Hardware Accelerator for Vector Multiplication and Matrix Multiplication

Language: Bluespec

Overview of the Project

This is a report for the **Domain-Specific Hardware Accelerators** for **Vector Multiplication and Matrix Multiplication** written in Bluespec. There is one package for vector multiplication and two packages for matrix multiplication. The matrix multiplication packages vary slightly, and the first one (MMHA 1) is more time and area optimised. The second package (MMHA 2) is more modular, and various modifications can be done with ease. The core module of the second package allows the CPU to pass the elements of the matrices directly without having to interface with the memory unit. Similarly, the vector package's core module can also be used directly by the CPU. Both the matrix multiplication packages utilise the core module of the vector multiplication package for efficient processing and calculation.
The hardware accelerators take in the address and size of the inputs (vectors or matrices) and compute the respective products (a scalar or matrix respectively) and store them in the memory at the address inputted. All the packages allow using any data type as input, and in our packages, we have used integer inputs and can be easily switched to enable floating-point inputs. Other data types may require more modifications to the code. We have predefined the sizes for the address and data lines (i.e., **8 Bytes for address and 32 bytes for data**).
The memory module we have used is a simple memory module which allows either reading or writing in any given cycle. All the main modules of the packaged can access the memory to get the values and to store the final result. This increases the efficiency as if a very complex and efficient CPU had to do the same it would every time store each product in the memory and retrieve it for addition and store the partial sum again and so on. In a practical case, the memory module takes about 100s of cycles to access. Hence a Hardware Accelerator is a must-have for a CPU for vector and matrix multiplications.

The Vector Multiplication Package

The Vector multiplication package represents our hardware accelerator for multiplying two vectors of equal length. The size of the vector is dynamic and the maximum size, which determines the number of registers used, can be controlled in the initial type definitions.
The Vector Multiplication Package consists of the following modules:

1. **The Memory Module:** It is common to all packages. It has been explained later in the report
2. Vector Multiplication Part (divided into two modules):
    a. **Inner Vector Multiplication Module:** This is the core module of the vector package. It does the calculations and returns the output to the outer module to be stored in the destination address given by the CPU.
    b. **Outer Vector Multiplication Module:** It retrieves data from memory and provides the inner module with the data elements to do the calculation and also stores the final result.
3. **Vector Multiplication Test Bench Module:** Test Bench to pass the addresses of the vectors in the memory and their size. It also checks the status of the vector multiplication by reading the control status register (CSR) of the module via back door interfacing.

a) Working of the Outer Vector Multiplication Module

The **outer module** receives the addresses of the first elements of both the vectors and the address where the final value is to be stored. It also gets the size of the vectors. Using the received addresses, it retrieves the vector elements from the memory one by one and then passes these values in pairs to the inner vector module and waits for the inner module to finish its task. Once the computation is over, it stores the value at the destination address in the memory. The status of the multiplication can be obtained by reading the CSR of the vector outer module. The outer module can be divided into three stages as follows:

1. **First stage:** It checks if all the input addresses and sizes are valid. If everything is valid, the outer module moves onto the second stage.
2. **Second stage:** This stage uses the input addresses to fetch the elements of both the vectors one by one from the memory. Elements of the vectors are accessed alternatively and passed to the inner module. The retrieved values are not stored in any register and are directly passed to the inner module.
3. **Third stage:** Once the computation is done the final dot product is received from the inner module and it is directly passed to the memory to be stored in the given destination address.

b) Working of the Inner Vector Multiplication Module

This is the core module which performs the vector dot multiplication in a pipelined manner. The module has three input ports and two output ports. Two input ports get each element of the vectors and the third input port tells the module if the input values are the last pair of elements of the vectors. The inner module multiplies the pair received from the outer vector multiplication module and passes the product to the next stage to be added to a sum register which returns the sum based on the flag from the third input. The outer module checks the flag from the inner module to know if the computation is over. The inner module has three stages:

1. **First stage:** This stage has a single rule for multiplying the received vector element pair. It multiplies the pair and passes the product to the next stage. A flag which indicates if the pair of values passed are the last of the vectors is also passed along with the product.
2. **Second stage:** At this stage the flag is checked to see if they are the last elements of the vectors and based on that the final result may be returned to the outer module. Two registers are used to store the accumulated sum, one for passing to the next stage and the other to be used for further accumulation of sum of products. Naturally, the latter is initialised to zero when the flag indicates that the last values have entered. A flag is also passed to the next stage when the dot product has been computed.

   In this stage, the previous stage product is added to a register which stores the accumulated sum. If the flag from the previous stage is True, then the final sum of products is stored in another register which is then passed for output. Boolean True is also passed to indicate the outer module that the product is ready.
3. **Third stage:** The flag from the previous stage is used to enable passing the output to the outer module. The flag also is passed through a method which the outer module can use to know if the output is ready or not.

Matrix Multiplication Hardware Accelerator Package 1 (MMHA 1)

This package represents our hardware accelerator for multiplying two matrices of the same dimensions. Inner module of vector multiplication is used by the matrix module for computations. The size of the matrix is dynamic and the maximum size, which determines the number of registers used, can be controlled in the initial type definitions. We assumed the max size of the matrix to be 10, thereby fixing the maximum number of elements in a single matrix to be 100. The advantage of using matrix multiplier over vector multiplier multiple times is that it uses the core module of the vector package efficiently to compute the elements of the resulting matrix. For a 3X3 matrix, the vector package alone will take about twice as much time as the matrix package.

MMHA 1 package consists of the following modules:

1. **The Memory Module:** It is common to all packages. It has been explained later in the report.
2. **Matrix Multiplication Module**: This is the crux of the package. It fetches all the elements of the matrices from the memory, enables computations, and stores the result in the memory. An assumption was made that the first matrix is present in the memory contiguously and row-wise and the second matrix is present in the memory contiguously and column-wise. The resulting matrix is stored row-wise in the memory.
3. **Inner Vector Multiplication:** This module is the same as the one used for vector multiplication. Pipelining of this module greatly reduced the time taken for computation of the resulting matrix.
4. **Matrix Multiplication Test Bench Module:** Test Bench to pass the addresses of the matrices in the memory and their size. It also checks the status of the matrix multiplication by reading the control status register (CSR) of the module via back door interfacing.

Working of the Matrix Multiplication Module

The **matrix module** receives the addresses of the first elements of both the matrices and the address where the resulting matrix is to be stored. It also gets the size of the matrices. Using the received addresses, it retrieves the vector elements from the memory one by one and stores them. Then the values of the elements are passed in pairs to the core module of vector multiplication and the resulting values are stored and then passed to memory. The status of the multiplication can be obtained by reading the CSR of the matrix module. The matrix module can be divided into three stages as follows:

1. **First stage:** After the size of the matrix is received, the number of elements in the matrix is calculated. Then the elements of the matrices are retrieved from memory one by one in an alternating manner. If the inputs are invalid, the CSR gets the value 4 and the whole module is reset.
2. **Second stage:** The elements are passed pair-wise to the core module for vector multiplication. The vectors for computation are passed sequentially with every column of the second matrix passed for every row of the first matrix. This happens very quickly as the core module is pipelined. The elements of the resulting matrix are got and stored as and when they are ready.
3. **Third stage:** Once all the elements of the resulting matrix have been got, the values are passed to the memory for storage. Since it takes two clock cycles for the memory to store, a buffer rule has been used which also resets the module once all the elements have been stored in the memory.

Matrix Multiplication Hardware Accelerator Package 2 (MMHA 2)

This package represents our modular hardware accelerator for multiplying two matrices of the same dimensions. The basic logic of the working of this package is the same as MMHA 1 and the only difference being is that the matrix multiplication is done in two separate modules, the outer module and the inner module.
The advantage of this package is that it provides a separate module for computations. The inner module for matrix multiplication stores the elements of the matrices and uses the core module of the vector package for computations. The constraint on this module is that it requires all the elements to be passed continuously. This is to avoid steps involving computing the total number of elements of the matrix.
Fail checks have been used to check if all the elements have been got and they occur in parallel to all the stages. The size of the matrix is dynamic and the maximum size, which determines the number of registers used, can be controlled in the initial type definitions. We assumed the max size of the matrix to be 10, thereby fixing the maximum number of elements in a single matrix to be 100.
MMHA 2 package consists of the following modules:

1. **The Memory Module:** It is common to all packages. It has been explained later in the report.
2. **Outer Matrix Multiplication Module**: This module It fetches all the elements of the matrices from the memory and also passes the resulting matrix to store it in memory. An assumption was made that the first matrix is present in the memory contiguously and row-wise and the second matrix is present in the memory contiguously and column-wise. The resulting matrix is stored row-wise in the memory.
3. **Inner Matrix Multiplication Module**: This is the core module for matrix multiplication. It receives all the elements of the matrices from the outer module and uses the inner vector module for computing the elements of the resulting matrix. It includes a failure check to see if all the elements have been got from the outer module. This module can be integrated with other modules which require a matrix multiplier.
5. **Inner Vector Multiplication:** This module is the same as the one used for vector multiplication. Pipelining of this module greatly reduced the time taken for computation of the resulting matrix.
6. **Matrix Multiplication Test Bench Module:** Test Bench to pass the addresses of the matrices in the memory and their size. It also checks the status of the matrix multiplication by reading the control status register (CSR) of the outer module via back door interfacing.

a) Working of the Outer Matrix Multiplication Module

The **outer module** works very similar to the matrix module of the MMHA 1 package. The key difference is that the second stage of this module passes the elements of the matrices to the inner matrix module and also gets the elements of the resulting matrix in the same stage. When the inner module has completed all the calculations, the outer module gets the values and stores them in the memory at the destination address inputted. The outer module can be divided into four stages:

1. **First stage:** Exactly like in MMHA 1, after the size of the matrix is received, the number of elements in the matrix is calculated. Then the elements of the matrices are retrieved from memory one by one in an alternating manner. The only difference is if the inputs are invalid, the CSR gets the value 5.
2. **Second stage:** The elements are passed in pairs to the inner module and the elements of the resulting matrix are also computed and stored in the same stage. The inner module has a flag to tell if the inputs were invalid.
3. **Third stage:** After the second stage is completed, the resulting matrix is stored row-wise in the memory. As writing into memory takes two cycles, a buffer rule has been used which also doubles as the reset rule to reset the module once all the values are stored.

b) Working of the Inner Matrix Multiplication Module

The **inner module** enables computation in a similar manner to the matrix module of MMHA 1. It gets the elements from the outer module and stores them. After receiving all the elements of both the matrices, it passes the values to the vector package's inner module to do the computations. The output from the vector module is stored and then passed to the outer module once all the values have been computed. The inner module also consists of three stages:

1. **First stage:** Pair of valid inputs are received in a continuous manner and stored. The size of the matrix is also received and it is used to check if the number of elements is the square of the matrix size. This is a failure check for the module and if it fails then the module gets reset to its initial state and no output is given.
2. **Second stage:** This is the crux of the MMHA 2 package. The elements are passed pair-wise to the core module for vector multiplication. The vectors for computation are passed sequentially with every column of the second matrix passed for every row of the first matrix. This happens very quickly as the core module is pipelined. In this same stage, the elements of the resulting matrix are also got and stored.
3. **Third stage:** In this stage, the output matrix size flag is used to determine if the multiplication process is finished, then the values of the resulting matrix are passed to the outer module row-wise and a flag is raised to indicate to the outer module that the output is ready. Lastly the whole module is reset for next matrix multiplication.

Comparing and Understanding MMHA 1 and MMHA 2

The basic logic used for matrix multiplication is the same for both the packages but the MMHA 2 is less optimised as it has a separate core module for the matrix multiplication. The main difference is in the application of the two modules. Both modules can be integrated with a simple memory unit but MMHA 2 can be easily integrated with other types of memory modules. For example, if the memory unit can read and write in parallel, then the core matrix module of MMHA 2 can be more easily used as compared to the matrix module of MMHA 1. There are other cases where the hardware units might require only the matrix multiplier and do not need to access the memory to retrieve the data. Our core matrix module of MMHA 2 does exactly this.
MMHA 2 package uses more registers and takes slightly more time to compute the resulting matrix (about 10 percent more). This delay can be reduced if more parallel computations can be done.
Both of the packages follow the same matrix multiplication logic:

Logic for matrix multiplication:

Let the input matrices be A and B and the output matrix be C such that $A * B = C$.

1. First the first element of the matrix A and matrix B are passed to the vector module.
2. Then the second elements of the matrices are passed and this repeats till all the elements are passed. Here the matrix A is stored in the module in a row-wise fashion and the matrix B is stored in the module in a column-wise fashion.
3. A flag is also passed along with the pair of elements to indicate if that pair is the last of the column or row.
4. The core vector module raises a flag each time a dot product has been calculated and returns it to the outer module.
5. This repeats till all the elements of the resulting matrix are computed.

We are using only one vector multiplication module at a time to limit the hardware required in the actual accelerator. Adding more vector multiplication modules would increase the complexity of the

module greatly and would require more registers. There is also no generalization and the maximum size needs to be fixed. Since the maximum size of a matrix of our package can be changed easily at the type definitions, we haven't used multiple vector multiplication modules.

The Memory Module

The memory module is a part of both the vector and matrix multiplication packages. We have taken the memory to be of 256 bytes (2^8 bytes) for both the packages and it can be changed easily in the type definitions. The memory module includes rules for both reading from and writing into the memory. It is ensured that these two rules are never fired in the same cycle thereby not allowing any module to read and write simultaneously. It takes three cycles to read and two cycles to write. Mode of operation of the memory is controlled by the control register of the memory module. This register can be both read and written via back door interfacing.
The **CR** has three possible states:

1. 1 0: Memory Enable, **read mode**
2. 1 1: Memory Enable, **write mode**
3. 0 X: Memory Disable

We have initialized the memory with a series of values (in arithmetic progression). The memory has four interfaces which can be accessed via front-door interfacing:

1. **MAR**: Method to pass an address to the memory.
2. **MODR_ready**: Method to check if the read output is ready.
3. **MODR**: Method to read the data stored in some address in the memory. It shares the same flag as MODR_ready() and returns a value only if the value is ready.
4. **MIDR**: Method to pass a data value to be stored in some address.

Libraries and Bluespec Packages used:

We have imported and used the following Bluespec libraries:

1. CBus package:

CBus package is a predefined library in Bluespec. Complex designs typically include a back door mechanism for reading and writing the values of certain control status registers contained in the design. Each register that is to be accessed in this way has an associated address (unique identifier). The library package requires the registers to be of type CRAddr#(numeric type sa, numeric type sd). The register is of struct data type and has two components 'a' and 'o'. The size of such a register is:

$$size\ of\ the\ register = sa + (sd)$$

2. FloatingPoint Package:

FloatingPoint package is a predefined library in Bluespec. It provides all the basic computations like addition and multiplication of floating-point values. Floating-point values are stored in variables of type FloatingPoint#(e,m). In our packages, it is ideal to use "Float", a predefined data type in the library using FloatingPoint#(e,m), as it is of size 32 bits with 8 bits for the exponent part and 23 bits for the mantissa part. To display the values, use fshow(<variable>) which returns a format string.

3. Vector Package:

Vector Package is a predefined library in Bluespec. It provides the Vector type which allows storing arrays of any data type. It also provides the predefined function replicateM() to initialise in the declaration statement.

Possible Improvements To our Matrix Multiplier Accelerators

We have optimised our accelerators to reduce the hardware cost, but it leads to an increase in time complexity. If the maximum size of the matrix is fixed, then all the elements of the resulting matrix can be computed in parallel or as and when the values are obtained from the memory. So, for a matrix of size 10, 100 vector modules will be used by the core matrix module to compute the elements of the resulting matrix.
Here the package can be optimised as per the real situation.
If the memory access is fast enough, the same values can be gotten from the memory multiple times to avoid storing the elements. For a 3X3 matrix, each element of the first matrix will be retrieved from the memory every time an element of a new column of the second matrix is being retrieved. So totally the memory will be read 36 times.
If instead the memory access is slow, the elements of the matrices need to be stored as and when they are got and then passed to the appropriate core module. In both cases, the resulting matrix need not be stored and can be directly sent for storing in the memory.

Switching between Integer and Floating Values

The packages allow easy switching between integer inputs and Float inputs. Float is a predefined floating-point data type of size 32 bits. Floating-point values are displayed in hexadecimal notation. The user needs to uncomment some codes in the package and also comment the related ones. The code that needs to be uncommented has been mentioned in the code and they are in the following parts of the modules:

1. **Importing the FloatingPoint library** – This has been commented to save compilation time.
2. **Initializing the memory with floating points instead of integers** – Outside the memory module two variables have been declared which need to be initialized with floating point values instead of integer values.
3. **Commenting and uncommenting the respective display rules** – Display rules have been used to display the input and output vector or matrices. They are present in the modules which access the memory. Floating-point values are displayed by using the predefined function ‘fshow()’ which returns a format string.

Code Execution

All the packages have been executed using Bluespec's bsc compiler. Note that the warnings displayed during compilation have all been verified and can be removed by simply adding appropriate rule attributes but it slows the compilation by minutes. The following commands are for executing the packages using bsc compiler.

A) Vector Multiplication Package

Commands to be used to execute the Package in Ubuntu Terminal:

- bsc -verilog Vector_Multiplication.bsv
- bsc -o sim -e mkTestBench mkTestBench.v
- ./sim

This is how the results will be displayed:

```
Address of A[0] is   4 and address of B[0] is   2.
The destination address is 100. The size of the vectors is          3.
The time at which the addresses are sent is                    25

Vector multiplication has been completed.
time at which it completed -                  265


a[          1] =         29
a[          2] =         35
a[          3] =         41


b[          1] =         17
b[          2] =         23
b[          3] =         29


Dot product =       2487
```

In the above picture, $r[i] = c$, means that i^{th} element of vector r is equal to c. Where a and b are the input vectors.

B) Matrix Multiplication (MMHA 1) Package:

Commands to be used to execute the Package in Ubuntu Terminal:

- bsc -verilog Matrix_Multiplication1.bsv
- bsc -o sim -e mkTestBench mkTestBench.v

- ./sim

This is how the results will be displayed:

```
Address of A[0,0] is   4 and address of B[0,0] is   2.
The destination address starts at 100. The size of the matrices is          3.
The time at which the addresses are sent is                   25
a[          1,          1] =          29
a[          1,          2] =          35
a[          1,          3] =          41
a[          2,          1] =          47
a[          2,          2] =          53
a[          2,          3] =          59
a[          3,          1] =          65
a[          3,          2] =          71
a[          3,          3] =          77


b[          1,          1] =          17
b[          1,          2] =          35
b[          1,          3] =          53
b[          2,          1] =          23
b[          2,          2] =          41
b[          2,          3] =          59
b[          3,          1] =          29
b[          3,          2] =          47
b[          3,          3] =          65


r[          1,          1] =        2487
r[          1,          2] =        4377
r[          1,          3] =        6267
r[          2,          1] =        3729
r[          2,          2] =        6591
r[          2,          3] =        9453
r[          3,          1] =        4971
r[          3,          2] =        8805
r[          3,          3] =       12639

Matrix multiplication has been completed.
Time at which it completed -                1145
```

In the above picture, $r[i, j] = c$, means that element at the i^{th} row and j^{th} column of matrix r is equal to c. a and b are the input matrices and r is the resultant matrix.

C) Matrix Multiplication (MMHA 2) Package:

Commands to be used to execute the Package in Ubuntu Terminal:

- bsc -verilog Matrix_Multiplication2.bsv
- bsc -o sim -e mkTestBench mkTestBench.v
- ./sim

The results are displayed similar to the way results are displayed in MMHA 1 package. The number of cycles taken to complete is 1285 cycles.

Integrating With The CPU

To integrate the packages with the CPU, the following are the steps to be done:

1. Import the respective packages.
2. Declare an interface using the keyword ‘let’ and use it to instantiate the appropriate synthesizable CBus module.
3. This interface is used as an alias to interface with the module.
4. Front-door interfacing is by using the alias interface followed by “.device_ifc” and back door interfacing is by using the alias interface followed by “.cbus_ifc”.
5. All the packages use similar methods – address of first input, address of second input, destination address for final result and size of the inputs – which are accessed through front-door interfacing
6. Back-door interfacing is available for the control status registers whose associated addresses are given in the packages.

Project Submitted by:

EP17B002 - Aman Singhal

CS18M026 - Kamal Kishore

EE19B121 - Surya Prasad S