

1

Introduction to Vectorization

1.1 What Is Vectorization

Vectorization is a type of parallel computing paradigm that performs arithmetic operations on an array of numbers [Intel, 2022] within a single data processing unit (either CPU or GPU). Modern CPUs can perform between 4 and 16 single precision (float32) parallel computations, depending on the type of instructions [Intel, 2022] (see Figure 1.1).

Here, SSE, or streaming SIMD (single-instruction-multiple-data) extension, has 128 registers; each register is used to store 1-bit of data for computations. This is equivalent to performing four single-precision (float32) operations or two double-precision (float64) operations. Intel's AVX512, or Advanced Vector Extension 512, on the other hand, has 512 registers and can therefore process 512 bits of data simultaneously. This is equivalent to performing 16 single-precision or 8 double-precision operations.

The term “**Vectorization**” also refers to the process of transforming an algorithm computable on one data point at a time to an operation that calculates a collection of data simultaneously [Khartchenko, 2018]. A classic example is the dot product operation,

$$\sum_i^n a_i b_i = \mathbf{a} \cdot \mathbf{b}$$

where $\mathbf{a} := [a_1, a_2, a_3, \dots, a_n]$, and $\mathbf{b} := [b_1, b_2, b_3, \dots, b_n]$.

The dot product operation on the right-hand side of the equation is the vectorized form of the summation process on the left. Instead of operating on one element of the array at a time, it operates on the collections or arrays, $\mathbf{a}$ and $\mathbf{b}$.

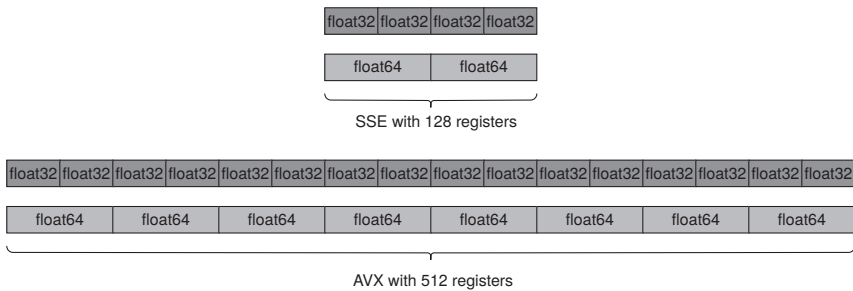


Figure 1.1 CPU vector computations.

i Note

Another definition of the term “vectorization” is related to the more recent advances in large language models: using a vector or an array of values to represent a word. This is more commonly known as **embedding** in the literature which originated from the word2vec model in natural language processing. In Chapters 6 and 7, we will introduce the idea of representing string or categorical features as numerical values, i.e. integer indices, sparse one-hot encoding vectors, and dense embedding vectors.

1.1.1 A Simple Example of Vectorization in Action

Suppose we want to take the dot product of 1 million random numbers drawn from the uniform distribution. Using Python’s `for` loop, we would typically implement this algorithm as the follows:

```
1 import random
2
3 def sqrt_of_n_random_numbers(n=1_000_000):
4     result = 0
5     for _ in range(n):
6         a = random.uniform(0, 1)
7         b = random.uniform(0, 1)
8         result += a * b
9     return result
```

On my M1 MacBook Pro under Python 3.11.4, this is the runtime of the above function:

```
294 ms ± 1.39 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

If we implement the algorithm using numpy with vectorization

```
1 import numpy as np
2
3 def sqrt_of_n_random_numbers_np(n=1_000_000):
4     a = np.random.rand(n)
5     b = np.random.rand(n)
6     result = np.dot(a, b)
7     return result
```

Under the same setup, we have the following runtime of the vectorized implementation.

```
7.9 ms ± 119 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

This is a $\sim 37\times$ speed-up by vectorization!

1.1.2 Python Can Still Be Faster!

However, the above example is not to discount that certain native Python operations are still faster than NumPy's implementations, despite NumPy being often used for vectorization operations to simplify for-loops in Python.

For example, we would like to split a list of strings delimited by periods “.”. Using Python's list comprehension, we have

```
1 X = ["foo.bar", "bar.foo", "x.y", "z.k"] * 1000
2 [x.split(".") for x in X]
```

Line 2 runs for

```
299 µs ± 5.98 µs per loop (mean ± std. dev. of 7 runs, 1,000
↳ loops each)
```

However, in comparison, if we use NumPy’s “vectorized” split operation

```
1 Y = ["foo.bar", "bar.foo", "x.y", "z.k"] * 1000
2 Y = np.array(Y)
3 np.core.defchararray.split(Y, ".")
```

Line 3 runs for

```
1.18 ms ± 4.37 µs per loop (mean ± std. dev. of 7 runs, 1,000
↳ loops each)
```

which is about $\sim 4.0\times$ slower than native Python.

1.1.3 Memory Allocation of Vectorized Operations

When using vectorized operations, we need to consider the trade-off between memory and speed. Sometimes, vectorized operations would need to allocate more memory for results from intermediate steps. Therefore, this has implications on the overall performance of the program, in terms of space vs. time complexity. Consider a function in which we would like to compute the sum of the squares of the elements in an array. This can be implemented as a for-loop, where we iterate through each element, take the square, and accumulate the square values as the total. Alternatively, the function can also be implemented as a vectorized operation where we take the square of each element first, store the results in an array, and then sum the squared vector.

```
1 import numpy as np
2
3 def sum_of_square_vectorized(X):
4     x = np.asarray(X)
5     x_sq = np.square(x) # x ** 2
6     res = x_sq.sum()
7     return res
8
9
10 def sum_of_square_for_loop(X):
11     res = 0
12     for x in X:
13         res += x ** 2
14     return res
```

Let us use the `memory_profiler` package to examine the memory usage of each line of the two functions. We can first install the package by `pip`

install `memory_profiler==0.61.0` and load it as a plugin in a Jupyter Notebook.

```
1 !pip install memory_profiler==0.61.0
2 %load_ext memory_profiler
```

We also save the above script into a file (which we call `memory_allocation_example.py`), then we measure the memory trace of each function in the notebook as follows:

```
1 from memory_allocation_example import sum_of_square_vectorized
2 X = np.linspace(-50, 50, 100_000).tolist()
3 %mprun -f sum_of_square_vectorized sum_of_square_vectorized(X)
```

This gives us the following line-by-line memory trace of the function call:

Line #	Mem usage	Increment	Occurrences	Line Contents
4				def sum_of_square_vectorized(X):
5	104.3 MiB	104.3 MiB	1	x = np.asarray(X)
6	105.1 MiB	0.8 MiB	1	x_sq = np.square(x) # x ** 2
7	105.4 MiB	0.3 MiB	1	res = x_sq.sum()
8	105.4 MiB	0.0 MiB	1	return resx

We can also use `%timeit` magic function to measure the speed of the function:

```
1 %timeit sum_of_square_vectorized(X)
```

which gives

```
3.91 ms ± 64.7 µs per loop (mean ± std. dev. of 7 runs, 100
↳ loops each)
```

Similarly, let us also look at the for-loop implementation

```
1 from memory_allocation_example import sum_of_square_for_loop
2 X = np.linspace(-50, 50, 100_000).tolist()
3 %mprun -f sum_of_square_for_loop sum_of_square_for_loop(X)
```

Line #	Mem usage	Increment	Occurrences	Line Contents
11	68.2 MiB	68.2 MiB	1	def sum_of_square_for_loop(X):
12	68.2 MiB	0.0 MiB	1	res = 0
13	68.2 MiB	0.0 MiB	100001	for x in X:
14	68.2 MiB	0.0 MiB	100000	res += x ** 2
15	68.2 MiB	0.0 MiB	1	return res

And we also test the run time as

```
1 %timeit sum_of_square_for_loop(X)
```

which gives

```
5.72 ms ± 39 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

We can see that although the for-loop function is slower than the vectorized function, the for-loop had almost no memory increase. In contrast, the vectorized solution has a ~1M increase in memory usage while reducing the run time by about 30%. If X is several magnitudes larger than our current example, then we can see that even more memory will be consumed in exchange for speed. Hence, we need to consider carefully the trade-off between speed and memory (or time vs. space) when using vectorized operations.

1.2 Case Study: Dense Layer of a Neural Network

Vectorization is a key skill for implementing various machine learning models, especially for deep learning algorithms. A simple neural network consists of several feed-forward layers, as illustrated in Figure 1.2.

A feed-forward layer (also called dense layer, linear layer, perceptron layer, or hidden layer, as illustrated in Figure 1.3) has the following expression:

$$z = \sigma(xw + b)$$

Notice that $y = xw + b$ is a linear model. Here,

- x is the input, and typically of shape (batch_size, num_inputs)
- w is the weight (a.k.a. kernel) parameter matrix of shape (num_inputs, hidden_size)
- b is the bias parameter vector of shape (1, hidden_size)

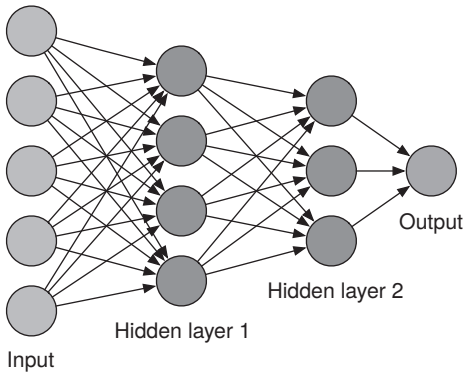


Figure 1.2 Multilayer perceptron.

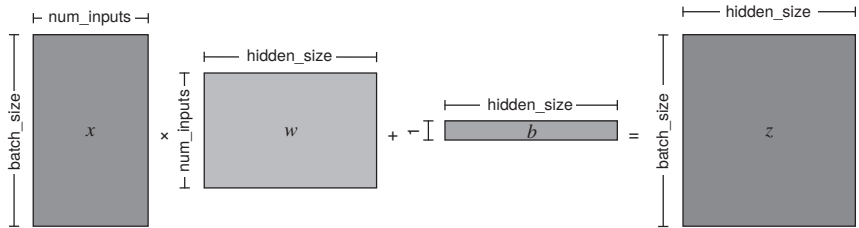


Figure 1.3 Feed-forward layer.

- σ is an activation function that is applied to every element of the resulting linear model operation. It can be, e.g., `relu`, `sigmoid`, or `tanh`.
- z is the output of shape `(batch_size, hidden_size)`

Stacking multiple feed-forward layers creates the **multilayer perceptron model**, as illustrated by our example neural network above.

If we were to implement a feed-forward layer using only Python naively, we would need to run thousands to millions of iterations of the `for-loop`, depending on the `batch_size`, `num_inputs`, and `hidden_size`. When it comes to large language models like generative pretrained transformer (GPT) which has hundreds of billions of parameters to optimize, using `for loops` to make such computations can become so slow that training or making an inference using the model is infeasible. But if we take advantage of vectorization on either CPU or GPU, we would see significant gains in the performance of our model. Furthermore, certain operations like matrix multiplication are simply inefficient to perform if implemented naively by simply parallelizing a `for-loop`, so in

practice must rely on special algorithms such as Strassen's Algorithm [Strassen, 1969] (or more recently, a new algorithm discovered by AlphaTensor [Fawzi et al., 2022]). Implementing these algorithms as an instruction on either CPU or GPU is a non-trivial task itself. Practitioners of machine learning and data science should therefore take advantage of pre-existing libraries that implement these algorithms. But to use these libraries efficiently, one needs to be adept in thinking in terms of vectorization.

In the following, let's take a look at an example of how to implement the above perceptron layer using `numpy`.

```

1  import numpy as np
2
3  def sigmoid(x):
4      """Implement sigmoid activation function."""
5      return 1. / (1 + np.exp(-x))
6
7
8  def dense_layer(x, w, b):
9      # linear model: (batch_size, hidden_size)
10     y = x @ w + b
11     # activation
12     y = sigmoid(y)
13     return y

```

- Lines 3–5 define the sigmoid activation function, $f(x) = \frac{1}{1+e^{-x}}$
- Line 10 implements the linear model
 - Here, the `@` operator is the matrix multiplication operator that is recognized by typical vector libraries like `numpy`, `tensorflow`, and `torch`. We will discuss more on matrix operations in Chapter 4.
 - The result of multiplying `x` and `w` should be a matrix of shape `(batch_size, hidden_size)`, but in this implementation, we have `b` of shape `(1, hidden_size)`. How can a vector be added to a matrix? In Chapter 2, we will explore the concept called **broadcasting**.
- Line 12 applies the sigmoid activation we defined previously to compute the final output of this dense layer. This is an element-wise operation, where we apply the sigmoid function to each element of the matrix. We will explore the idea of element-wise operations in Chapter 2.

As you can see, vectorization can greatly simplify the representation of neural networks operating on high-dimensional arrays with a large number of elements or data points.

1.3 Vectorization vs. Other Parallel Computing Paradigms

Knowing that vectorization is a type of parallel computing paradigm, it is natural to ask the question: how does vectorization compare to other parallel computing paradigms such as multithreading, multiprocessing, or multiworker distributed computing?

1.3.1 Multithreading

Like vectorization, multithreading operates on a single processing unit, with shared memory. This can be a single core of a CPU or GPU. Vectorization processes multiple data inputs within the register in parallel, given a single instruction. In contrast, multithreading can only process one stream of data at a time, instead of processing data in parallel. It can switch between different threads of the same program (or sometimes different threads of different programs), with each thread representing a different part of the process. This is referred to as “concurrency.” Of course, multithreading can be used in combination with vectorization. In fact, modern GPUs operate by combining vectorization, multithreading, and multiprocessing to increase their processing powers [NVIDIA, 2012].

The multithreading paradigm is quite advantageous in tasks like reading data from remote servers. For example, a database has 10 parts. If we have only a single thread, then for each part of the data to be read, we need to wait for one minute until the data is downloaded before reading the next part. Then, it will take 10 minutes in total to download the entire database. However, if we have 10 threads, each making a request to read one part of the data. Since the time to make a request to read the data is negligible compared to the time to download the data, each thread takes one minute to download the data. However, since the downloading process is in parallel with each other, it will take only one minute in total to download the entire database. See Figure 1.4 for an illustration.

1.3.2 Multiprocessing

Multiprocessing, on the other hand, operates on multiple processing units, e.g. multi-core CPUs or GPUs. Each core of the processing unit can access data through a shared memory (the RAM). Unlike vectorization and multithreading, multiprocessing does not share the same data between the processes. Therefore, to process the same data in parallel using different instructions, multiprocessing needs to duplicate the data across the cores. Unlike multithreading, multiprocessing will not be efficient if the process is bounded by I/O (i.e. data reading and writing). Conversely, multiprocessing is a great choice if the process is bounded

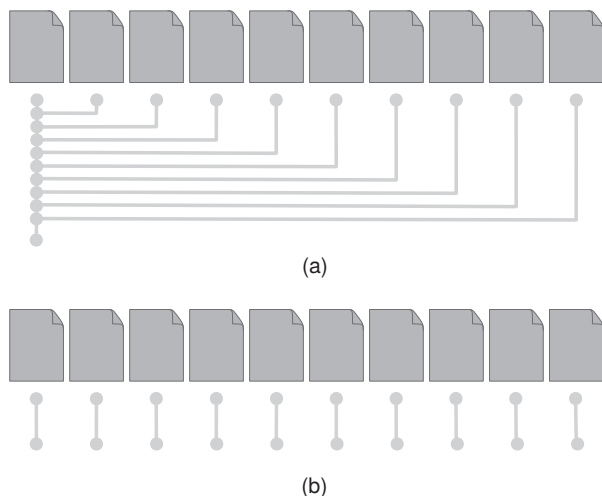


Figure 1.4 Reading remote database (a) with single thread and (b) with multithreading.

by the processing power (a.k.a. CPU bound) and can greatly speed up the process by distributing the computational load to multiple cores of the processing unit.

A typical use of multiprocessing is to perform in-memory batch data processing. Suppose we want to compute the min-max normalized score for a bunch of users:

```
1 import pandas as pd
2
3 df = pd.DataFrame({
4     "id": [
5         "A", "A", "A", "A", "B", "B", "B", "B", "B",
6         "C", "C", "C", "C", "C", "C", "C", "C", "C"
7     ],
8     "scores": [
9         98, 97, 100, 80, 78, 80, 82, 81, 61, 51, 72,
10        70, 82, 80, 75, 62, 67, 53
11    ],
12 })
```

We can define the following user-defined function (UDF) to be used to compute each user's normalized score:

```
1 def apply_func(pdf):
2     """Compute the min-max normalized score"""
3     min_val = pdf["scores"].min()
4     max_val = pdf["scores"].max() + 1e-6
5     pdf["scores"] = (pdf["scores"] - min_val) / (max_val - min_val)
6     return pdf
```

Each grouped batch based on the user's id will be transformed using the `apply_func`:

```
1 df_out = df.groupby(
2     by="id", as_index=False
3 ).apply(apply_func)
```

This takes about 1.5 ms on my MacBook Pro M1 machine with 8 cores, yielding the following results in Table 1.1.

We can perform the same transformation using multiprocessing, where each process will operate on each of the grouped batches of data. We first define the helper function `apply_parallel` which will use Python's `joblib` to create the multiprocessing environment.

Table 1.1 Min–Max normalized scores

Id	Scores
A	0.9
A	0.85
A	1
A	0
B	0.809524
B	0.904762
B	1
B	0.952381
B	0
C	0
C	0.677419
C	0.612903
C	1
C	0.935484
C	0.774194
C	0.354839
C	0.516129
C	0.0645161

```

1  from tqdm import tqdm
2  import multiprocessing
3  from joblib import Parallel, delayed
4  import pandas as pd
5
6  def apply_parallel(df_grouped, func):
7      """
8      Apply a function to grouped pandas dataframe using
9      multiprocessing.
10     Usage:
11     def apply_func(pandas_df):
12         ...
13
14     df = apply_parallel(
15         df.groupby(
16             by=grouped_by_columns,
17             as_index=False,
18         ), apply_func)
19     """
20     ret = Parallel(n_jobs=multiprocessing.cpu_count())(
21         delayed(func)(group) for name, group in tqdm(df_grouped)
22     ) # enumerate(tqdm(dfGrouped))
23     return pd.concat(ret)

```

We then use this helper function to perform our batchwise transformations

```

1  df_out_parallel = apply_parallel(
2      df.groupby(by="id", as_index=False), apply_func
3  )

```

This takes 677 ms, which is much slower than using the native `pandas.DataFrame.apply` function. This is because setting up multiprocessing has an overhead, so most of the process is used to set up the multiprocessing environment instead of performing the actual computation. However, when the size of the data and the number of batches are large, we can expect that the additional overhead can be worthwhile for the overall speed gain from multiprocessing. Let's now instead randomly generate 100,000 rows of data for our transformation.

```

1  import string
2  import random
3
4  df = pd.DataFrame({
5      "id": [
6          "".join(
7              random.choices(string.ascii_letters, k=3)
8          ) for _ in range(100_000)
9      ],
10     "scores": [
11         random.randint(1, 100) for _ in range(100_000)
12     ],
13 })

```

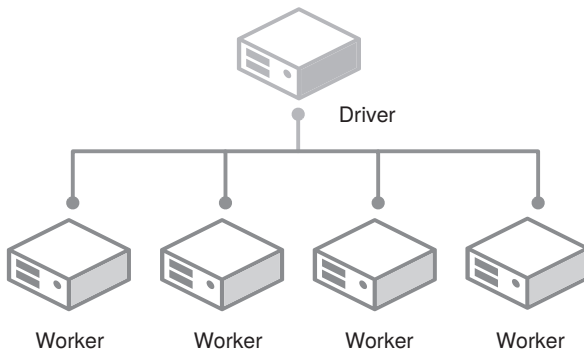


Figure 1.5 Driver and workers in distributed computing. A driver (or main) node issues a series of commands to create workers and assigns each worker a task to process a portion of the data. Once each worker completes their own task, the worker will send the results back to the driver for aggregation. The driver then waits for all the workers to finish their tasks before returning the results to the user.

If we use `pandas.DataFrame.apply`, the transformation will take about 19 seconds. If we use `apply_parallel`, it now takes 16 seconds to perform the transformation, a ~16% gain in processing speed.

1.3.3 Multiworker Distributed Computing

In modern big data analytics, practitioners often need to process data ranging from anywhere between several gigabytes and hundreds of terabytes in size (and this is often only the compressed size on disk; the size of the data occupying the memory can be much larger). No single consumer-grade laptop or desktop computer can hold hundreds of gigabytes to terabytes of data in memory, nor is any single CPU or GPU fast enough to process this much data without the user waiting hours to days (well, at least no such powerful processing unit exists yet as I am writing this book). Data analysts, data scientists, and machine learning engineers instead rely on cloud technologies to host such a large amount of data on distributed file storage systems and process the data using distributed computing. The term “distributed” here simply means multiple machines are storing, accessing, and processing the data, whereas machines communicate with each other via networking. Figure 1.5 illustrates a multiworker distributed computing system.

When dealing with the task of processing a large amount of data, a “divide and conquer” approach is key. The idea is if it takes 100 hours for a single computer to process our data, then 100 computers can finish the same task in about one hour. If we can somehow divide such a large amount of data into smaller pieces,

let each computing node processes that small piece of data, and finally aggregate the results at the end from all the parallel workers, it can then greatly reduce the physical (or wall) time we need to wait for our data processing task to finish. In fact, this is the core idea behind MapReduce [Dean and Ghemawat, 2008].

A “Hello World” example often used to illustrate MapReduce is the task of counting the number of occurrences of each word from a large body of documents. In this example, we will use Apache Beam [Apache Foundation, 2023a] to illustrate how to write a MapReduce algorithm for word count. We can install `apache-beam` with `pip install apache-beam` (I am using `apache-beam==2.49.0`. Note that Apache Beam currently does not support `pandas >=2.0.0` [Apache Foundation, 2023b] at the time of writing, so I am using `pandas==1.5.3` in this example).

```

1  import apache_beam as beam
2  from apache_beam.options.pipeline_options import PipelineOptions
3
4  # Setting up running environment
5  pipeline_args = [
6      "--job_name=word_count",
7      "--runner=DirectRunner", # local runner
8  ]
9  pipeline_options = PipelineOptions(pipeline_args)
10
11 # Run the beam pipeline
12 with beam.Pipeline(options=pipeline_options) as p:
13     _ = (
14         p
15         # Creating the example document
16         # This can also be read from files
17         | "Create texts" >> beam.Create([
18             "lorem ipsum dolor sit amet",
19             "sit amet dolor lorem",
20             "ut enim ad minim veniam",
21             "ut enim ad dolor lorem",
22         ])
23         # Splitting the documents into words
24         | "Split" >> beam.FlatMap(lambda x: x.split())
25         # Convert to a tuple (key, value) = (word, 1)
26         | "Pair with 1" >> beam.Map(lambda x: (x, 1))
27         # Summing the value by each key/word
28         | "Count" >> beam.CombinePerKey(sum)
29         # Print the output
30         | "Print" >> beam.Map(print)
31     )

```

- Lines 1–2 import the `apache-beam` library and the `PipelineOptions` class to configure the running environment.
- Lines 5–9 set up the pipeline running environment. Beam can automatically figure out when to use multithreading, multiprocessing, or distributed

computing depending on the specified runner type. In this small example, since we are running using `DirectRunner`, Beam assumes this is running on a single machine, so it will find the opportunity to use multiprocessing whenever appropriate. But, if we run the pipeline on a distributed environment such as Apache Flink or Apache Spark, Beam will use multiworker processing, while each worker will then also use multiprocessing if multiple cores are also available on each worker. This small technical detail, however, should not prevent us from understanding how MapReduce works in a distributed, parallel environment.

- Line 12 starts the context of the Beam pipeline, with the options we specified.
- Lines 14–22 initialize the pipeline by creating some mock documents. In this small example, we assume we have four documents. Each document contains words separated by spaces.
- Line 24 splits the documents into individual words. Here, `FlatMap` means applying the word splitting function to each document and then flat the results. So instead of returning the list of lists, it returns a flattened list.
- Line 26 performs the classic Map step of MapReduce, where it creates the tuple `(word, 1)` for each word in the flattened list. This is often described as the `<key, value>` map format, where the word is the key and the static value 1 is the value.
- Line 28 performs the classic Reduce step of the MapReduce, where it aggregates the tuples from the previous step by each key, summing together the values. This is equivalent to `.groupby(...).sum()` operations in `pandas.DataFrame`.
- Line 30 prints the results of the pipeline.

After executing the above pipeline, we have the following printout that counts the number of occurrences of each word:

```
( 'lorem', 3)
( 'ipsum', 1)
( 'dolor', 3)
( 'sit', 2)
( 'amet', 2)
( 'ut', 2)
( 'enim', 2)
( 'ad', 2)
( 'minim', 1)
( 'veniam', 1)
```

If the number and size of documents are both very large, we could imagine that each document will be read, split, mapped, aggregated, and returned as the above summary by each worker node. Then, the driver node can do further merging of the aggregated results from each worker to output the final results.

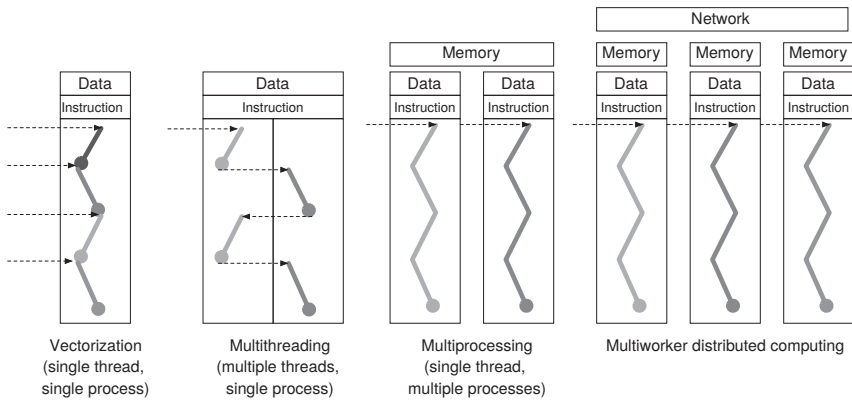


Figure 1.6 Comparison of parallel computing paradigms. Vectorization stores data within a single registry of the processing unit and processes multiple data representations on the registry using a single instruction simultaneously. Multithreading needs to switch between different threads of the processing unit and instructions are executed concurrently. Multiprocessing initiates multiple processes across different computing cores or different processing units on the machine, where data are communicated through shared memory. Multiworker distributed computing uses multiple machines to process a small piece of data, where the data are hosted on distributed file systems and communicated through networking.

Figure 1.6 summarizes and compares different paradigms of parallel computing we have discussed so far.

I hope this section provides enough information on different paradigms of parallel computing. It can be deduced that multithreading, multiprocessing, and multiworker paradigms are agnostic to the exact instructions of the program, whereas vectorization manipulates specific data using dedicated algorithms. Therefore, it is correct to expect that when combining vectorization with other types of parallel computing techniques such as multiprocessing and multiworker distributed computing, we can maximize our computing efficiency given limited resources.

Bibliography

- Apache Foundation. Apache Beam. <https://beam.apache.org/>, 2023a.
- Apache Foundation. Support Pandas==2.x in Apache Beam. #27221. <https://github.com/apache/beam/issues/27221>, June 2023b.
- Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, January 2008. ISSN 0001-0782, 1557-7317. doi: 10.1145/1327452.1327492.

- Alhussein Fawzi, Matej Balog, Aja Huang, Thomas Hubert, Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Francisco J. R. Ruiz, Julian Schrittwieser, Grzegorz Swirszcz, David Silver, Demis Hassabis, and Pushmeet Kohli. Discovering faster matrix multiplication algorithms with reinforcement learning. *Nature*, 610(7930):47–53, October 2022. ISSN 1476-4687. doi: 10.1038/s41586-022-05172-4.
- Intel. Get Outstanding Computational Performance without a Specialized Accelerator. *Intel Technology Brief*, July 2022.
- Evgueny Khartchenko. Vectorization: A Key Tool To Improve Performance On Modern CPUs. <https://www.intel.com/content/www/us/en/developer/articles/technical/vectorization-a-key-tool-to-improve-performance-on-modern-cpus.html>, January 2018.
- NVIDIA. NVIDIA’s Next Generation CUDATM Compute Architecture. Technical report, 2012.
- Volker Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13:354–356, 1969.

