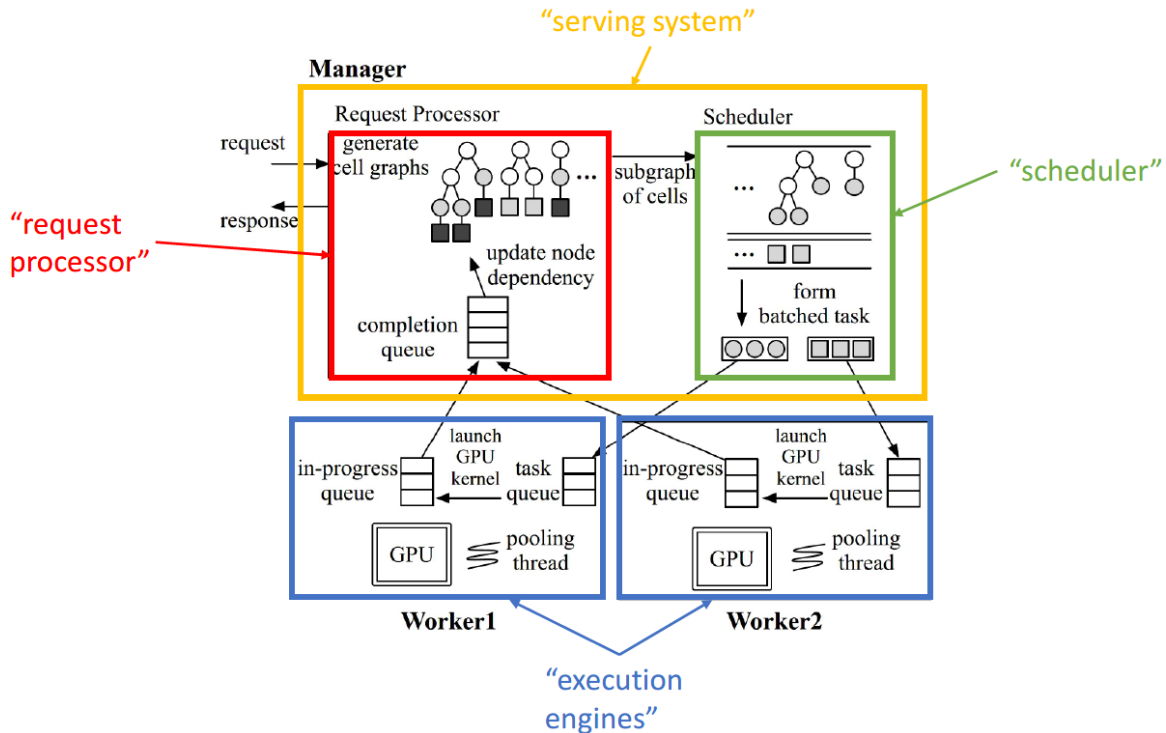
To the extent the preamble of claim 1 is limiting, it is taught by Gao. As explained with respect to the limitations below, Gao teaches a method of dynamically executing batches of requests on one or more execution engines. *See* Ex-1008, 532-533 (inventors of '775 acknowledging the same). *See* Ex-1002, ¶¶112-115.

Moreover, as discussed, a POSITA would have been motivated to combine Gao's teaching of dynamically executing batches of requests on one or more execution engines with Katharopoulos's teaching of a running a machine-learning transformer model. *See* Section (X)(B)(1). *See* Ex-1002, ¶¶116-119.

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;

Gao teaches a system architecture of a "Manager and Worker. The manager processes arriving requests and submits batched computation tasks to workers for execution." Ex-1004, 5. A color-coded annotation of Gao's Figure 6 below shows

how Gao's system reads on various claim limitations of the '775 Patent. *See* Ex-1002, ¶120.



Ex-1004, FIG. 6 (annotations added)³.

As shown in Figure 6, Gao includes a manager as a serving system that “processes arriving requests and submits batched computation tasks to workers for execution.” Ex-1004, 5. The manager in Gao 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.” Ex-1004, 5. *See* Ex-1002, ¶121.

³ Colorized annotations always added by Petitioner unless otherwise noted.

Gao 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.” Ex-1004, 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.” Ex-1004, 6. Gao also discloses the BatchMaker system uses a set of “decoders” as well as noting “decoders” are common for popular RNN models. Ex-1004, 11. *See* Ex-1002, ¶122.

As noted, Gao relates to RNN and not a machine-learning transformer model. However, a POSITA would understand Gao’s scheduling with dynamically batching cells would improve the latency associated with inference in using the machine-learning transformer model using a set of decoders, such as the linear transformer model taught by Katharopoulos. *See* Section (X)(B)(1) (motivation to combine discussion). *See* Ex-1002, ¶123.

Katharopoulos introduces a machine-learning transformer model that represents the transformer model as a Recurrent Neural Network (RNN). *See, e.g.*, Ex-1005, 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).”). *See* Ex-1002, ¶124.

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:

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

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

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

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

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

Ex-1004, 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.” Ex-1005, 5. *See* Ex-1002, ¶¶125-126.

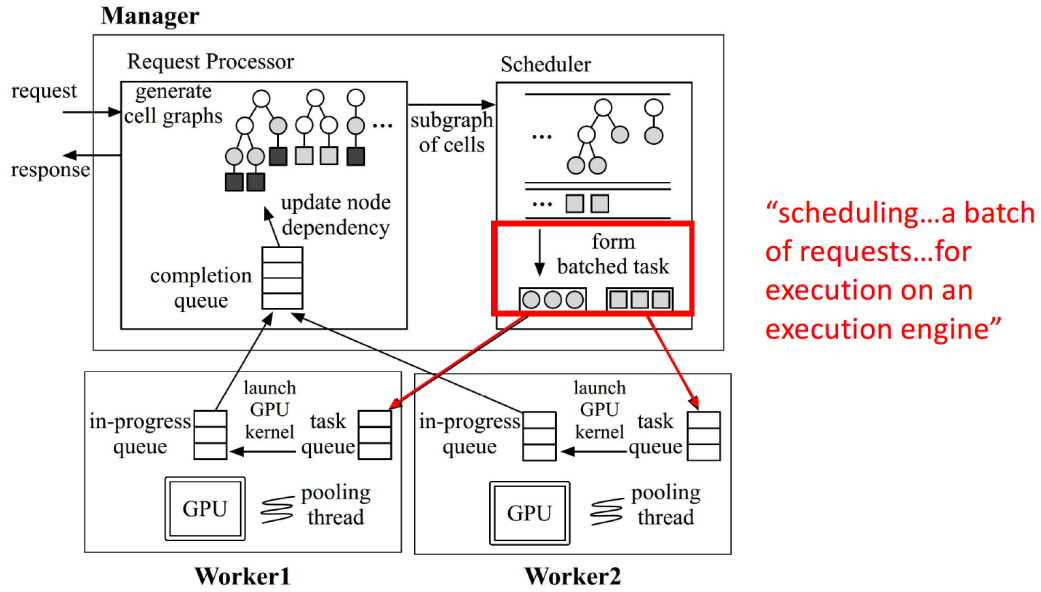
In view of the above, the combined teachings of Gao and Katharopoulos render obvious a system where Gao’s Workers (i.e., execution engines) each have access to a machine-learning transformer model including at least a set of decoders. *See* Section (X)(B)(1) (motivation to combine discussion). *See* Ex-1002, ¶127.

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

As discussed above, Gao teaches a system architecture with a “scheduler.” Ex-1004, 5. As can be seen from Figure 6, Gao includes a manager that “processes arriving requests and submits batched computation tasks to workers [i.e., execution

engines] for execution.” Ex-1004, 5. The manager in Gao 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.” Ex-1004, 5. *See* Ex-1002, ¶¶128-129.

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.” Ex-1004, 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.” Ex-1004, 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.” Ex-1004, 6. “Priority refers to the ability to prefer the execution of one type of cell over another.” Ex-1004, 6. *See* Ex-1002, ¶130.



Ex-1004, FIG. 6.

Gao also provides the basic algorithm employed by its BatchMaker system for scheduling and creation of batches. Ex-1004, 6. *See* Ex-1002, ¶131.

Algorithm 1: Scheduling and Batching Algorithm

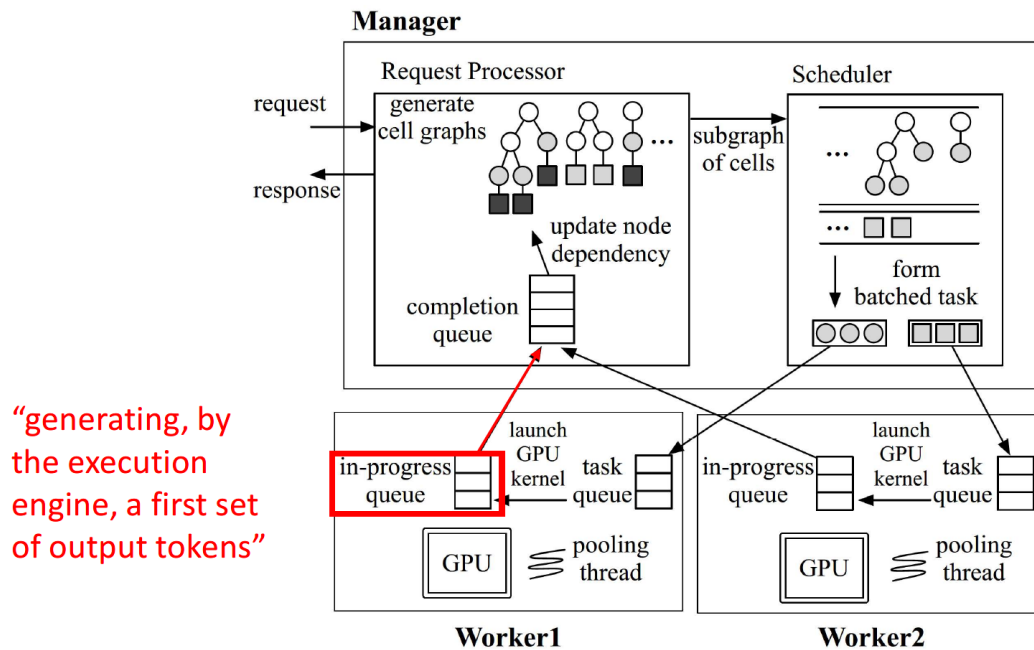
```
1 Bsizes: a set of supported batch sizes.
2 CellTypes: a set of cell types, each associated with a priority.
3 MaxTasksToSubmit: the maximum number of tasks that can
  be submitted to a worker.
4 def Schedule(worker):
5   S ← {ct ∈ CellTypes | ct.NumReadyNodes() ≥
     Bsizes.Max()};
6   if S.Size() = 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;
```

Ex-1004, 6.

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;

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.” Ex-1004, 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.” Ex-1004, 6. In addition, Gao notes that “each request can return to the user as soon as its last cell finishes.” Ex-1004, 2. *See* Ex-1002, ¶132.



Ex-1004, FIG. 6.

Gao also batches requests, which is illustrated in Gao’s cellular batching shown Figure 5 where requests 1-4 are initially batched together. *See* Ex-1002, ¶133.

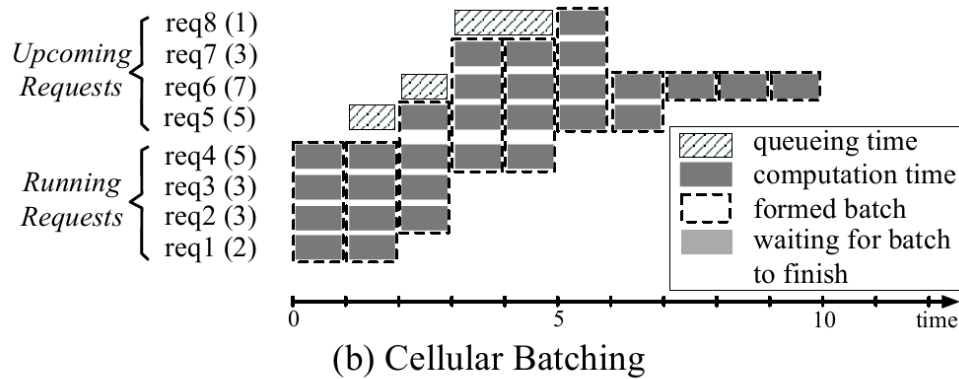


Figure 5: The timeline of graph batching and Cellular Batching when processing 8 requests from req1 to req8. The number shown in parenthesis is the request's sequence length, e.g. req1(2) means req1 has a sequence length of 2. Each row marks the lifetime of a request starting from its arrival time. Req1-4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

Ex-1004, FIG. 5.

Gao describes this 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 (4-way 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.

Ex-1004, 5.

Gao also provides an example of generating output tokens using a set of decoders using Sequence to Sequence (“Seq2Seq”), a widely used RNN model for machine translation. See Ex-1002, ¶¶134-140. 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.”

Ex-1004, 11.

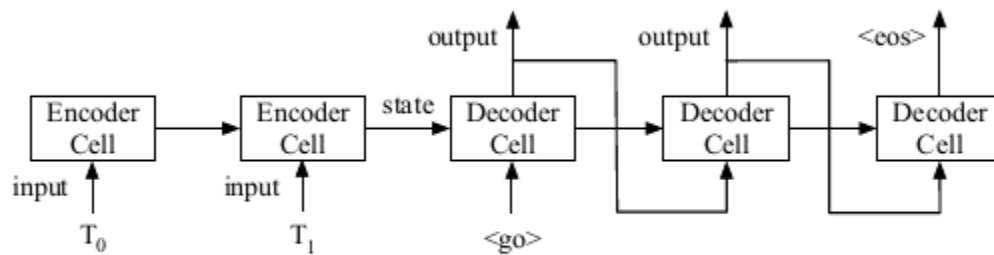


Figure 12: Seq2Seq with “feed previous” decoder

Ex-1004, FIG. 12.

In addition to Gao’s teaching to generate output tokens from batched requests, Katharopoulos introduces a machine-learning transformer model representing the standard transformer model as a Recurrent Neural Network (RNN). Ex-1005, 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).”). *See* Ex-1002, ¶141.

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. *See* Ex-1002, ¶142.

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

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

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

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

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

Ex-1004, 5.

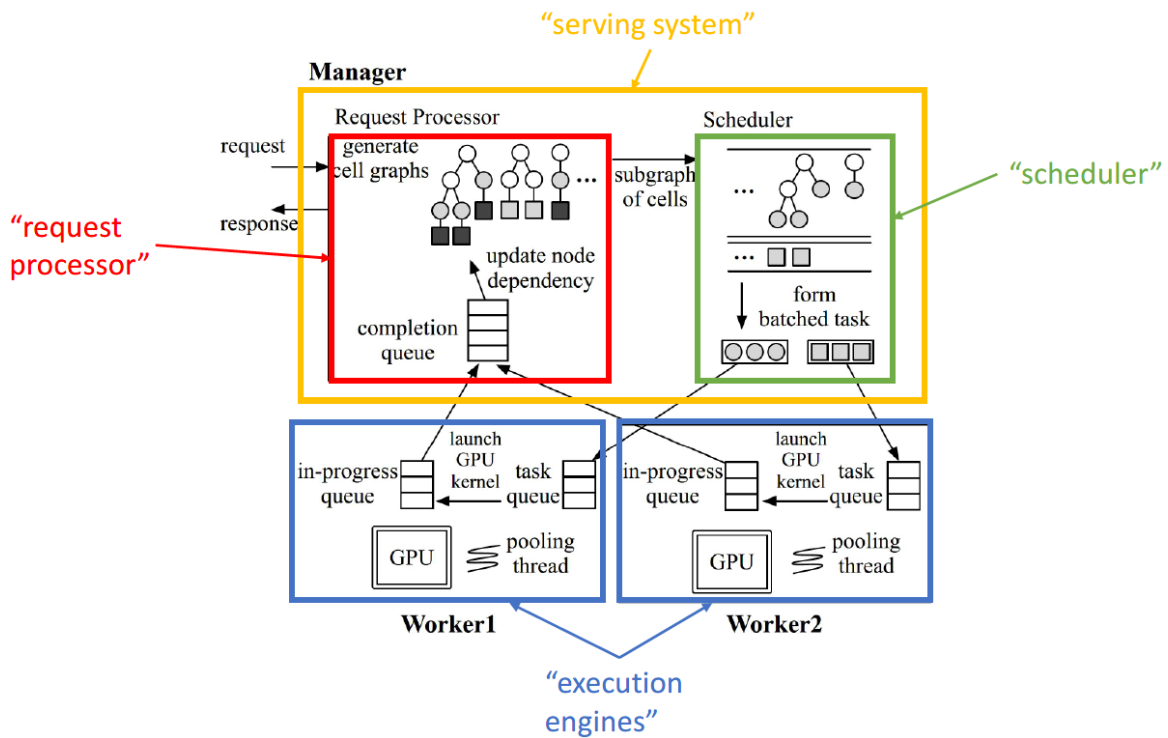
Importantly, Katharopoulos recognized that the “formulation does not impose any constrain on the feature function, and it can be used for representing any transformer model, in theory even the ones using softmax attention.” Ex-1005, 5. “A simple cell contains a few tensor operators... Two cells are of 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.” Ex-1005, 4. “For a cell to be

considered batchable, the first dimension of each of its input tensors should be the batch dimension.” Ex-1005 at 5. *See* Ex-1002, ¶143.

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. *See* Section (X)(B)(1) (discussing motivation to combine). *See* Ex-1002, ¶144.

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

Gao teaches a system architecture of a “Manager and Worker. The manager processes arriving requests and submits batched computation tasks to workers for execution.” Ex-1004, 5. The manager includes a request processor that receives new requests from a client device as indicated below. *See* Ex-1002, ¶¶145-146.



Ex-1004, FIG. 6.

Gao also teaches receiving new requests having a sequence of input tokens as seen in Figure 5 below. See Ex-1002, ¶147.

“new request
including a sequence
of input tokens”

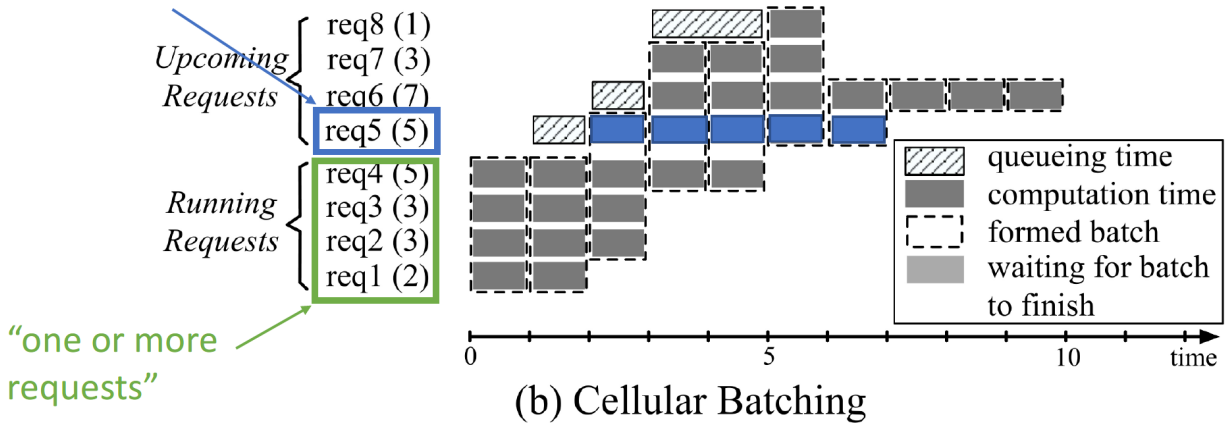


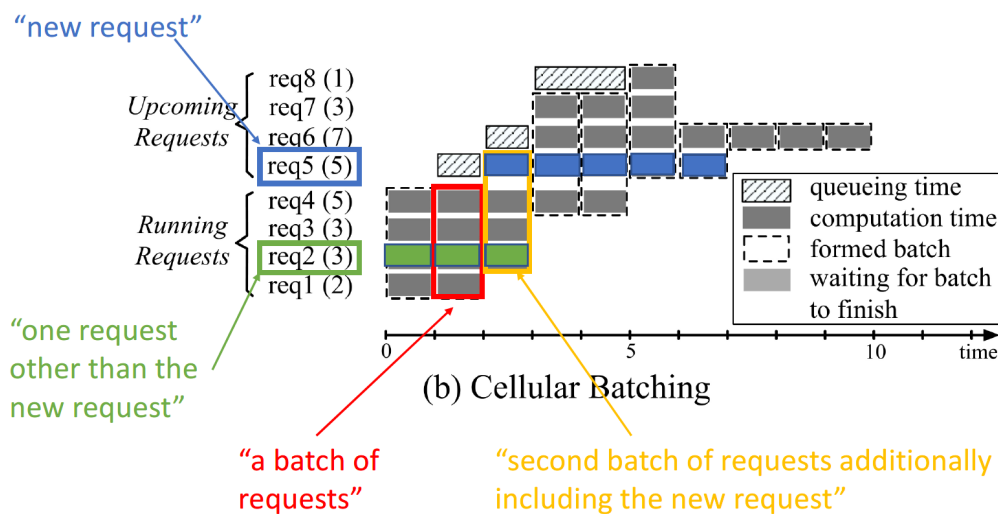
Figure 5: The timeline of graph batching and Cellular Batching when processing 8 requests from req1 to req8. The number shown in parenthesis is the request’s sequence length, e.g. req1(2) means req1 has a sequence length of 2. Each row marks the lifetime of a request starting from its arrival time. Req1-4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

Ex-1004, FIG. 5.

As can be seen in the above, a new request (e.g., request 5) includes five input tokens, indicated by the number in parenthesis, which is a different length than requests 1, 2, and 3. *See* Ex-1002, ¶148-151.

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

Gao teaches that as new requests are received, they are scheduled and moved to the execution engine. *See* Ex-1008, 532-533 ('775 inventors acknowledging that "BatchMaker allows a newly arrived request for RNN to join (or a finished request to leave) the current executing batch without waiting for the batch to completely finish.")). Additionally, the second batch of requests (e.g., batch at $t=3$) includes the new request for execution on the execution engine (e.g., request 5 with a sequence length of 5) that has a different length than one of the other request in the batch (e.g., request 2 with a sequence length of 3) as illustrated below. *See* Ex-1002, ¶¶152-158.



Ex-1004, FIG. 5.

When executing a request in an execution engine it would be obvious to a POSITA to first ensure that memory is available before execution. Gao makes clear that the system is configured to account for requests of varying lengths that are processed efficiently – regardless of their arrival time. Ex-1004, 2. Additionally, Gao

teaches that batches have allocated memory that depends on the batch size. Ex-1004, 8. See Ex-1002, ¶159.

Gao’s batches have allocated memory depending on their size so 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, as disclosed in Gao’s Algorithm 1 (below). See Ex-1002, ¶¶160-161.

```

24 def FormBatchedTask(ct, worker):
25     batch ← { };
26     for subgraph ∈ ct.subgraphs :
27         if subgraph.pinned ∈ {None, worker.id} :
28             for node ∈ subgraph’s ready nodes :
29                 batch ← batch ∪ {node};
30                 if batch.Size() = Bsizes.Max()
31                     return batch;
32     return batch;

```

Ex-1004, 6.

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, a POSITA understands that the scheduler taught by Gao is responsive to determining that the

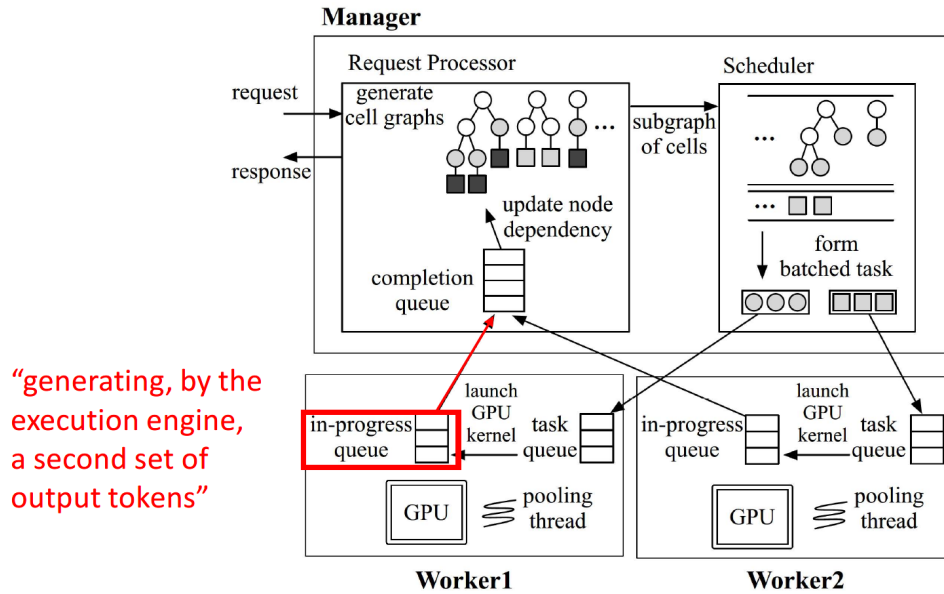
execution engine has memory availability to execute a request (whether new or old).

See Ex-1002, ¶162.

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.

This limitation is simply a second iteration of the first “generating” limitation except that the inputs have changed (i.e., the second set of inputs) and hence the outputs have changed (i.e., a second set of output tokens). Since Gao teaches receiving new request for execution it renders this limitation obvious for substantially the same reasons as the first “generating” limitation. Ex-1004, 5. Nonetheless, the full analysis of how the combination generates outputs is provided again below. *See* Ex-1002, ¶163.

As discussed above, Gao discloses an execution engine, namely workers, that use GPUs execute tasks and move finished tasks to the completion queue. Ex-1004, 6. In addition, Gao notes that “each request can return to the user as soon as its last cell finishes.” Ex-1004, 2. *See* Ex-1002, ¶159.



Ex-1004, FIG. 6.

Gao also batches requests, which is illustrated in Gao’s cellular batching shown Figure 5 where requests 1–4 are initially batched together. *See* Ex-1002, ¶164.

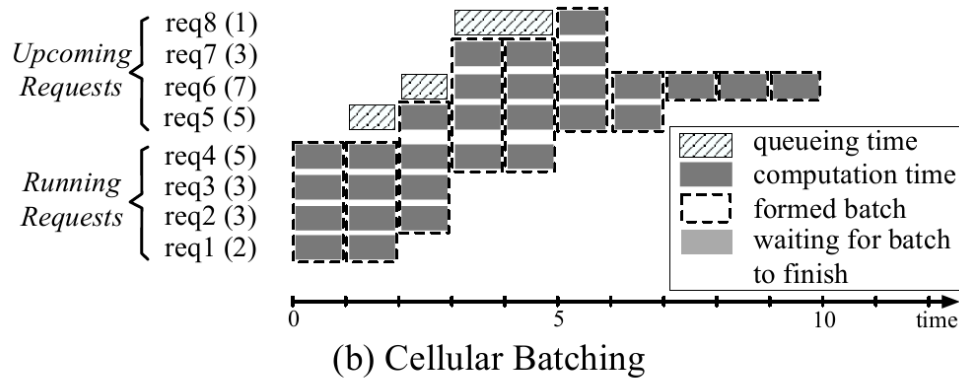


Figure 5: The timeline of graph batching and Cellular Batching when processing 8 requests from req1 to req8. The number shown in parenthesis is the request's sequence length, e.g. req1(2) means req1 has a sequence length of 2. Each row marks the lifetime of a request starting from its arrival time. Req1-4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

Ex-1004, 5 and FIG. 5.

Gao also provides an example of generating output tokens using a set of decoders using Sequence to Sequence (“Seq2Seq”), a widely used RNN model for machine translation. Ex-1004, 11. *See* Ex-1002, ¶¶165-170.

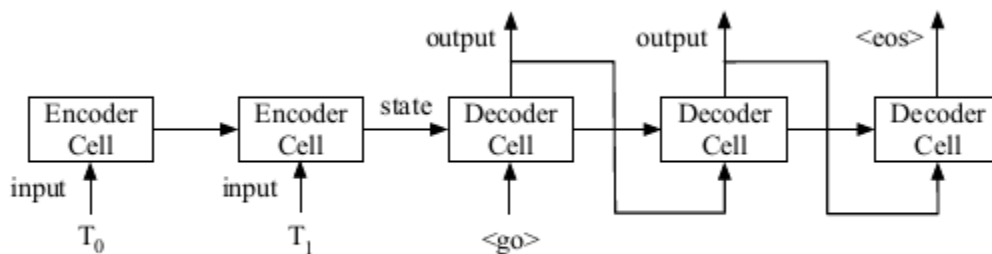


Figure 12: Seq2Seq with “feed previous” decoder

Ex-1004, FIG. 12.

In addition to Gao’s teaching to generate output tokens from batched requests, Katharopoulos introduces a machine-learning transformer model representing the standard transformer model as a Recurrent Neural Network (RNN). Ex-1005, 5. 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. *See* Ex-1002, ¶¶171-173.

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

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

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

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

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

Ex-1004, 5.

Accordingly, Katharopoulos teaches that a second set of output tokens (*e.g.*, “ y_i , the i -th output”) by applying the transformer model (*e.g.*, equation 20) to a second 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. *See* Section (X)(B)(1) (discussing motivation to combine). *See* Ex-1002, ¶174.

3. Claim 2

The method of claim 1, 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

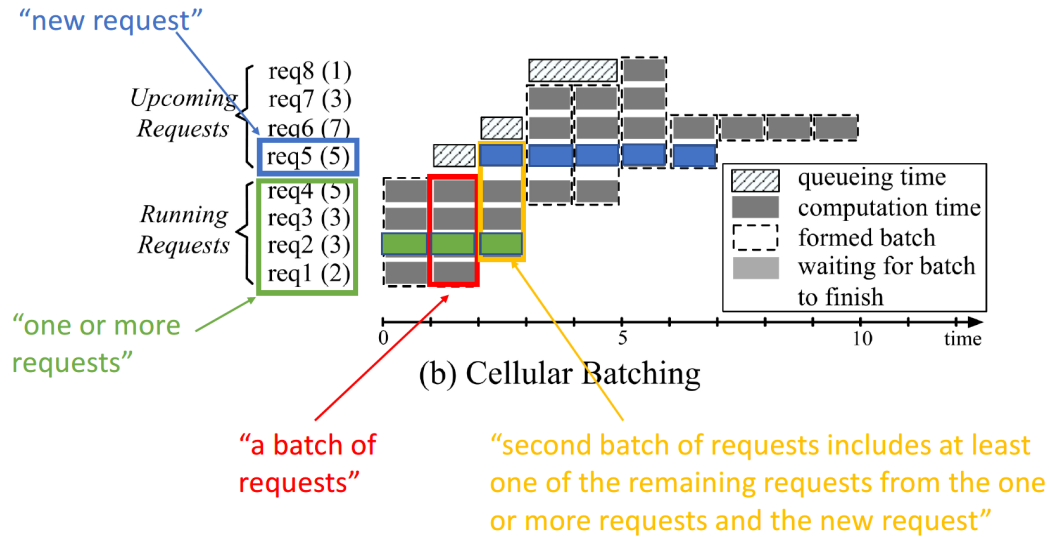
Gao teaches the behavior of cellular batching when being processed as shown in Figure 5, which has been previously described. Ex-1004, 6. In Figure 5, Gao shows that when a task finishes with a request completing execution (*e.g.*, $t=2$) it causes the completed request (*e.g.*, request 1) to leave the batching process. Ex-1004, FIG. 5. Accordingly, Gao teaches that a request will leave the batching process when it is determined the request is completed. *See* Ex-1008, 532-533 ('775 inventors acknowledging that Gao teaches this limitation)). *See* Ex-1002, ¶¶175-177.

In addition, Gao notes that “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.” Ex-1004, 2. As such, once the request leaves the batching process, the output tokens generated from the request are provided to a client device as a response to the request for the reasons previously provided for claim 1’s “generating” limitations. *See* Ex-1002, ¶¶178-181.

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

Similarly, Gao makes clear that 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. Ex-1004, 2. “As a result, a newly arrived request can immediately join the ongoing execution of existing requests, without needing to wait for them to

finish.” Ex-1004, 2. Gao’s Figure 5 further illustrates this by showing how a second batch (e.g., $t=2$) includes both remaining request 2 and new request 5. *See* Ex-1002, ¶¶182-184.



Ex-1004, FIG. 5.

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.

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. *See* Ex-1004, 5. As each request corresponds to a cell graph, the internal state associated with a specific request is continuously updated and stored. 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. *Id.*, 6–7. As such, Gao provides for a request’s internal state to be stored within a worker. *See* Ex-1002, ¶¶185-190.

Further, “[s]ince 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.” Ex-1004, 6. Thus, Gao provides for a request’s internal state to be stored in the GPU’s memory for further usage. *See* Ex-1002, ¶191.

Gao also teaches 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.”

Ex-1004, 2 (emphasis added). See Ex-1002, ¶192.

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. Ex-1004, 6. Thus, since batched requests have subgraphs pinned to a worker (executing on a GPU), and the nodes in a request subgraph have outputs, which include the interdependent cell hidden state, that are stored in contiguous GPU memory, then 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. Ex-1004, 6. See Ex-1002, ¶¶193-194.

According to Gao, and as detailed in Figure 5 (Ex-1004, 5) and Algorithm 1 (Ex-1004, 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. As stated by Gao:

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.

Ex-1004, 7. *See* Ex-1002, ¶195.

Gao further teaches that batches, along with their input and output tensors, have allocated memory that depends on the batch size. *See* Ex-1004, 8. Thus, Gao allocates GPU cache memory in the execution engine for storing the hidden internal states for respective batched requests, where the and the amount of GPU cache memory used for such storage must be allocated for each batch (depending on the batch size) during initializing. Gao also teaches the stored cell hidden state outputs corresponding to a batch of requests will correspond to a specific batch and will be updated when a task is completed. *See* Ex-1004, 6; Ex-1002, ¶¶196-198.

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 frees the GPU cache memory allocated for storing the cell hidden internal states. Ex-1004, 7. *See* Ex-1002, ¶199.

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.

As discussed, 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.” Ex-1004, 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.” Ex-1004, 6; *see also* Ex-1004, FIG. 5. *See* Ex-1002, ¶200.

As one example Gao notes that a “basic Seq2Seq model” where 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.” Ex-1004, 11. *See* Ex-1002, ¶201. An illustration of this process is shown below:

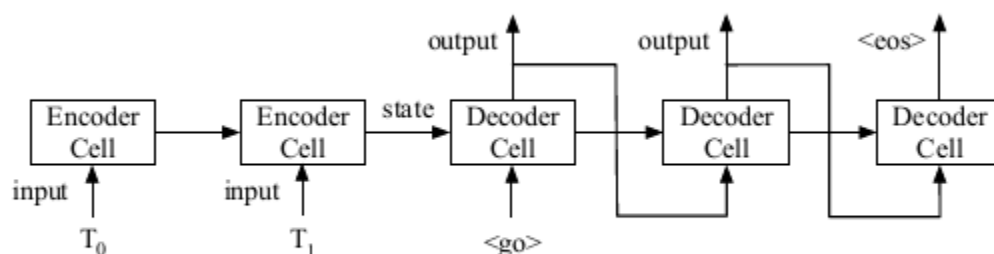


Figure 12: Seq2Seq with “feed previous” decoder

Ex-1004, FIG. 12. 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. *See* Ex-1002, ¶202-206.

Moreover, Gao teaches that the input tokens from new requests can be a sequence of more than one token (e.g., request 5 has an input length of five), which means the new request has a different length than “*the* output token” (singular) for the at least one request. *See* Ex-1004, FIG. 5. Notable, the antecedent basis for “the

output token” is “an output token.” Ex-1001, 29:35–39. Thus, the length of the sequence of input tokens needs to be more than one in order to have a different length than an “an output token.” *See* Ex-1002, ¶207.

Even if claim 4 recited an output token length greater than one, this limitation would still be obvious in view of Gao’s teaching that the output token length can longer than the input token length. Ex-1004, 11 (“Deployed systems typically configure the maximum decoding length to be the input sequence length plus a fixed threshold of extra steps.”). *See* Ex-1002, ¶207.

6. Claim 5

[A] The method of claim 4, further comprising: 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

As discussed with respect to claim 3 above, 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. *See* Section (X)(B)(4). 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. *See* Ex-1002, ¶208.

Gao further describes this process in the context of the BatchMaker’s scheduling and batching algorithm. Ex-1004, 6 (disclosing the basic algorithm employed by BatchMaker for scheduling and creating batches). *See* Ex-1002, ¶209.

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;

```

Ex-1004, 6 (Algorithm 1).

As shown above, “[o]nce 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.” Ex-1004, 7; *see also id.*, 6 (“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.”). Thus, Gao caches the data dependency or internal state for a request in the GPU’s cache memory. *See* Ex-1002, ¶¶210-213.

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 $\phi(K_j)V_j^T$ 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.” Ex-1005, 4. Katharopoulos further teaches the Transformer RNN has two internal states, which includes attention memory and normalizer memory. Ex-1005, 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.” *See* Ex-1002, ¶214.

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

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

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

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

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

Ex-1005, 5.

Thus, a POSITA would have understood that Katharopoulos teaches 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 Trasnformer 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). *See* Ex-1002, ¶215.

Gao further teaches, “[d]uring 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].” Ex-1004, 8. *See* Ex-1002, ¶216.

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. *See* Ex-1002, ¶217.

Thus, Gao teaches that a request's internal state is stored in cache memory for further usage, a POSITA would have understood that that Katharopoulos teaches storing Key and Value tensors (matrix) in the execution engine's cache memory for the combined Gao and Katharopoulos system. *See* Ex-1002, ¶218.

[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.

According to Gao, different requests join different batches of different sizes. (Ex-1004 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.” Ex-1004, 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.” Ex-1005, 5–6.

See Ex-1002, ¶219.

Further, “[d]uring 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].” Ex-1004, 8. *See* Ex-1002, ¶220.

Thus, 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. *See* Ex-1002, ¶221.

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

As discussed above, Gao provides a basic algorithm employed by BatchMaker for scheduling and creation of batches. Ex-1004, 6. *See* Ex-1002, ¶222.

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;

```

Ex-1004, 6 (Algorithm 1).

Gao further 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, p. 5 (discussing that “Each type of cell has a desired maximum batch size, which is determined through offline benchmarking.”); *See also* Ex-1004, 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.”). *See* Ex-1002, ¶223.

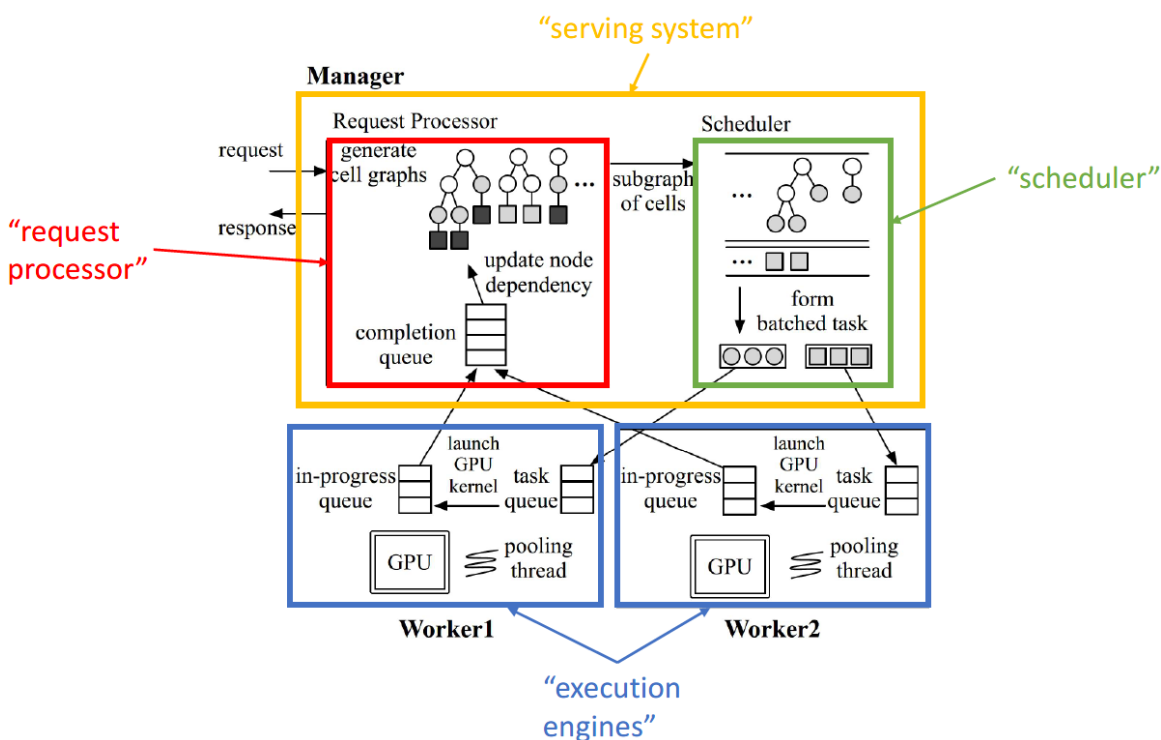
Further, batches of different sizes, along with their input and output tensors, have allocated memory that depends on the batch size. Specifically, Gao provides that “BatchMaker aims to materialize all the cells for each supported batch size on every available GPU device...[which] requires knowledge 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.” Ex-1004, 8 (emphasis added). *See* Ex-1002, ¶224.

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. *See* Ex-1002, ¶225.

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.

As discussed for claim 1, teaches the scheduler prepares batches of requests to send for execution on different Workers, including Worker 1 (i.e., “execution engine”) and Worker 2 (i.e., “a second execution engine different from the execution engine”). Ex-1004, 5. *See* Ex-1002, ¶¶120-123; 226.



Ex-1004, FIG. 6.

When executing a request in an execution engine it would be obvious to a POSITA to first ensure that memory is available before execution. Gao makes clear that the system is configured to account for requests of varying lengths that are processed efficiently – regardless of their arrival time. Ex-1004, 2. Additionally, Gao teaches that batches have allocated memory that depends on the batch size. Ex-1004, 8. See Ex-1002, ¶¶227-228.

Since Gao’s 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, as disclosed in Gao’s Algorithm 1. See Ex-1002, ¶¶154-155; 228.

```

24 def FormBatchedTask(ct, worker):
25     batch  $\leftarrow$  { };
26     for subgraph  $\in$  ct.subgraphs :
27         if subgraph.pinned  $\in$  {None, worker.id} :
28             for node  $\in$  subgraph's ready nodes :
29                 batch  $\leftarrow$  batch  $\cup$  {node};
30                 if batch.Size() = Bsizes.Max()
31                     return batch;
32     return batch;
```

Ex-1004, 6.

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, a POSITA understands that the scheduler taught by Gao is responsive to determining that the execution engine has memory availability to execute a request (whether new or old). *See* Ex-1002, ¶162.

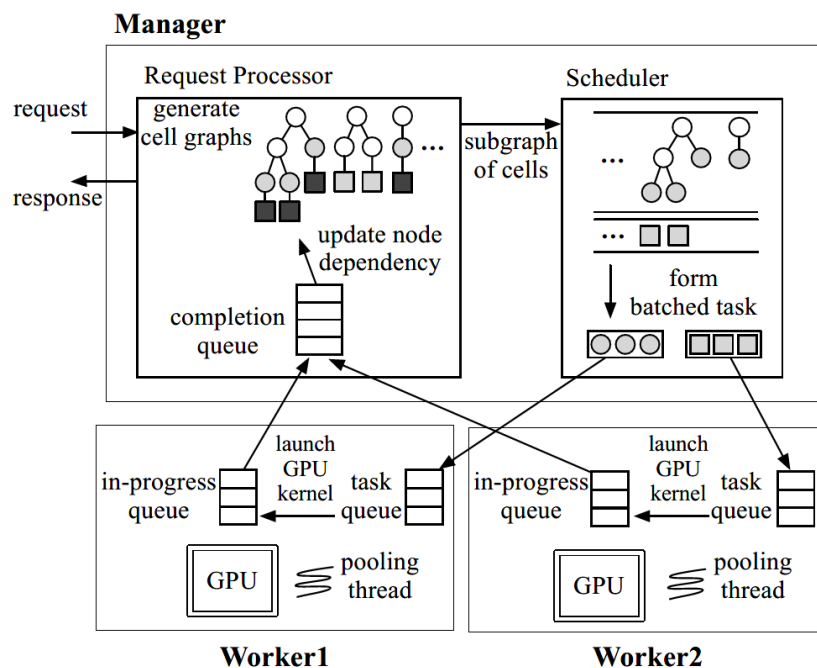
In addition, Gao also teaches the scheduler will be invoked when a “worker becomes idle.” Ex-1004, 6. Thus, when multiple Workers are used, a new request can be executed through a first worker running on an execution engine responding to memory availability, otherwise, if there is insufficient memory available on the first worker then 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. *See* Ex-1002, ¶227.

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).

As previously discussed Gao teaches the Workers are execution engines. Ex-1004, 5. Each worker is configured as a graphical processing unit (GPU) for executing tasks. Ex-1004, 5 (“When a task has batched sufficiently many cells, it is submitted to a GPU device for execution.”); *see also id.* (“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.”). *See* Ex-1002, ¶229.

The use of the Workers as GPUs is also illustrated in Figure 6 of Gao, below.



Ex-1005, FIG. 6. *See* Ex-1002, ¶229-232.

10. Claim 9

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

Gao teaches a RNN is a Deep Neural Network (DNN) architecture designed to process sequential data of variable length – often used for language processing applications. Ex-1004, 2. RNNs help solve tasks such as predicting which words would follow in a sentence. Ex-1004, 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.” Ex-1004, 2 and FIG. 1. *See* Ex-1002, ¶233.

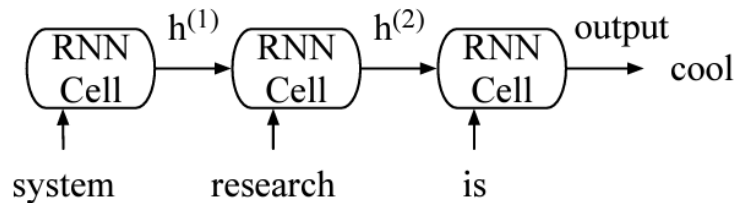


Figure 1: An unfolded chain-structured RNN. All RNN Cells in the chain share the same parameter weights.

Ex-1004, 2.

Accordingly, Gao teaches 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. Indeed, Katharopoulos also teaches that transformer models are operable with text such as natural language processing. Ex-1005, 1. *See* Ex-1002, ¶234.

11. Claims 10-18

Claims 10-18 are substantively the same as claims 1-9 and is unpatentable for the same reasons as taught by the combination of Gao and Katharopoulos with respect to claim 1. The only difference in claims 10-18 from claims 1-9 is that claims 10-18 recites the limitations are performed by a “non-transitory computer-readable storage medium storing computer program.” *See* Ex-1022 (claim appendix showing differences in claim 1-9 to 10-18); *see also* Ex-1010, 191 (applicant stating during prosecution that “Claim 10 recites similar features as claim 1. Thus, claim 10 and its dependent claims...” are distinguishable over the prior art for the same reasons as claim 1). *See* Ex-1002, ¶¶235-244.

Gao and Katharopoulos both disclose their systems are computer programs executed on non-transitory computer-readable storage medium. *See* Section (X)(B). 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. Ex-1004, 5. In addition, both Gao and Katharopoulos disclose algorithms which is executable code that runs on non-transitory computer-readable storage medium. Ex-1004, 6; Ex-1005, 5; *see* Ex-1002, ¶236.

Therefore, claims 10-18 are unpatentable for the same reasons as claims 1-9 since both Gao and Katharopoulos teach their systems computer programs executed

on non-transitory computer-readable storage medium in addition to all of the other limitations of claims 10-18. *See* Ex-1002, ¶¶235-244.

XI. DISCRETIONARY DENIAL IS NOT SUPPORTED

A. 35 U.S.C. § 325(d)

The Board should not exercise discretionary denial under § 325(d), particularly in view of the new prior art presented and strong showing of invalidity in the foregoing Grounds.

First, Katharopoulos has not been previously presented to the Office. While Gao, is cited in an information disclosure statement during prosecution along with numerous other references, but was not used to reject the claims in any office action and the Examiner did not otherwise provide any comments about Gao's teachings during the prosecution of the '775 Patent. Nonetheless, since every proposed Ground of rejection in this IPR relies on a reference never previously presented to the Office, the arguments presented in this IPR have not been previously considered by the Office. In addition, Petitioner provides the testimony of Dr. Hammouri to explain how a POSITA would view the teachings of these references. *See* Ex-1002. Therefore, part one of the *Advanced Bionics* test favors institution, as the combination set forth herein was not previously considered by the Office.

Further, under part two of *Advanced Bionics*, assuming the examiner was aware of the relevant teachings in the cited prior art, the examiner erred in a manner

material to the patentability of challenged claims in allowing the claims of the '775 Patent. As discussed with respect to the prosecution history, Applicants argued that none of the cited prior art taught the claimed scheduling. Ex-1010, 190–191. However, the inventors of the '775 Patent have acknowledged that Gao taught this claimed scheduling feature. Ex-1008, 532–533. This admission in the Orca paper was never submitted to the Examiner.

The only other point of distinguishment identified during prosecution was applying this claimed scheduling to a transformer model. Yet, numerous variations of machine learned transformer models existed by the time of the alleged invention on Dec. 3, 2021, including those taught by Katharopoulos. Applying the scheduling features taught by Gao to the transformer models taught by Katharopoulos would have been obvious to a POSITA as explained in this petition. Thus, the examiner materially erred in allowed the '775 Patent claims.

B. 35 U.S.C. § 314(a)

Apple, Inc. v. Fintiv, Inc., IPR2020-00019, Paper 11 (Mar. 20, 2020) (precedential) favors institution because this Petition presents compelling evidence of unpatentability. Furthermore, Petitioner stipulates that if the Board institutes this IPR then Petitioner will not to pursue in a parallel district court proceeding the same grounds as in this Petition or any grounds that could have reasonably been raised in this Petition. *See Sotera Wireless, Inc. v. Masimo Corp.*, IPR2020-01019, Paper 12,

17 (PTAB Dec. 1, 2020) (precedential); *see* Director’s Interim Guidance Memo, 7 (“the PTAB will not discretionarily deny institution of an IPR or PGR” in view of such a stipulation).⁴

XII. CONCLUSION

The Board should institute this IPR proceeding and find that claim 1-18 are unpatentable.

Dated July 28, 2024:

Respectfully submitted,
/James P. Murphy/
James P. Murphy
Reg. No. 55,474
Polsinelli PC
1000 Louisiana Street
Suite 6400
Houston, Texas 77002
Tel: (713) 374-1600

Adam P. Daniels
Reg. No. 66,681
Polsinelli LLP
2049 Century Park E.
Suite 2900
Los Angeles, CA 90067
Tel: (310) 556-6754

Attorneys for Petitioner

⁴ Available at:

https://www.uspto.gov/sites/default/files/documents/interim_proc_discretionary_denials_aia_parallel_district_court_litigation_memo_20220621_.pdf.

CERTIFICATE OF SERVICE UNDER 37 CFR § 42.6(e)(4)

The undersigned hereby certifies that a copy of the accompanying Petition for *inter partes* review, all accompanying exhibits, and the Power of Attorney is being served via Federal Express to the correspondence address of record for U.S. Patent No. 11,442,775 on July 28, 2024 upon the following:

Jae Won Song
Fenwick & West LLP
Silicon Valley Center
801 California Street
Mountain View, CA 94041

/Ericka McNeil/
Ericka McNeil

Polsinelli, PC
1000 Louisiana Street, Suite 6400
Houston, Texas 77002
Tele: (713) 374-1600