**THE WORLD OF TENSORS**

# 

Table of Contents

# 

[**INTRODUCTION**](#_jhosz9fyrc6) **1**

[**THE TENSOR UNIT**](#_5exjwrgdq58f) **1**

[**TWO PARTY TENSOR BACKEND**](#_zcu34go65hew) **2**

[**TENSOR BEAVY OPERATIONS**](#_z2okdrqqipg3) **3**

[basic\_make\_arithmetic\_tensor\_input](#_4fekukxmbndp) 3

[basic\_make\_arithmetic\_tensor\_input\_shares](#_1rvfwet6o5ii) 5

[basic\_make\_arithmetic\_tensor\_output](#_9s1xsk2p6h1k) 5

[make\_tensor\_conversion](#_szt8cjof8sx0) 6

[make\_tensor\_conv2d\_op](#_5662l4o3k967) 6

[make\_tensor\_gemm\_op](#_94vb1h3vfy53) 9

[make\_tensor\_add\_op](#_uvo12yjbqbzi) 10

[**CONCLUSION**](#_16nvuym69z6m) **11**

| **NOTICE:**  The following documentation is created purely based on the interpretation and supervision of the MOTION2NX repository made by IUDX, and can be subjected to changes based on correction in existing information, addition of new discoveries and removal of redundant data. |
| --- |

# 

# 

# **INTRODUCTION**

The term **Tensor** is used to refer to a mathematical object, similar to an array, that can be used to represent scalar and vector quantities. They can have varying dimensions, different input types and so on. They play an important role when used to represent large or unique entities such as physical units, object sizes, image pixels and so on. The popularity of tensors lies in the fact that it becomes much simpler to perform complex operations such as multiplication or comparison of matrices of high orders of magnitude and dimensions.

The MOTION2NX repository is no stranger to the concept of tensors, though the definition takes a slightly different approach due to the specific requirements of handling the tensor values in the form of **shares** instead of normal quantities. Nevertheless, it allows us to perform matrix operations at a level comparable to the normal mathematical operations as done on numerals. In this document, we shall explore the definition of a Tensor in this repository, and the different terms and operations associated with the same.

# **THE TENSOR UNIT**

As shown in Fig 1, the Tensor is a structure that consists of multiple values, some of which retain the default values such as Number of channels or Num\_simd in majority of the operations. Along with the input itself, we can access the width, height and overall dimensions of the tensor we work with. In the MOTION2NX repository, the basic tensor library can be found in the **motioncore/tensor/** directory, which consists of the basic definition of the tensor and a list of operations that allow us to use and manipulate the tensors. Further, the tensor structure can be modified into a TensorP and TensorCP, which are shared pointer types for a Tensor and a constant Tensor respectively.

Apart from the above mentioned folder, the Tensor library extends to each protocol such as beavy and yao to implement their own unique functions using the Tensor values. Though the list of operations can be found in the original Tensor library, each world presents a different way of doing the same operations based on the type of inputs being handled.



Fig 1: The Tensor Unit

# **TWO PARTY TENSOR BACKEND**

This is a single operation similar to the two party backend with arithmetic operations. Its sole purpose is to initialise and prepare the servers by providing them with the list of all possible tensor backend operations, the communication setup with oblivious transfers, the structures and gates required for building the circuit operations, the protocols used and provides a monitoring service as the servers perform the necessary operations.

This can be found in the **motioncore/base/** folder, which in general consists of a variety of initialisation operation files similar to this one used for circuits, arithmetic operations, party-specific operations and so on. The importance of initialising this is so that no matter what operation the servers need to perform, this provides a single large database-like structure that calls every type of operation and tensor type. An example of the initialisation that can be seen in files is usually in the form of:

*auto create\_composite\_circuit(const Options& options, MOTION::****TwoPartyTensorBackend****& backend)*

This is followed by an initialization of the **get\_tensor\_op\_factory** in most cases, a file that uses the structure to execute a large list of tensor-based operations, as will be discussed in the next section.

# **TENSOR BEAVY OPERATIONS**

In this section we look into the list of operations that make use of tensors in the arithmetic or beavy world. They are commonly found in the **beavy\_provider.cpp** file, and the supporting classes can be found in the **tensor\_op.cpp**  file, both of which are under the **motioncore/protocols/beavy/** folder. The list of some of the operations are as follows.

## **basic\_make\_arithmetic\_tensor\_input**

This function is used by the party that is trying to create an input tensor. The function can either be *basic\_make\_arithmetic\_tensor\_input\_my* or *basic\_make\_arithmetic\_tensor\_input\_other* based on whether the party is creating the shares for itself or for the other party.

The function is initialised using the pair of classes **ReusableFiberPromise** of type **IntegerValues** and **TensorCP**. The first class is commonly seen among a variety of functions including arithmetic operation functions. It is used to create a mutex that holds the inputs, the shared state between the inputs and the CVType, which is the encryption type of the given inputs based on their size. Since they are all Integer values, the size can range from 16 to 64 bits.

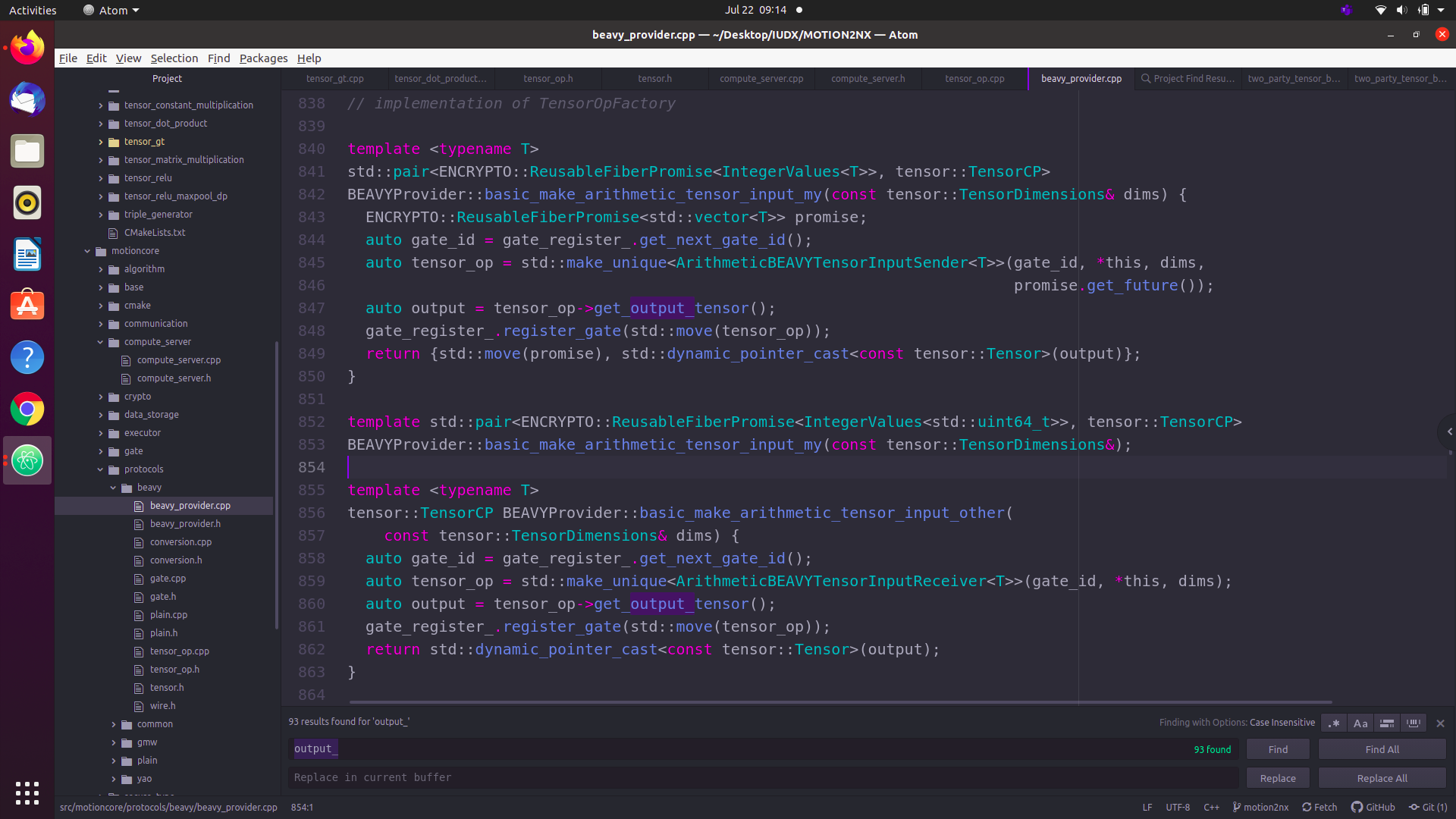


Fig 2: The basic\_make\_arithmetic\_tensor\_inputoperations

The second class is a shared\_pointer of a const Tensor where the input values once declared become immutable. This is so that in case of any errors between the code, the input values should remain unchanged. The reason behind having the declaration type as a pair is so that the function can return the share values in the form of promises, and the original input tensor in shares format as a **TensorCP**. The dimensions of the expected input tensor is given to the function as an argument of type **TensorDimensions**.

The first step is to retrieve the gate\_id for the input operation, then it creates a new empty tensor of type **ArithmeticBEAVYTensorInputSender.** This is another class as a part of tensor\_op.cpp where the inputs given by the user are transformed into shares format and placed in a single tensor. It consists of the **setup** phase and the **online** phase, similar to what has been explained in the Yao’s millionaires problem.

The **setup** phase declares the gateID, the dimensions of the new tensor, and generates the initial shares with a little help from the **Helpers:RandomVector** class to obtain a randomness generator. Once the private shares have been created, they are added into the public share variable. On coming to the **online** phase, the inputs are received, and this is also added to create the final version of the public shares. This is now visible to both parties while the private shares remain a secret within the respective parties.

Back to the function, we see that a function call to get the output tensor is created. This simply gives us the input tensor leaving out the delta shares and promise values created earlier. This is finally registered onto an input gate for future operations. The last return statement differs between the *input\_my* and *input\_other*, where in the case of the former, we return both the promise values created by the ReusableFiberPromise class as well as the input tensor, whilst for the latter, we only return the input tensor. In both cases, the tensors are dynamically casted as mutable tensor types to allow future operations to make changes in the input values as required.

## **basic\_make\_arithmetic\_tensor\_input\_shares**

The same as the previous function, the only variation in this function is that the shares for Delta and delta are created externally separate from creating with the tensors, and are allocated to a temporary variable. The return function remains the same as before, but now we have an additional variable created specifically for the share values.

## **basic\_make\_arithmetic\_tensor\_output**

With the operations being similar to that of its input counterpart, this function starts off with a different argument, where an input tensor is given as the argument, and the return type of the function contains only the **ReusableFiberFuture** from which we can extract the final output value.

Once we typecast the input into a regular beavy Tensor type, the gateID is retrieved for the output gate register, and the **ArithmeticBEAVYTensorOutput** class is used to obtain the output promise value. Once again this has a **setup** and an **online** phase. The **setup** phase is responsible for building the secret\_shares and storing them, so once the public shares are obtained in the **online phase,** the secret share can be subtracted from it and the final output value can be obtained, which is returned as a promise value to the output tensor in the function. The output gate is created as a final gate to be processed by the operation flow, and the output promise value is returned to the party requesting it.

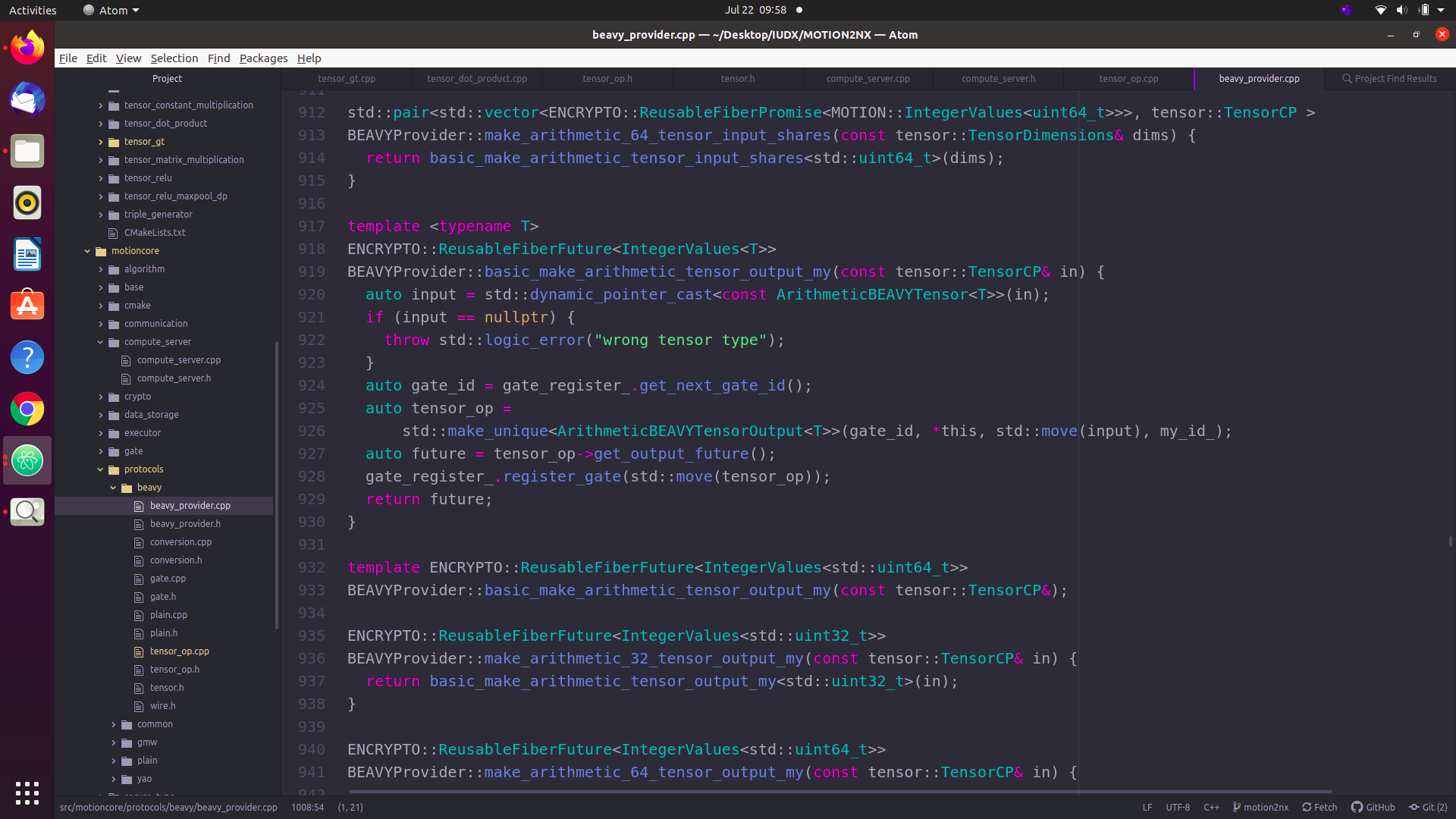


Fig 3: The basic\_make\_arithmetic\_tensor\_output function

## **make\_tensor\_conversion**

This is a basic function ( incomplete ) that has the potential to convert a tensor from the boolean world back into the arithmetic world. Though the function does not work, the underlying code provides an insight into how to go about building a conversion process in relation to the ABY2.0 paper.

## **make\_tensor\_conv2d\_op**

A crucial function in the world of Neural Networks, it allows the parties to add a convolution layer to perform operations of the same name. This is especially useful when dimensionality becomes an issue, or image processing applications come into picture. In addition to giving the tensor as the input, this function requires the kernel size it would be working on, the convolutional weights as well as the biases, and finally the fractional bits allowed for each input value.

After certain steps of error handling, the function creates a gate for the convolution operation, and initialises the kernel, input, and the bias as ArithmeticBEAVYTensor values. Next the **ArithmeticBEAVYTensorConv2D** class is called to create an output tensor that performs the convolution. In the **setup** phase, the secret shares of the two parties and the output are created and stored. On these delta shares, an initial **convolution** is performed to give us [Delta\_y]\_i = [delta\_a]\_i \* [delta\_b]\_i. This function found in the file **motioncore/utility/linear\_algebra.cpp** is unique since it utilises a new header file called **EIGEN.** This is an in-built C++ header file most commonly used to perform a variety of linear algebraic operations, some of which includes its own interpretation of tensor values called CTensors for convolution types.

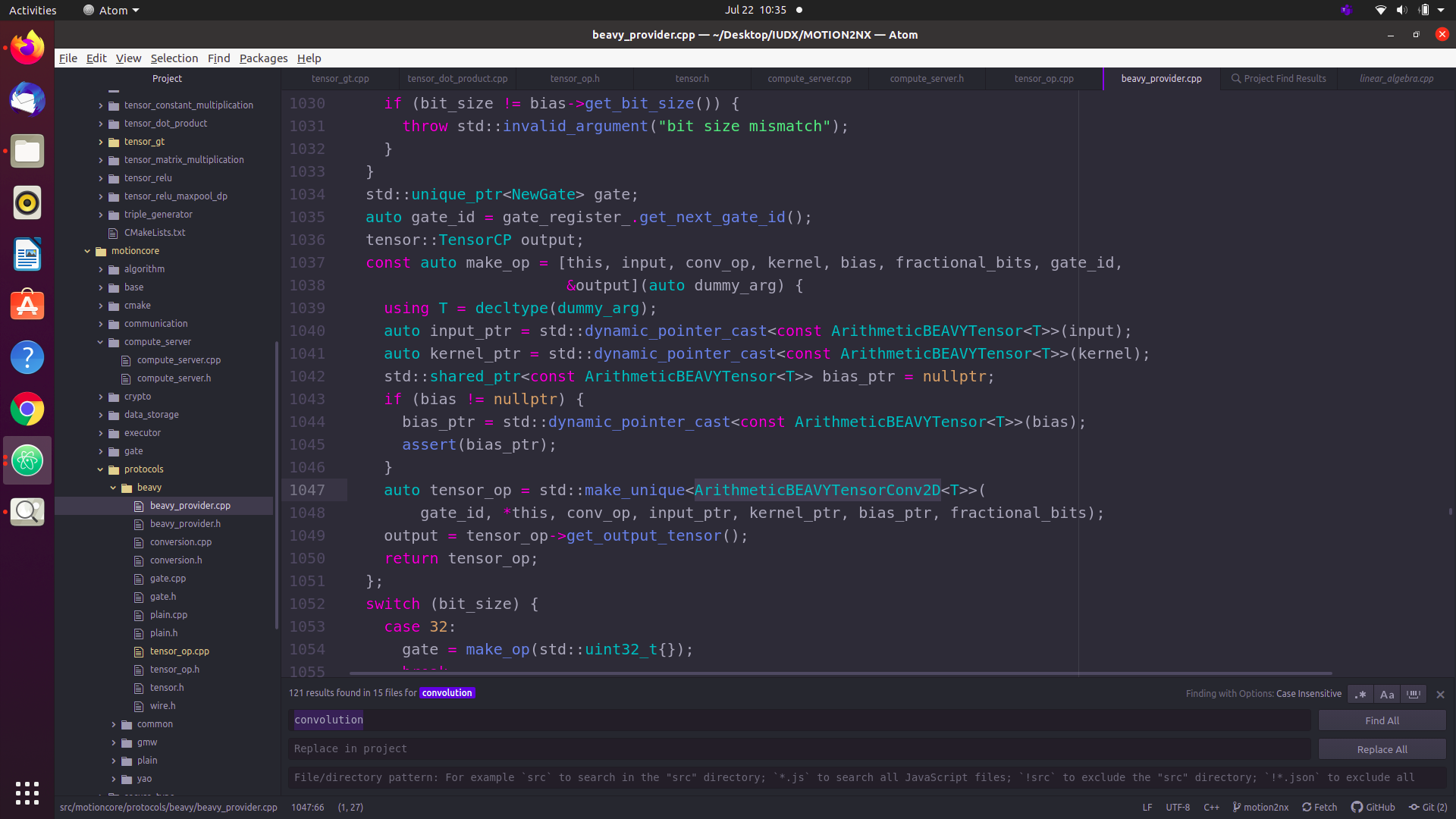


Fig 4: The make\_tensor\_conv2d\_op function

The convolution function is specifically created for image classification, having variables such as strides, dilations and pads of image pixel values being taken care of within the function. To give a top level overview, the input values such as kernel, bias and so on are type casted into eigen CTensors, and the input values are split into the variables mentioned above and an in-built eigen convolution using alternate contract and shuffle operations is performed, and the final output is returned to the convolution class.

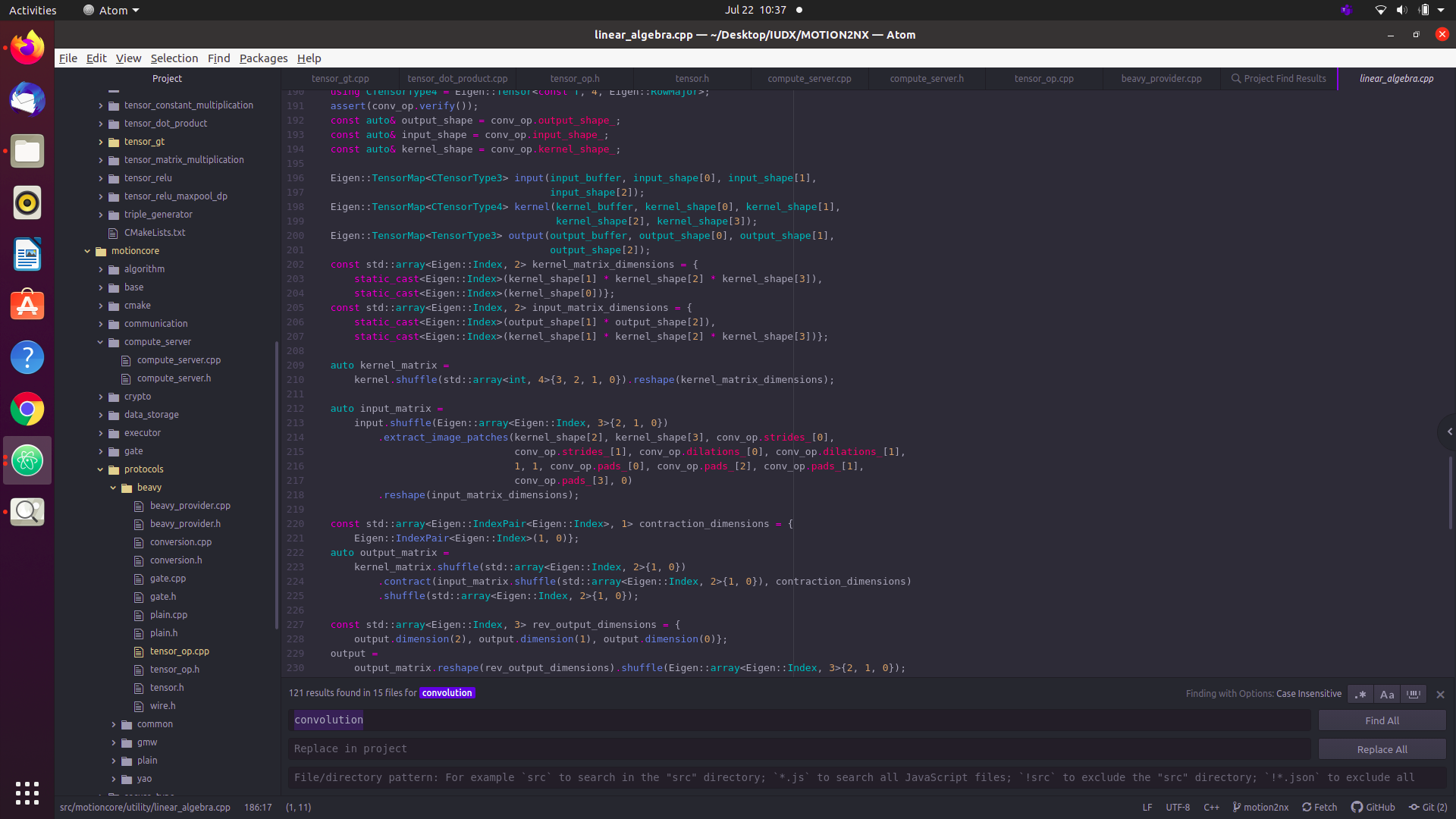


Fig 5: The convolution function using EIGEN

So now, we have the shares convoluted for the Delta y output value, and it is stored in a temporary variable until the **online phase** begins. The ab shares are created using the **RandomVector** function from before. Each of these is added with the DeltaY share using transforms.

Once the online phase begins, the input tensors, kernel, bias values and so on are obtained from the parties, and two convolution followed by transform operations are performed for partyA and partyB to obtain and convolute the values from the DeltaY share set on their respective input values. These are the cross terms being multiplied since the delta values were done in the setup phase itself. A final transform after a truncation function performed post convolution binds all the shares back into the DeltaY share, and the gateID along with the output DeltaY share is broadcasted to all the parties. ( The function was too big for an image, but it is found in tensor\_op.cpp under ArithmeticBEAVYTensorConv2D class ).

## **make\_tensor\_gemm\_op**

An important function, the **general matrix multiplication** or **gemm** for short, is responsible for performing the matrix multiplication and dot product operations using the in-built tensors. Similar to the convolution function, the gemm operation initialises the four input values: the gemm\_output, the two inputs from each party and the number of fractional bits allocated. A new gate is created for the gemm operation to be added to the final operation flow, and finally the **ArithmeticBEAVYTensorGemm** class is called to create the output tensor for the gemm operation.

The above class follows a pattern similar to ArithmeticBEAVYTensorConv2D but instead of a kernel or bias input, before the setup phase is declared two sides of a matrix multiplication operation: the LHS and the RHS. This aids the **setup phase** to create a distinction between matrices, and a location to store the values once the operation for each set of input values is performed.

In the setup phase, Shares for party A, B and output Y are created, and the **matrix\_multiply** function is called to perform an initial setup multiplication with the shares. This is once again present in the linear\_algebra.cpp file, and is overridden in 4 different ways, each performing an initialisation of inputs, input mapping or an Eigen mapping for multiplication. The matrix output size is initially set using the dim (dimension) variables, mapped to the correct values by Eigen::Map, followed by an arithmetic multiplication between the two input matrices into this output matrix.

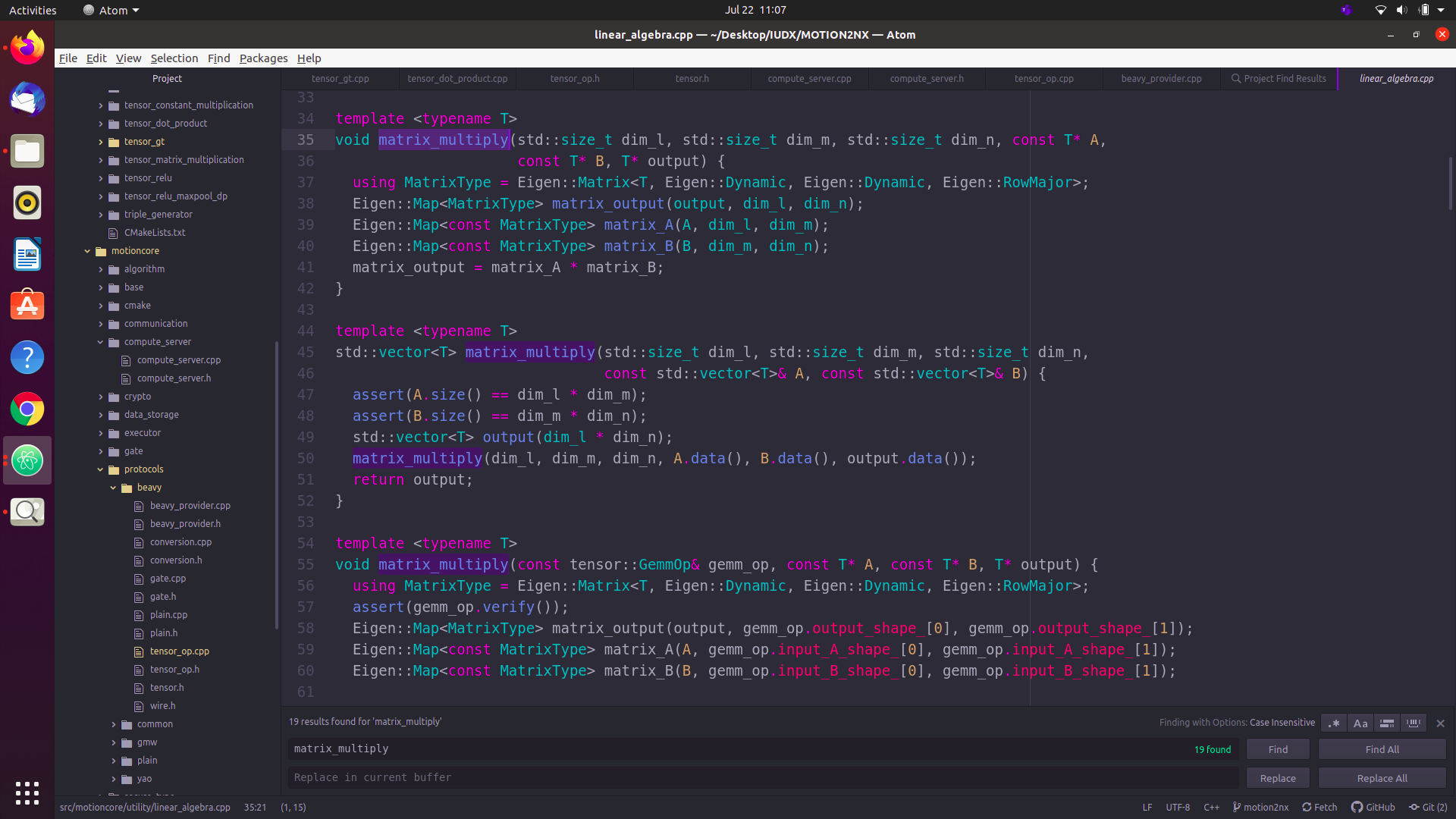


Fig 6: The matrix multiplication function

Once this process is done for the DeltaY shares in the setup phase, the DeltaA and DeltaB shares are created using **RandomVector** and added with the DeltaY final shares. In the **online phase,** the matrix\_multiply and transform occur simultaneously to execute the following:

1. [Delta\_y]\_i -= Delta\_a \* [delta\_b]\_i
2. [Delta\_y]\_i -= Delta\_b \* [delta\_a]\_i
3. [Delta\_y]\_i += Delta\_ab (== Delta\_a \* Delta\_b)
4. [Delta\_y]\_i += [delta\_y]\_i

These steps are done to multiply the cross terms and add them to the final output value, same as in the case of convolution. Finally the gateID and the output DeltaY shares are broadcasted to both parties.

## **make\_tensor\_add\_op**

One of the simpler functions present, this function is used to add two tensors together. The function starts off similar to the multiplication functions, where two inputs are given as the argument and having a TensorCP return type. Once a new gate is created for the process, we call the **ArithmeticBEAVYTensorAdd** class to perform the addition operation.

The **setup phase** for this class is the same as seen earlier - generating the Delta shares and adding them together into a public share variable. This upon reaching the **online phase** gets incremented by the value of the inputs provided by the parties. The final output tensor is given out to the function, which returns it to the main function call.

Similar to this function is the **make\_tensor\_negate\_op**, where instead of adding the two values, the process simply replaces the std::plus() with std::minus() in the transform function.

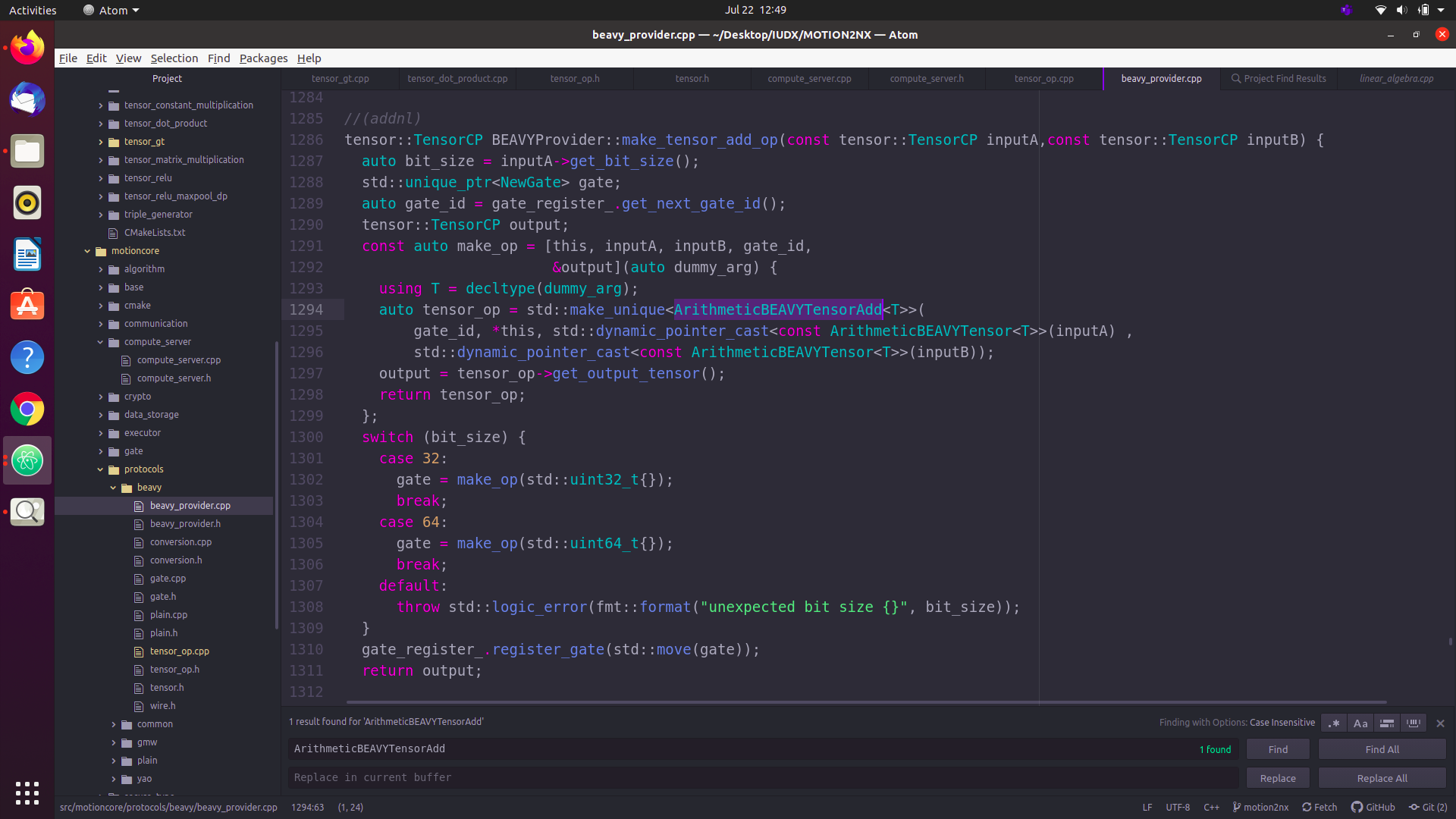


Fig 7: The addition operation

# **CONCLUSION**

Though the number of functions explained here are limited, more functions such as TensorMul and Join operations have not been used or explored yet. The applications that can make use of these functions are numerous, and a deeper understanding of them may allow us to improve the efficiency of earlier examples too.