

Honours Individual Project Dissertation

# Neural Search on Modern Consumer Devices

Ben Brown March 2023

## Abstract

Mobile phones have had different kinds of processor architecture due to power constraints, and many of these developments from this different architecture, such as neural accelerators and unified memory, are now starting to appear in desktop computers. As desktops support a complete development environment, they allow us to investigate if these hardware developments can improve the efficiency of neural information retrieval systems. If these techniques are sufficiently efficient, they could improve search engines on user devices, such as systems that let users find local files. We find that neural accelerators can be used to improve the speed of encoding queries using language models. Still, the performance depends on the model's properties, so it does not improve on the performance of running on the GPU when encoding a large number of passages at once. For scoring encoded queries against the large index matrices, we find that the neural accelerator performs worse than the CPU and GPU. We also find that copying memory for GPU uses takes a long time, but currently, unified memory does not have software support to stop these time-consuming copies. Given these findings, neural retrieval methods are likely not efficient enough for consumer devices, but if unified memory gains software support, this will likely change.

## **Education Use Consent**

I hereby grant my permission for this project to be stored, distributed and shown to other University of Glasgow students and staff for educational purposes. Please note that you are under no obligation to sign this declaration, but doing so would help future students.

Signature: Ben Brown Date: March 2023

## Contents

1	Intr	oducti	on	1
	1.1	Why	on user's devices?	1
	1.2	Neura	al Information Retrieval	1
	1.3	Aims		2
	1.4	Struct	ture	2
2	Neu	3		
	2.1	Backs	ground	3
		2.1.1	Neural Accelerators	3
		2.1.2	Neural Networks on Mobile Devices	3
		2.1.3	Compilation for Neural Accelerators	4
		2.1.4	Accelerators for Information Retrieval Tasks	5
	2.2	Meth	od	5
		2.2.1	Query and Document Encoding	5
		2.2.2	Document Scoring	5
	2.3	Resul	ts	6
		2.3.1	Experimental Environment	6
		2.3.2	Query and Document Encoding	7
		2.3.3	Document Scoring	8
		2.3.4	Effectiveness of Compiled Models	10
	2.4	4 Discussion		11
		2.4.1	Query Encoding	11
		2.4.2	Document Encoding for Indexes	11
		2.4.3	Document Scoring	12
	2.5	5 Summary		12
3	3 Unified Memory			
	3.1	Backg	ground	13
		3.1.1	Unified Memory Architecture	13
		3.1.2	NVIDIA Unified Memory	14
		3.1.3	Memory Mapping	14
	3.2	2 Method		15
	3.3	3 Implementation		15
		3.3.1	PyTorch	15
		3.3.2	Tensorflow	16
	3.4	Theo	retical Results	16
	3.5	Discu	ssion	16

	3.6	Summary	17		
4	4 Conclusion		18		
	4.1	Summary	18		
	4.2	Future work	19		
	4.3	Reflections	19		
Bi	Bibliography				

## 1 Introduction

Many recent approaches in information retrieval (Xiong et al. 2020; Khattab and Zaharia 2020) use a dense neural network approach rather than the traditional sparse approach using inverted indexes. These new neural network approaches are much more compute-intensive than previous approaches, so they are usually only used with powerful servers.

Mobile phones typically have a different processor architecture than standard laptops and desktops to be more power efficient. Mobile processors tend to be a System on a Chip (SoC) design, where the CPU, GPU, and memory are all on the same chip. This design allows the different processor parts to have a unified memory bank, allowing all access to the same memory and stopping memory having to be copied between parts. In recent years, some mobile phones have also featured neural accelerators, dedicated parts of the processor designed for running neural networks efficiently. Neural accelerators have been around for a few years but are uncommon outside of mobile phones and specialised servers for machine learning.

Apple's recent desktops/ laptops (*Apple unleashes M1* 2020) with their in-house designed chips have brought the low-power design from phones to "full" devices. These new devices have a systemon-a-chip design and a neural accelerator. Unlike mobiles, these devices run a standard desktop operating system that supports common software development tools. This easier development environment and the increased power from being larger devices allow us to investigate if this new architecture can yield performance improvements for neural information retrieval tasks.

## 1.1 Why on user's devices?

Running neural networks on users' devices can bring many benefits, such as reduced power usage, as they can run on specialised low-power hardware in devices rather than being sent over power-heavy wireless networks to large GPUs in the cloud. The time taken for the user to get results may also be shorter as no network transfer is required around the globe, and there is no wait for compute space on a server. It also means sensitive data does not need to leave the user's device, protecting the users' sensitive data and privacy. Not having to transfer sensitive data encryption for transit does not need to be considered, and it allows neural networks to be used with data, such as medical records, which cannot be transferred.

Users' phones and devices have an increasing amount of data available to access. It is hard to remember where everything is, so search allows people to get the information they need. There are many scenarios where a user may want to search for information on their device, such as to find a contact, specific emails, or an app.

## 1.2 Neural Information Retrieval

Traditional sparse information retrieval can be used to search text documents by creating an inverted index which maps each term to all the positions where it occurs. This inverted index is then used to score each document in a corpus against a query to produce a ranking of documents to display.

Modern dense retrieval techniques take a different approach, using neural network language models to produce dense vector representations (embeddings) of each document in the corpus. The index for dense retrieval is a large matrix containing all the vectors for all the documents. The query is also encoded using the language model to produce a vector, which is then compared against the index of all the vector embeddings for documents in the corpus. The comparison gives a score used to rank the documents, and a standard vector similarity measure such as cosine similarity is often used. Images and other formats can also be encoded into a vector embedding to be searched simultaneously, which is impossible with sparse retrieval. This report predominately looks at textual retrieval tasks. Many approaches perform sparse retrieval before the slower dense retrieval to re-rank a smaller number of documents.

### 1.3 Aims

This project examines if new processors designed for personal devices allow neural network-based information retrieval techniques to be used locally. We break the overall research question into several sub-questions:

- Can neural accelerators be used to speed up the encoding of queries?
- Can neural accelerators be used to speed up the encoding of documents for creating indexes?
- Can neural accelerators be used to speed up the scoring of documents during retrieval?
- Can a unified memory architecture improve the performance of retrieval systems by reducing data copies?

We investigate these questions using an Apple Studio with an Apple M1 Max chip and 32GB memory. This machine features both a unified memory architecture and a neural accelerator. It is a high-end machine and is more powerful than a typical consumer device, but it allows us to see the possible performance of this new architecture.

## 1.4 Structure

We split the body of this report into two main sections.

In chapter 2, we introduce neural accelerators and previous work exploring accelerators for information retrieval. We then investigate if language models can be run on a neural accelerator to speed up query and document encoding and if running query-index scoring on the neural engine improves efficiency.

In chapter 3, we introduce unified memory and investigate if it can be used to improve retrieval performance by reducing the time spent copying data.

# 2 Neural Accelerators

In this chapter, we investigate if neural accelerators can improve the efficiency of dense information retrieval systems. We examine if the language models used to encode documents and queries can be run on a neural accelerator to improve retrieval performance. We also look to see if the scoring of documents can be performed on the neural accelerator to improve the performance.

## 2.1 Background

We introduce neural accelerators, neural networks on mobile devices, and previous work on hardware accelerators for information retrieval.

#### 2.1.1 Neural Accelerators

Neural networks are now used for various tasks, including speech recognition, image recognition, and search. In recent years, mobile phone manufacturers have added specific cores to efficiently run neural networks. These cores are application-specific integrated circuits (ASICs) that are limited in what instructions they can compute but highly optimised for the operations they can perform. They are known by various names by different manufacturers; Google calls them Tensor Processing Units (TPUs), whereas Apple calls them Neural Engines (ANEs).

Similar to the way that GPUs improve performance for specific graphics tasks compared to CPUs but are not very useful for other tasks, neural accelerators improve the performance for running neural networks but are hard to use for general computation.

As neural networks' primary operations are repeated operations (such as multiplication) on large matrices of floating point numbers, many neural accelerators use reduced precision numbers to speed up computation. Reducing single-precision (float32) to half-precision (float16) allows operations to be performed quicker and reduces the memory needed, enabling more simultaneous work. However, this comes with the downside of lower precision, which can alter the results obtained.

#### 2.1.2 Neural Networks on Mobile Devices

Ignatov et al. (2018) investigated running neural networks on various Android phones with a wide range of different processors. They investigated various image processing tasks, such as image recognition and enhancement, using pre-trained concurrent neural networks running on the phones. Some of their tests run fully with hardware acceleration on the devices, whereas others fall back to running on the main CPU. They measured the time it took to run each task and combined them to get an overall score for each device. Despite benefits, there must be a careful balance between a user's battery life and the extra computation on the device. If it is constantly running computations to recompute indexes and train models, any power benefits could be removed.

The large size of neural networks is a problem for running models on mobile devices. Liu et al. (2019) could not run models such as BERT (Devlin et al. 2019) in their testing as they were

too large to be trained on the device used. Not training the models on consumer devices is not an issue for information retrieval, as a generic language model can be used, and a new model does not need to be trained for each user. Using the same model, representations could be easily shared between users.

Servers will always have more memory than users' portable devices, so models must be adapted to run efficiently on a different type of device and its constraints. They also found that using memory consumed more power than the GPU performing the calculations. Molina et al. (2021) and Ignatov et al. (2018) both identified that reducing the representation size of feature vectors could increase the number of requests processed at once, thus improving the system's latency. Although some of the optimisations halved the execution time, the accuracy of the model is reduced by roughly 1–2%. Memory has been identified to be the largest difficulty of neural models on consumer devices.

#### 2.1.3 Compilation for Neural Accelerators

The only way to use the neural engine in the Mac Studio is to compile models from machine learning frameworks like PyTorch and Tensorflow using Apple's CoreMLTools (*Core ML Model Format Specification — Core ML Format Reference documentation* n.d.). The compilation transforms the models into a format that can be run without a full development stack that frameworks such as PyTorch need, so they can be run on devices where such stacks are not available, such as on mobile devices. Historically, mobile phones and tablets have been the main devices with neural accelerators; however, laptops and desktops, such as those powered by Apple's M-class processors, have neural accelerators and can support the full development stack.

Compilation can bring many other benefits, such as optimising models for the available hardware and making assumptions to increase performance and reduce memory usage. Compilation can also mask differences between different platforms so models always act the same to consumers. However, there are various downsides to compilation, including the lack of control, the removal of flexibility in the models, and the compiler only supporting a subset of the very flexible frameworks.

Openja et al. (2022) investigated the performance of models once compiled using CoreMLTools (*Core ML Tools* n.d.) and ONNX (*ONNX* | *Home* n.d.) compilers. They found that the accuracy of the compiled models was comparable to the accuracy of the original models, though CoreMLTools compiled models sometimes gave the wrong output in classification tasks. Models compiled with ONNX do not have classification errors. As changes are made to the model's layout during compilation, the results will differ due to numerical instability from floating point numbers.

They also found that the compilation did not lead to consistent speedups. For some models, the compiled versions performed better, whereas the compiled version performed considerably worse for others. Most of the models tested performed worse when compiled with CoreMLTools. The speedups are also inconsistent between CoreMLTools and ONNX, which shows that it is highly dependent on the compiler. CoreMLTools drastically reduces the size of models from Keras when compiled but has almost no effect on models from PyTorch. They ran their benchmarks on a Macbook Air with a 1.6 GHz Dual-Core Intel Core i5 CPU with an integrated graphics chip but do not mention if they used the GPU when running the compiled models.

Kasperek et al. (2022) found that the compiled models perform better on Apple's new processors than the previous generation of devices using Intel chips. They claim the performance increase is due to the neural accelerator in the new chips but do not compare the performance of the CPU and GPU without the neural accelerator. They mainly look at the performance of image classifiers rather than language models.

#### 2.1.4 Accelerators for Information Retrieval Tasks

Molina et al. (2021) looked at using Field Programmable Gate Arrays (FPGAs) on Systems on Chips (SoCs) to run decision tree ML algorithms. This allows them to design dedicated circuits for various tasks much quicker than getting custom silicon designed and manufactured. They found that their architecture could process more input vectors (up to the saturation of the device's memory) in a constant execution time. This doesn't occur when using the CPU.

Gil-Costa et al. (2022) addressed the memory problems of learn-to-rank models running on SoC-FPGA systems by using binning and quantisation to shrink the models and feature vectors. By shrinking the model's size, several inference tasks can be run at once on the device, and the time taken to copy the vectors to the device is reduced. They reduced the required memory by three-quarters without reducing the prediction accuracy. The execution time was similar to a high-end CPU (on a low-cost device) but used a magnitude less power. Binning was found to have no impact on effectiveness, only quantisation. When quantised to 8 bits, there is no performance penalty, but when reduced to 4 bits, there is a noticeable drop in the model's performance.

The approach of an ultra-specialised circuit for a task should lead to better results, as the design can be tailored to the exact computations that need to occur. This can mean better hardware utilisation, power efficiency and potentially more parallelisation. Similar to how a GPU is faster for specific tasks than a CPU, a specialised chip for information retrieval should be quicker but not worth the cost for a typical consumer task as it would not have enough use.

No papers were found on evaluating the effectiveness or efficiency of neural search models on consumer-grade accelerators.

## 2.2 Method

We introduce our experiments to investigate whether a neural accelerator can improve the performance of document and query encoding and document scoring.

#### 2.2.1 Query and Document Encoding

We investigate if language models can be run on the ANE (Apple Neural Engine) to improve the efficiency of query transformation at query time and for the production of indexes.

Apple published a guide on optimising transformer models running on the ANE (*Deploying Transformers on the Apple Neural Engine* 2022). They provide an optimised DistilBERT model (Sanh et al. 2020) to show the possible benefits of the optimisations. They claim that their optimisations led to the model running ten times faster and using 14 times less memory. They claimed latency for inference of the optimised model is comparable to optimised ASIC server hardware. They do not provide any results to show the optimised model has similar effectiveness as the original.

We compile the model using CoreMLTools and examine the throughput and latency of creating embeddings of passages. Throughput is important for encoding documents for an index as we have many documents that we want to process quickly. Latency is important at query time as we want to encode the query as quickly as possible so it can be compared against the index.

We also look at a standard BERT model without any modifications to see how it compares.

#### 2.2.2 Document Scoring

We investigate if the ANE can be used to speed up the scoring of documents when given a query.

**Matrix Multiplication** Once a query has been transformed into an embedding, a matrix multiplication is used to compare it with the documents in the index. The critical operation in a neural network is usually matrix multiplication. As the ANE is designed to improve the running of neural networks, we investigated if the ANE could be used to improve the efficiency of computing the similarity scores.

We compile a PyTorch model with CoreMLTools, which multiplies two tensors, and we examine how it performs relative to the original model.

**Convolutions** Apple is very secretive about the designs of the ANE and provides few details about how it works. There is an internal interface, "AppleH11ANEInterface", but it is not available for any public software. A couple of people (Wu 2021; Hotz n.d.) have been able to reverse-engineer parts of the interface and work out how parts of the system work. From this reverse engineering work, some people speculate (Fee 2021) that the ANE has been optimised as a convolution accelerator rather than for general-purpose neural networks. In their marketing materials, when the first ANE was released (*The future is here: iPhone X* 2017), Apple highlighted facial recognition, augmented reality and image processing as key features enabled by the ANE, all of which are tasks which often use convolutions. An ex-Apple machine learning engineer (Wang et al. n.d.) also predicted that the main use of the ANE would be for future virtual reality features.

A convolution is a repeated multiplication and accumulation of two tensors. It is typically a smaller tensor multiplied multiple times with various parts of a larger tensor. Each time, the smaller tensor is moved across the larger one by a stride. If we set the smaller tensor to have a singular dimension of the correct size and set the stride to the correct value, the convolution can act as a matrix-vector multiplication.

We compiled a PyTorch network, which consisted of a singular 1D convolution layer, which would be the equivalent of multiplying the query vector against the documents.

We compare the results of the compiled model running with all components of the processor (ANE, GPU, and CPU), without the ANE (GPU and CPU), and just the CPU, as well as the original model running on the GPU and CPU.

To ensure that the compilation does not affect the convolutions' results or effectiveness, we use PyTerrier (Macdonald and Tonellotto 2020) to calculate the effectiveness metrics of a retrieval pipeline using the convolutions. We transform the queries and documents using TCT-Colbert (Lin et al. 2020), and use the compiled convolution to score the documents, rather than the standard matrix multipication.

## 2.3 Results

We report our experimental results, showing the performance of compiled models for the document encoding and scoring tasks.

#### 2.3.1 Experimental Environment

All experiments are performed on a Mac Studio with an Apple M1 Max chip and 32GB memory.

CoreMLTools is deeply integrated with MacOS, and some features are tied to the operating system's version. MacOS 13 brought full support for float16 inputs to models, even though the ANE's precision is float16. The experiments were first attempted on MacOS 12, and after MacOS 13 was released, they were retried using MacOS 13. CoreMLTools 6.0 was used with MacOS 12, and CoreMLTools 6.1 was used with MacOS 13.

PyTorch 1.13 (Paszke et al. 2019), Python 3.9, PyTerrier 0.9.1 were used. We used the Asitop

tool (Liu n.d.) to look at the utilisation of the different processor cores whilst the models were running.

#### 2.3.2 Query and Document Encoding

**DistilBERT Model** We followed the instructions in Apple's article and their published sample code (*Apple Neural Engine (ANE) Transformers* 2022), but when run, we can compile it into the mlmodel format, but when we try and use the model to make predictions, the process freezes and makes no progress. When looking at the power statistics for the processor, when started, the CPU usage peaks before quickly dropping back to near zero. This indicates that the model is deadlocked internally.

The same results were obtained when running on MacOS 12 and 13.

As we could not run the optimised model, we cannot compare its efficiency or effectiveness to a model running on the CPU/ GPU or verify Apple's claims.

**BERT Model** Initially, when we compiled a standard BERT model with MacOS 12, the output given by the model was incorrect. It would always give the same embedding vector no matter what the input was. When looking at power usage, the ANE was being used despite not outputting anything useful. After the operating system update, the compiled BERT model gave the expected output embedding vectors.

We recorded the time taken to process the first 2048 documents from MS MARCO-Passage data set (Bajaj et al. 2018) for various batch sizes of documents (1, 2, 4, 8, 16) and for a range of embedding sizes (32, 64, 128, 256, 512).



(a) Batch Size 1 - All Models

(b) Batch Size 1 - GPU and ANE Models

Figure 2.1: Plot of the latency of encoding passages for given embedding sizes, with a batch size of 1

Figure 2.1b shows for a batch size of 1 that for smaller embeddings, the ANE performs the best, and for larger embeddings, it has comparable performance to the GPU models.

Both the PyTorch GPU model and the ANE model show a plateau at first, indicating some overhead with using the models. This overhead is likely from transferring the model weights to the core/ relevant memory so they can be accessed quickly. After the plateau, the models scale linearly with the number of calculations needed.

Figure 2.1a shows the CPU models perform significantly worse than those using the GPU or ANE. The compiled CPU model performs significantly worse than the PyTorch CPU model. This is likely due to the compilation framework being optimised for running things for the GPU and ANE, and the CPU only option is just a fallback to support older hardware.

Both the compiled GPU model and the PyTorch GPU model use the GPU at full utilisation. The compiled ANE model uses 50% GPU, 50% ANE, and 10% CPU. This shows the compiler has optimised to try and make the best use of all hardware available. We haven't been able to measure the power draw for each model, but the ANE model likely uses less power as the GPU is not at maximum utilisation and the ANE is smaller than the GPU.

The amount of calculations a model has to do correlates with the size of the embeddings and the batch size. Figure 2.2 shows the latency against the "model size" being defined at the embedding size multiplied by the batch size. We see that the ANE has an optimum model size of 256, and after that point, the ANE performance is comparable to the performance of the GPU models. Thus, for larger batch sizes, the performance of the ANE model is comparable to that of the GPU models.



Figure 2.2: Plot of the latency of encoding passages against the "model size" of batch size \* embeddings

#### 2.3.3 Document Scoring

**Matrix Multiplication** We compiled a simple PyTorch model that took two tensors as input and multiplied them together. When we ran the compiled model and examined the power usage, the ANE was not used, and the computation took place on the GPU. Despite being supposedly optimised for this type of task, the compiler did not choose to use the ANE.

**Convolutions** We recorded the time taken to perform a convolution of a random query vector with a random matrix to act as the document index 100 times for each model, repeating the measurements five times. We vary the number of documents in the document matrix to see how performance changes as we have more documents. Each document is an embedding vector of size 768. We plot the lowest time of the five measurements for each model at the document count to try and remove the effects of noise of other processes on the machine and show the best performance of the models.

When the number of documents is small, less than 100, the compiled models all run on the CPU. This is to be expected as there is some overhead when interfacing with a different core on the processor. This is also unlikely to be applicable, as most retrieval situations will deal with significantly (possibly orders of magnitudes) more documents, so this is unlikely to limit retrieval performance.

Figure 2.3 shows that after 1000 documents, the performance of the compiled model using the ANE is worse than all the other models. This is likely due to the operation being trivial compared to large networks for which it is likely optimised and the overheads associated with using the ANE. 2.3b indicates when the number of documents increases above 17000, the performance



Figure 2.3: Plot of the time taken for the compiled models to run the convolution.

is equivalent to the results of the compiled GPU model. Looking at the processor utilisation statistics, after 17000 documents, the ANE model doesn't use the ANE and only uses the GPU. The compiled GPU model performance flattens out after 35000 documents, and the linear increase in time is less than that of smaller numbers of documents. This is likely due to a change in how the memory is copied at the system level, giving efficiencies for more data.

The results for the ANE follow an unusual pattern when there are fewer documents, with every eighth point being an outlier significantly quicker than the trend of points. This indicates that there could be large benefits from operating with tensors that exactly match set sizes for the ANE or that they could just arise from some regular background process.



*Figure 2.4:* Plot of the time taken for convolution to score a single query against a given number of documents for all the models.

Figure 2.4 shows the original PyTorch models perform better than all compiled models. The PyTorch GPU model is substantially faster than any other model, but if the time taken to copy the tensors to GPU memory is included, it performs similarly to the PyTorch CPU model. The

overhead of memory and copying the data so it can be used is a substantial amount of time for these computations. The compiled models manage their memory copying, so it cannot be split out to compare with PyTorch GPU. PyTorch is well-optimised for these calculations, and the results show that the CPU, despite being for general usage, is very capable when performing these types of calculations.

The compiled CPU model performs much worse than the PyTorch CPU model, but this matches the results from the BERT model (2.3.2). Both models should have the same operations available to them, but it is possible that the compiled model only uses a subset of available CPU operations, so the model is more portable between different processors. In an ideal world, the compiled model could detect if it performed worse and then use the original model. The compiled CPU model is also less flexible as the input sizes need to be decided at compile time, this should lead to more optimisation opportunities but those appear to be unused. This also matches the results found by Openja et al. (2022), which showed poor performance of models compiled by CoreMLTools.

When the number of documents increases above several million, compilation becomes prohibitively costly, and it takes several minutes to compile the model. When there are enough documents, the model runs only on the CPU, likely because it needs more data than the GPU can access directly. Full results for this number of documents are not included as it takes significant time to run.

#### 2.3.4 Effectiveness of Compiled Models

We evaluate the effectiveness results of a modified version of the TCT-ColBERT implementation from the PyTerrier DR library (MacAvaney and Macdonald 2022), which scores the documents in the index, using the compiled convolution model instead of matrix multiplication on the CPU. We run the retrieval on the TREC 2019 and 2020 Deep Learning Track judged query subset of MSMARCO-Passage (Craswell et al. 2020; 2021). We evaluate the retrieval on mean average precision (map) and precision for the top five documents (P@5). We consider mean average precision to see the performance for the whole ranking and precision at five to see the performance at the top.

Tables 2.1 and 2.2 show the results for mean average precision are similar, with results agreeing to two decimal places, but some results have been improved, and some have regressed. The precision at five of the compiled models has slightly reduced, indicating the results have become less relevant.

**Table 2.1:** Effectiveness results of a TCT-ColBERT model with compiled convolutions and original matrix multiplication versions on the TREC Deep Learning Track 2019 MSMARCO-Passage Judged subset

	map	map +	map –	map p-value	P@5	P@5 p-value
Original (CPU) Compiled (GPU)	0.41725 0.41748	25	18	0.48388	0.855814 0.851163	0.323037

The results are not statistically significant but indicate that the effectiveness of the compiled versions is broadly similar to the original version. The results are not expected to be identical due to floating point instability and the different models doing different operations in different orders. It also shows that the evaluation metrics are very sensitive to minor changes in the scores. The gap between the scores in the rankings is likely small, thus a small change from floating point instability could cause two documents to swap, this is indicated by more queries changing mean average precision but the documents remaining in the top 5 remain more constant. As the

**Table 2.2:** Effectiveness results of a TCT-ColBERT model with compiled convolutions and original matrix multiplication versions on the TREC Deep Learning Track 2020 MSMARCO-Passage Judged subset

	map	map +	map –	map p-value	P@5	P@5 p-value
Original (CPU)	0.439075				0.844444	
Compiled (GPU)	0.439268	26	16	0.799024	0.840741	0.658939
Compiled (ANE)	0.439338	28	24	0.726879	0.840741	0.658939

ANE uses half-precision floats, the gaps between representable numbers are larger, so results will differ.

#### 2.4 Discussion

We discuss how our results from Section 2.3 apply to neural information retrieval.

#### 2.4.1 Query Encoding

When encoding queries, a batch size of one is the most relevant; as for consumer search, there will likely only be one query at a time, and the query is the only text which needs to be encoded at runtime, as the index is computed ahead of time. The results show that models compiled to use the ANE run faster than models running on the GPU when the embedding size is smaller. For larger embedding sizes, the ANE model performs similarly to models running on the GPU. Using quantisation, binning, and arranging the model more optimally for the ANE, it is likely possible to improve the performance for larger embedding sizes.

We find that neural accelerators can be used to improve the performance of query encoding.

#### 2.4.2 Document Encoding for Indexes

We want to process many documents simultaneously when encoding, so we look at larger batch sizes. Similar to a batch size of one, the ANE performs better for the smaller embeddings, but for the larger embedding sizes, the performance is reduced to similar to the performance of the GPU models. The throughput of the models increases with the batch size.

To be used for dense retrieval, all documents in the corpus need to be encoded with the model. When the compiled ANE model is used, an estimate of around three days is given to index the nine million documents of MS MARCO-Passage. This is a significant improvement over the CPU version of roughly seven days. The time taken for indexing, using any of the cores, is significantly larger than the minutes taken to produce an index for sparse retrieval, thus making dense retrieval less attractive on the devices.

However, most people don't have nine million documents to search through on their computers instantly. Based on the average number of emails an average person receives a day (Ceci 2022), most people are unlikely to have more than one million emails. Most people will have a smaller number of documents which can be processed in a reasonable amount of time. Applications with more documents will likely be in a commercial setting, where a delay may not be such an issue.

The documents could be processed overnight to use the device when not being used, and more documents can be added to the index over time. The documents could also be sent to an external server for the indexing phase, but that removes privacy, which is the main advantage of local processing.

We find that for processing a large number of documents, the neural accelerator does not improve over the performance of the GPU, and indexing a large number of documents can still take a considerable time.

#### 2.4.3 Document Scoring

The results show that the best approach is to perform the similarity computation on the CPU using PyTorch. We found that ANE does not improve the performance of document scoring, and the compilation process does not significantly change the output of calculations.

As the compiler is updated and improved, this could change in the future, but the compiled models are a large distance behind PyTorch. The memory overhead from transferring the documents index to memory for the GPU is significant. Neural accelerators are likely optimised for repeated calculations with the same set of data rather than loading new data from the index each time.

If we wanted to use the compiled convolution model for retrieval, we would have to undertake more work to ensure the system's efficiency. As the convolution layer contains the "smaller matrix" inside, it is static after convolution. This means the query vector must be decided when the model is compiled. The query used won't be static, so the model would need to be compiled each time. As the model is relatively simple, the compilation takes only a couple of seconds if the document count is not too high. However, any additional latency may not be acceptable for some applications. The mlmodel format which the model is compiled into is openly documented (*Core ML Tools* n.d.), so it is possible that the file could be altered directly to set the query rather than needing to compile a new model each time. Other approaches for the convolution, including the PyTorch functional API (*torch.nn.functional.conv1d — PyTorch 1.13 documentation* n.d.), which would allow for the query vector to be changed dynamically, are not supported by the CoreMLTools compiler.

### 2.5 Summary

We found that running on the neural engine improves the performance of encoding singular passages for small embedding sizes. However, when the model has more calculations (for example, when there are more passages or larger embeddings), the performance is similar to that of the GPU. Thus, it could improve the performance when encoding singular queries at runtime but not when encoding many documents for the index.

We found that the compiled models perform worse for scoring documents, and the best approach is to perform the calculations on the CPU. When used in a full retrieval pipeline, the compiled models slightly changed the ranking scores, but likely due to floating point instability.

We also found that for the convolution experiment, the time taken to transfer the data to the GPU is the majority of the processing time when using the GPU. In Chapter 3, we look at whether unified memory can make the copies redundant to improve the performance of scoring documents.

# 3 Unified Memory

In the previous chapter (2.3.3), we found that transferring the tensors to the GPU can take up a large amount of the total processing time. Unified memory allows the GPU and CPU to access the same memory, which should reduce the time taken. In this chapter, we examine if unified memory can improve information retrieval performance.

## 3.1 Background

We introduce the unified memory architecture and some potential benefits, as well as memory mapping, which could be used to reduce copies of the large index matrices.

#### 3.1.1 Unified Memory Architecture

"Traditional" discrete GPUs have a separate memory bank (Peddie 2022). This simplifies the GPU integration as it can be considered an external device, and the CPU does not need any specific support for a GPU. But this means that any data they work on needs to be copied over to its separate memory before any calculations can begin, adding overhead to using the GPU. When the GPU is used, it is usually significantly faster than the CPU. Thus, the time copying does not slow down the processing. Memory is generally on a separate chip from both the CPU and GPU.

In recent years, many consumer processors have integrated GPUs, with the GPU on the same chip as the CPU. This allows lower power usage and longer battery life whilst having the benefit of specialised hardware for graphics to speed up computation. However, as they are built into the same chip as the CPU, they are less powerful than discrete GPUs. Integrated GPUs share the same memory bank as the CPU but have separate regions in memory allocated by the operating system. Depending on the operating system, the split may be infrequently changed at run-time, but the split is decided at boot time for most systems. Neither can access the other's memory, thus for the GPU to perform calculations on data it must first be copied into GPU memory. This will have less overhead than copying to another device but is still an additional step before the data can be worked on. Memory is generally on a separate chip from the processor.

The new System-on-a-Chip (SoC) processors we are looking at have the CPU, GPU and memory all on a single chip. This reduces the communication latency between the different parts of the system and the power usage as it does not need to communicate with another part. Apple has released a chip (*Apple unleashes M1* 2020), which allows all the different processor cores to access the same memory bank at once. Allocations can be interleaved rather than having separate partitions of memory. This will enable data to be shared without copying it to another device or part of memory. This means switching overhead between processing data on the CPU and GPU should be much lower, allowing the best hardware to be used more often, as switching has little overhead. It also allows the GPU to access much more memory for a specific computation than you would otherwise want to dedicate to the GPU, as it can be used for the computation and then freed for use on the CPU again. This means that the memory capacity can not be increased without swapping out the whole chip. Apple was the only manufacturer from which we could find a processor with unified memory; other SoC processors behave like integrated GPUs with memory on the same chip. Intel's eleventh-generation processor documentation (*Intel® Processor Graphics Gen11 Architecture* 2019) mentions unified memory, and Intel holds a patent (Rao and Sundaresan 2016) on unified memory architectures, but we could find no products mentioning using a unified memory architecture. Apple has published very little about its design other than claiming that it is faster and saying that the CPU and GPU can access the same memory simultaneously.

Kenyon and Capano (2022) compare the performance of Apple's new processors against top-ofthe-range NVIDIA GPUs. They found that unified memory's performance increased significantly across various parts of their benchmarks. However, they give no details about how they used the unified memory or the details of the changes they said they made to the algorithms to enable this. They also found that the M1 processors performed better than the top-of-the-range NVIDIA A100, which cost roughly ten times more. This is unexpected but could be due to the transfer time being a large part of the benchmarks.

#### 3.1.2 NVIDIA Unified Memory

NVIDIA has a feature called unified memory (Harris 2013). However, it is very different to Apple's unified memory. NVIDIA's unified memory uses virtual memory space so memory doesn't need explicitly transferred to GPU memory, memory pages are transferred on demand. The programmer doesn't need to think about explicit copy system calls; the underlying system manages the transfer. Both Li et al. (2015) and Landaverde et al. (2014) find that NVIDIA's unified memory has a high overhead and that the performance is highly dependent on memory access patterns, but marginally improves code complexity.

### 3.1.3 Memory Mapping

Memory mapping is a feature offered by operating systems (Bovet and Cesati 2006) which maps a file on disk to a program's memory. The program can act on the file as if it has all been loaded into memory. As the program accesses a pointer to the file, the operating system loads the required data pages from the disk. As the program uses parts of the file that haven't been previously accessed, those parts are loaded into memory. If memory gets full, the operating system removes older memory pages.

Memory mapping can offer several benefits over the standard file system calls. It can allow more efficient memory utilisation of the system, as the operating system manages which pages are in memory at once, so unused parts do not need to be loaded. This can also be a downside, as the program cannot specify which pages need to be kept in memory at all times. Memory mapping gives faster access to files than the standard read and write operations, as it leverages virtual memory capabilities in the operating system rather than having to allocate, copy and de-allocation data buffers in the process. It also allows multiple processes to access the same large file without making multiple copies.

Crotty et al. (2022) raise some problems with using memory mapping. Their main point concerns the difficulties of managing a mutable file's shared state due to the lack of control over when writes are flushed to disk. This is unlikely to be a problem when used for information retrieval, as the index is created ahead of time and can be static and immutable during retrieval. They also claim that the performance is worse, as any operation could cause a page fault and a wait for file I/O to occur. However, this highly depends on the setup, access patterns, and other caching being manually performed. Fedorova (2022) show memory mapping performs better than standard read system calls.

Memory mapping is used with inverted indexes from sparse retrieval (Hawking 2003) to reduce memory usage and improve the performance of access to the index. However, memory mapping

has not been used with neural retrieval, as the index is generally used on the GPUs, and operating systems do not support memory mapping to GPUs as they all have different interfaces.

## 3.2 Method

We attempt to use memory mapping to load the index directly into memory, which the GPU can access. Since the GPU and CPU share the same memory and can access it simultaneously, this should remove the need for a copy "on the CPU" before copying it over to the GPU. It also enables the operating system to manage the memory; thus, if there is memory pressure, it can load only relevant parts of the current index.

As operating systems do not support memory mapping to the GPU, no libraries support it, even with unified memory. Therefore, we use Numpy (Harris et al. 2020) to memory map the index from a file into memory, then load the Numpy array as a PyTorch tensor. PyTorch can load Numpy arrays directly (*torch.from\_numpy — PyTorch 1.13 documentation* n.d.); thus, no copy will be needed. The PyTorch tensor can then be set for the GPU, and the index can be used for retrieval. As the unified memory should be able to be accessed by both the CPU and GPU, the device setting should be a constant time operation.

## 3.3 Implementation

We look at two of the most popular deep-learning frameworks, PyTorch (Paszke et al. 2019) and Tensorflow (Abadi et al. 2016), to see if they can use the unified memory architecture.

#### 3.3.1 PyTorch

When we investigated setting the device a tensor is on, the pointer of the tensor storage buffer changed. The storage location of the tensor changes when it is set to the GPU and when it is set to the CPU. When the device is changed, the whole tensor is copied. This would remove any benefits of memory mapping, as we need to make a copy of the entire tensor to get the tensor to the GPU.

Apple's low-level framework for interacting with the GPU is called Metal Performance Shaders (MPS) (*Metal Performance Shaders* n.d.) and has various options (*MTLStorageMode* n.d.) for setting which cores can access buffers of allocated memory. There is likely an overhead of allowing multiple parts of the processor to access memory, so it could make sense to set the default CPU buffers to only be accessible to the CPU. However, PyTorch has no option to have a tensor accessible by both the CPU and GPU.

Some operations are not available in the MPS backend for PyTorch (albanD 2022), thus falling back to running on the CPU. When this happens, the tensors are copied to the CPU, the operation is applied, and then they are copied back to "GPU memory". If an operation has to fall back, it would make sense to copy into shared memory so the additional copies don't need to take place, and if the operation has been used once, it is likely to be used again.

We tried to compile a PyTorch version with the modified MPS backend so that tensors did not need to be copied to change if it was running on the CPU or the GPU. We could compile PyTorch on the Mac Studio with full MPS GPU support. However, we were unable to modify it to prevent unwanted copying. PyTorch's support for different backends is tightly coupled with the concept of storage buffers. This makes sense for devices with separate memory for the CPU and GPU, but it does not make sense when you have unified memory with multiple different devices that can access the same memory. The CPU backend has a different structure from the MPS backend, and we could find no method to convert between them apart from copying the whole tensor. The GPU operations must be in MPS storage, and similarly, for CPU operations, they must be in CPU storage. We were unable to modify PyTorch so that the CPU backend and MPS backends could each view the same buffer in memory.

It is likely possible to modify PyTorch to prevent these extra copies, but we do not have the required knowledge of the internal structures and how the foreign function interface with the MPS Objective C bindings works to be able to make the changes required.

#### 3.3.2 Tensorflow

Tensorflow is another popular machine learning framework that supports MPS through a plugin (*Tensorflow Plugin - Metal* n.d.). As PyTorch does not fully support unified memory, we investigated if Tensorflow allowed you to modify a tensor with the CPU and GPU without copying it.

However, Tensorflow gives less fine-grained control of which device tensors are on, only being able to specify on which device computation will take place. It appears to copy the tensor when the device is changed, but it is hard to verify as Tensorflow does not allow you to inspect the data buffer pointer, and the source for the MPS plugin is unavailable.

## 3.4 Theoretical Results

As the existing machine learning frameworks do not support unified memory, we cannot collect results for the benefits of unified memory. We instead look at the results of the convolution experiment previously performed (Section 2.3) to see the potential speed-up available if we don't have to copy the data. We compare the time taken to perform the convolution on a tensor set to the GPU and with a tensor on the CPU and then set to the GPU before performing the convolution.

Figure 3.1a shows that the overhead for up to 5000 documents is roughly constant, indicating a fixed overhead with creating the new tensor and copying over the data. Performing the convolution 100 times takes around 5.7ms for 1000 documents without the memory copy. With a memory copy, each time for 100 convolutions takes around 56.9ms, over ten times slower.

Figure 3.1b shows that for large numbers of documents, the time, including memory copying, increases roughly linearly, but it remains constant without the memory copy. For 100000 documents without a memory copy, it still takes around 5.7ms, whilst, with the memory copy, it takes 1122.7ms, almost 200 times slower.

## 3.5 Discussion

Due to a lack of support from current software, we cannot see if unified memory can speed up retrieval by reducing the copying of the indexes. We find that the time taken to copy the tensors so the GPU can use them is significant; the time taken to copy the memory is at least 90% of the total time. This is likely a worst-case scenario, as data is copied for a singular operation. As it is repeated 100 times, the pressure on the system memory allocator and Python's garbage collector will be high, as for a large number of documents, it will have to keep allocating large amounts of memory quickly. These results are likely an upper bound, as additional overheads with sharing memory simultaneously between the GPU and CPU are likely.

Software often limits what new hardware features in processors can achieve. Most software is run on different hardware platforms, so unique hardware features will not often be used, as only the lowest common denominator of features can easily be used. These limitations are often worked around in compilers to provide fallbacks so that the best possible code is generated.



*Figure 3.1:* Plot of the time taken for PyTorch model running on the GPU to perform the calculation, including and excluding the time taken to copy memory.

Apple's CoreMLTools compiler aims to compile models to run on all their hardware, but it gives little information about the output or how it utilises unified memory architecture to improve performance.

Tools such as PyTorch are hesitant to support new features such as NVIDIA unified memory (albanD 2019) without seeing a good enough performance improvement to justify the development required. Apple hardware is not currently used widely for production deployments of machine learning models, thus tool makers are more likely to focus on other platforms. As there are currently no results showing the performance benefits, and the change could require substantial architectural changes, tools are unlikely to support unified memory until results are available.

### 3.6 Summary

We found that unified memory is not supported by current software, and additional copies of data are needed. When we look at the potential time saved by not having to copy the data, we find that around 90% of the total time can be spent just copying data, which could be avoided using unified memory.

In Chapter 4, we summarise the findings of the whole project and suggest future research areas.

# 4 Conclusion

We have examined whether neural accelerators and unified memory can improve the performance of neural information retrieval tasks on consumer devices. We summarise our findings, suggest future directions for research, and reflect on what we learned from the project.

## 4.1 Summary

The state of neural accelerators is progressing very quickly, with software updates still regularly changing what is possible. Neural accelerators are still very new for desktop-type systems, with the first hardware only being available for around two years and only coming from a single manufacturer. The number of tasks the neural accelerator could be used for expanded throughout the project. Initially, the neural accelerator could not be used to encode documents or queries.

After the operating system update, we found that for singular passage encoding, or at small embedding sizes, the neural accelerator processes it faster than when the language models are run on the GPU or the CPU. This means that using the neural accelerator, we can speed up the encoding of queries at runtime, as we will likely have a single query on a user's device. However, it is highly dependent on the parameters of the model. When used with larger inputs, the neural accelerator performance regresses to a standard similar to the GPU performance, which is still significantly better than just using the CPU. Thus, we found that for encoding a large number of documents at once for indexing, the neural accelerator does not improve the throughput of document encoding.

We also found that we could not use the neural accelerator to speed up the scoring of documents during retrieval. When run on the neural accelerator, the convolution to score the query vector against the large index matrix is slower than running on the CPU, despite the neural accelerator being designed for this type of calculation. We find that the compilation process to run on the neural accelerator does not statistically significantly change the effectiveness of the retrieval results. The plain PyTorch model on the GPU is faster than other models, but if the time taken to copy memory is included, the performance is similar to running just on the CPU.

The time taken to copy data for use on the GPU is considerable. We found that the time taken to copy the data for use on the GPU can be over 90% of the total time taken. Unified memory should mean that these copies are redundant, and the CPU and GPU can access the memory without a copy. However, none of the existing tools support this, and we could not adapt them to use this for a potentially large performance increase. Thus, we are unable to determine if unified memory can improve the performance of retrieval systems, but the time taken to copy memory has a large potential for these changes.

Overall, the neural accelerator can speed up some calculations but is highly dependent on the exact workload running on the accelerator. There is little documentation of what workloads will benefit from running on the neural engine, so each workload must be executed to see if it will improve performance. Unified memory has big potential, but there is no software support currently.

Based on these studies, the new architecture slightly improves neural information retrieval performance, but it is probably not enough to enable efficient search on users' devices. If unified memory gains full software support, the performance improvements could be large, allowing neural search to be efficiently used on user devices.

## 4.2 Future work

We did not measure the power consumption of any of the methods. This is important for consumer devices like phones as they are more likely to be run on battery; thus, energy efficiency is crucial. Future work could be done to measure the power consumption of the models and see if the neural accelerator models have better energy performance than the GPU. Neural accelerators have specialised hardware for their computations; thus, they should have lower power usage. As the time reductions are not substantial, the low power could be the key benefit of the neural accelerators.

We could also examine the effectiveness of the compiled language models to see if the compilation process affects the overall retrieval performance. Alternative scoring methods could also be used to investigate the effectiveness listwise of all the compiled models.

In Section 3.3, we found that machine learning frameworks don't currently support the unified memory architecture, leading to excessive data copies. We found that memory transfer makes up a large amount of the processing time; therefore, work to adapt the leading frameworks to support a unified memory architecture could significantly impact processing time. Alternatively, the low-level bindings from Apple could be used to show the potential performance impact of using a unified memory architecture and justify work being done to update the high-level frameworks.

## 4.3 Reflections

By writing up this project, I better understand what to look for and what to focus on when doing research. At the start of the year, I felt lost, not knowing what was worth focusing on. By writing up what I have done, I am more confident in identifying what would be useful to examine and what to ignore.

In the future, I would keep much better records of what versions of libraries were used when experiments were run and all results that could be recorded, including rough processor utilisation and rough duration, so they can be referred back to if anything is needed rather than having to wait for them to be rerun. I would also not make any assumptions about results after updating software. After the operating system was updated, I tested the first language model, and it gave the same results, so I skipped the second, but the second would work with the update. Luckily, when I tested it again later, I found this out.

I would also regularly look for relevant literature after the initial literature review. Several of the most relevant papers on the topic were only published after my initial literature review and would have been useful earlier on rather than when I was writing up the project.

## Bibliography

- Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y. and Zheng, X. (2016), TensorFlow: a system for large-scale machine learning, *in* 'Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation', OSDI'16, USENIX Association, USA, pp. 265–283.
- albanD (2019), 'Pytorch with CUDA Unified Memory'. URL: https://discuss.pytorch.org/t/pytorch-with-cuda-unified-memory/60783/2
- albanD (2022), 'General MPS op coverage tracking issue · Issue #77764 · pytorch/pytorch'. URL: https://github.com/pytorch/pytorch/issues/77764
- Apple Neural Engine (ANE) Transformers (2022). URL: https://github.com/apple/ml-ane-transformers
- Apple unleashes M1 (2020). URL: https://www.apple.com/uk/newsroom/2020/11/apple-unleashes-m1/
- Bajaj, P., Campos, D., Craswell, N., Deng, L., Gao, J., Liu, X., Majumder, R., McNamara, A., Mitra, B., Nguyen, T., Rosenberg, M., Song, X., Stoica, A., Tiwary, S. and Wang, T. (2018), 'MS MARCO: A Human Generated MAchine Reading COmprehension Dataset'. arXiv:1611.09268 [cs].
  URL: http://arxiv.org/abs/1611.09268
- Bovet, D. P. and Cesati, M. (2006), Understanding the Linux kernel, 3rd ed edn, O'Reilly, Beijing ; Sebastopol, CA. URL: https://www.oreilly.com/library/view/understanding-the-linux/0596005652/ch16s02.html
- Ceci, L. (2022), 'Emails sent per day 2025'. URL: https://www.statista.com/statistics/456500/daily-number-of-e-mails-worldwide/
- *Core ML Model Format Specification Core ML Format Reference documentation* (n.d.). **URL:** *https://apple.github.io/coremltools/mlmodel/index.html*
- Core ML Tools (n.d.). URL: https://coremltools.readme.io/docs
- Craswell, N., Mitra, B., Yilmaz, E. and Campos, D. (2021), 'Overview of the TREC 2020 deep learning track'. arXiv:2102.07662 [cs]. URL: http://arxiv.org/abs/2102.07662
- Craswell, N., Mitra, B., Yilmaz, E., Campos, D. and Voorhees, E. M. (2020), 'Overview of the TREC 2019 deep learning track'. arXiv:2003.07820 [cs]. URL: http://arxiv.org/abs/2003.07820

Crotty, A., Leis, V. and Pavlo, A. (2022), Are You Sure You Want to Use MMAP in Your Database Management System?, *in* '{CIDR} 2022, Conference on Innovative Data Systems Research'.

**URL:** *https://db.cs.cmu.edu/mmap-cidr2022/* 

- Deploying Transformers on the Apple Neural Engine (2022). URL: https://machinelearning.apple.com/research/neural-engine-transformers
- Devlin, J., Chang, M.-W., Lee, K. and Toutanova, K. (2019), 'BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding'. arXiv:1810.04805 [cs]. URL: http://arxiv.org/abs/1810.04805
- Fedorova, A. S. (2022), 'Why mmap is faster than system calls'. URL: https://sasha-f.medium.com/why-mmap-is-faster-than-system-calls-24718e75ab37
- Fee, L. (2021), 'Low level API to take control of Neural Engine'. URL: https://developer.apple.com/forums/thread/673627?answerId=686278022
- Gil-Costa, V., Loor, F., Molina, R., Nardini, F., Perego, R. and Trani, S. (2022), Ensemble Model Compression for Fast and Energy-Efficient Ranking on FPGAs, *in* M. Hagen, S. Verberne, C. Macdonald, C. Seifert, K. Balog, K. Nørvåg and V. Setty, eds, 'Advances in Information Retrieval', Vol. 13185, Springer International Publishing, Cham, pp. 260–273. Series Title: Lecture Notes in Computer Science.
- Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M. H., Brett, M., Haldane, A., del Río, J. F., Wiebe, M., Peterson, P., Gérard-Marchant, P., Sheppard, K., Reddy, T., Weckesser, W., Abbasi, H., Gohlke, C. and Oliphant, T. E. (2020), 'Array Programming with NumPy', *Nature* 585(7825), 357–362. arXiv:2006.10256 [cs, stat]. URL: http://arxiv.org/abs/2006.10256
- Harris, M. (2013), 'Unified Memory in CUDA 6'. URL: https://developer.nvidia.com/blog/unified-memory-in-cuda-6/
- Hawking, D. (2003), Very Large Scale Information Retrieval, *in* S. Renals and G. Grefenstette, eds, 'Text- and Speech-Triggered Information Access: 8th ELSNET Summer School, Chios Island, Greece, July 15-30 2000. Revised Lectures', Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, p. 130.
  URL: https://doi.org/10.1007/978-3-540-45115-0\_5
- Hotz, G. (n.d.), 'tinygrad/accel/ane at master · geohot/tinygrad'. URL: https://github.com/geohot/tinygrad
- Ignatov, A., Timofte, R., Chou, W., Wang, K., Wu, M., Hartley, T. and Van Gool, L. (2018), 'AI Benchmark: Running Deep Neural Networks on Android Smartphones'. arXiv:1810.01109 [cs].
  - **URL:** *http://arxiv.org/abs/1810.01109*
- Intel® Processor Graphics Gen11 Architecture (2019). URL: https://www.intel.com/content/dam/develop/external/us/en/documents/the-architecture-ofintel-processor-graphics-gen11-r1new.pdf
- Kasperek, D., Podpora, M. and Kawala-Sterniuk, A. (2022), 'Comparison of the Usability of Apple M1 Processors for Various Machine Learning Tasks', Sensors 22(20), 8005. URL: https://www.mdpi.com/1424-8220/22/20/8005

- Kenyon, C. and Capano, C. (2022), 'Apple Silicon Performance in Scientific Computing'. arXiv:2211.00720 [physics]. URL: http://arxiv.org/abs/2211.00720
- Khattab, O. and Zaharia, M. (2020), 'ColBERT: Efficient and Effective Passage Search via Contextualized Late Interaction over BERT'. arXiv:2004.12832 [cs]. URL: http://arxiv.org/abs/2004.12832
- Landaverde, R., Zhang, T., Coskun, A. K. and Herbordt, M. (2014), An investigation of Unified Memory Access performance in CUDA, *in* '2014 IEEE High Performance Extreme Computing Conference (HPEC)', pp. 1–6.
- Li, W., Jin, G., Cui, X. and See, S. (2015), An Evaluation of Unified Memory Technology on NVIDIA GPUs, *in* '2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing', pp. 1092–1098.
- Lin, S.-C., Yang, J.-H. and Lin, J. (2020), 'Distilling Dense Representations for Ranking using Tightly-Coupled Teachers'. arXiv:2010.11386 [cs]. URL: http://arxiv.org/abs/2010.11386
- Liu, J., Liu, J., Du, W. and Li, D. (2019), 'Performance Analysis and Characterization of Training Deep Learning Models on Mobile Devices'. arXiv:1906.04278 [cs, stat]. URL: http://arxiv.org/abs/1906.04278
- Liu, T. (n.d.), 'asitop | Perf monitoring CLI tool for Apple Silicon'. URL: https://tlkh.github.io/asitop/
- MacAvaney, S. and Macdonald, C. (2022), 'PyTerrier DR'. URL: https://github.com/terrierteam/pyterrier\_dr
- Macdonald, C. and Tonellotto, N. (2020), Declarative Experimentation in Information Retrieval using PyTerrier, *in* 'Proceedings of the 2020 ACM SIGIR on International Conference on Theory of Information Retrieval', pp. 161–168. arXiv:2007.14271 [cs]. URL: http://arxiv.org/abs/2007.14271
- *Metal Performance Shaders* (n.d.). **URL:** *https://docs.developer.apple.com/documentation/metalperformanceshaders*
- Molina, R., Loor, F., Gil-Costa, V., Nardini, F. M., Perego, R. and Trani, S. (2021), 'Efficient traversal of decision tree ensembles with FPGAs', *Journal of Parallel and Distributed Computing* 155, 38–49.
  URL: https://linkinghub.elsevier.com/retrieve/pii/S0743731521000915
- *MTLStorageMode* (n.d.). **URL:** *https://docs.developer.apple.com/documentation/metal/mtlstoragemode*
- ONNX | Home (n.d.). URL: https://onnx.ai/
- Openja, M., Nikanjam, A., Yahmed, A. H., Khomh, F. and Jiang, Z. M. J. (2022), An Empirical Study of Challenges in Converting Deep Learning Models, *in* '2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)', IEEE, Limassol, Cyprus, pp. 13–23.

**URL:** *https://ieeexplore.ieee.org/document/9978197/* 

- Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J. and Chintala, S. (2019), 'PyTorch: An Imperative Style, High-Performance Deep Learning Library'. arXiv:1912.01703 [cs, stat]. URL: http://arxiv.org/abs/1912.01703
- Peddie, J. (2022), The GPU Environment—Hardware, *in* J. Peddie, ed., 'The History of the GPU Eras and Environment', Springer International Publishing, Cham, pp. 151–200.
  URL: https://doi.org/10.1007/978-3-031-13581-1\_5
- Rao, J. N. and Sundaresan, M. (2016), 'Memory sharing via a unified memory architecture'. URL: https://patents.google.com/patent/US9373182B2/en
- Sanh, V., Debut, L., Chaumond, J. and Wolf, T. (2020), 'DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter'. arXiv:1910.01108 [cs]. URL: http://arxiv.org/abs/1910.01108
- Tensorflow Plugin Metal (n.d.). URL: https://developer.apple.com/metal/tensorflow-plugin/
- *The future is here: iPhone X (2017).* **URL:** *https://www.apple.com/uk/newsroom/2017/09/the-future-is-here-iphone-x/*
- torch.from\_numpy PyTorch 1.13 documentation (n.d.). URL: https://pytorch.org/docs/stable/generated/torch.from\_numpy.html
- torch.nn.functional.conv1d PyTorch 1.13 documentation (n.d.). URL: https://pytorch.org/docs/stable/generated/torch.nn.functional.conv1d.html
- Wang, S., Fanelli, A. and Kilpatrick, L. (n.d.), 'ChatGPT, GPT4 hype, and Building LLM-native products — with Logan Kilpatrick of OpenAI'. URL: https://lspace.swyx.io/p/chatgpt-gpt4-hype-and-building-llm
- Wu, W. (2021), 'Apple Neural Engine Internal: From ML Algorithm to HW Registers'. URL: https://www.blackhat.com/asia-21/briefings/schedule/#apple-neural-engine-internal-fromml-algorithm-to-hw-registers-22039
- Xiong, L., Xiong, C., Li, Y., Tang, K.-F., Liu, J., Bennett, P., Ahmed, J. and Overwijk, A. (2020), 'Approximate Nearest Neighbor Negative Contrastive Learning for Dense Text Retrieval'. arXiv:2007.00808 [cs]. URL: http://arxiv.org/abs/2007.00808