Hardware Accelerated Vectorized Gradient Descent for Linear Regression

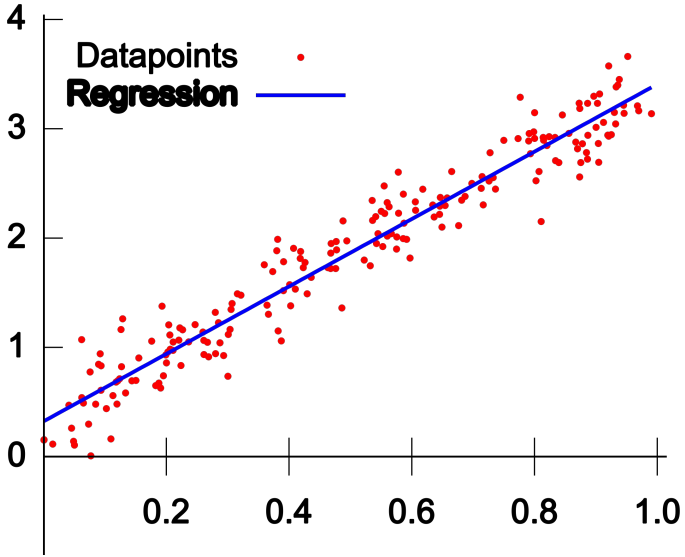
José Castanheira, 76545
Fábio Maia, 76380

Linear Regression

Linear regression is a linear statistical model between independent variables **X** and dependent variables **Y** that is parametrized by **θ** and expressed as

$$h_\theta(X) = \theta_0 + \theta_1 X_1 + \ldots + \theta_n X_n$$

where **n** is the number of features in the independent variables.



What is Gradient Descent?

Gradient descent is an iterative algorithm used to determine the minimum of a function, frequently used in the context of linear regression.

In linear regression, one's aim is to determine the line that best fits the points on the plane. The difference between the real points' label and the model's prediction is called the **cost**. The cost can be seen as a function of each line.

Gradient descent is used to determine the minimum of the cost function, and therefore, the line that best fits the points.

Gradient Descent Algorithm

The gradient descent algorithm is as follows:

repeat until convergence: {

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)} \qquad \text{for j} := 0...n$$

}

Each element θ_j of the θ vector is updated in each iteration, and the algorithm will stop when we decide the algorithm has converged.

repeat until convergence: {

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)} \qquad \text{for j} := 0...n$$

}

- **θ** is the parameters the algorithm intends to calculate and improve in each iteration;
- **α** is called the learning rate, and is usually a small value (e.g. 0.001) to allow the convergence of the algorithm;
- **m** is the number of data points, also called training samples;
- $\mathbf{x^{(i)}}$ and $\mathbf{y^{(i)}}$ constitute the “i-th” data point, or training sample and respective output value;
- $\mathbf{h_\theta(x^{(i)})}$ is the algorithm’s prediction for the respective training sample, using the current **θ** parameters

Vectorized Gradient Descent

As seen, the iterative algorithm updates each θ_j of the θ vector at a time, and does this by using each element of the training set at a time. Another approach would be to calculate all the θ vector in one turn, by looking at all the elements of the training set at the same time. This yields the following vectorized form of gradient descent:

$$\theta := \theta - \frac{\alpha}{m} X^T (g(X\theta) - \vec{y})$$

Hardware Accelerator

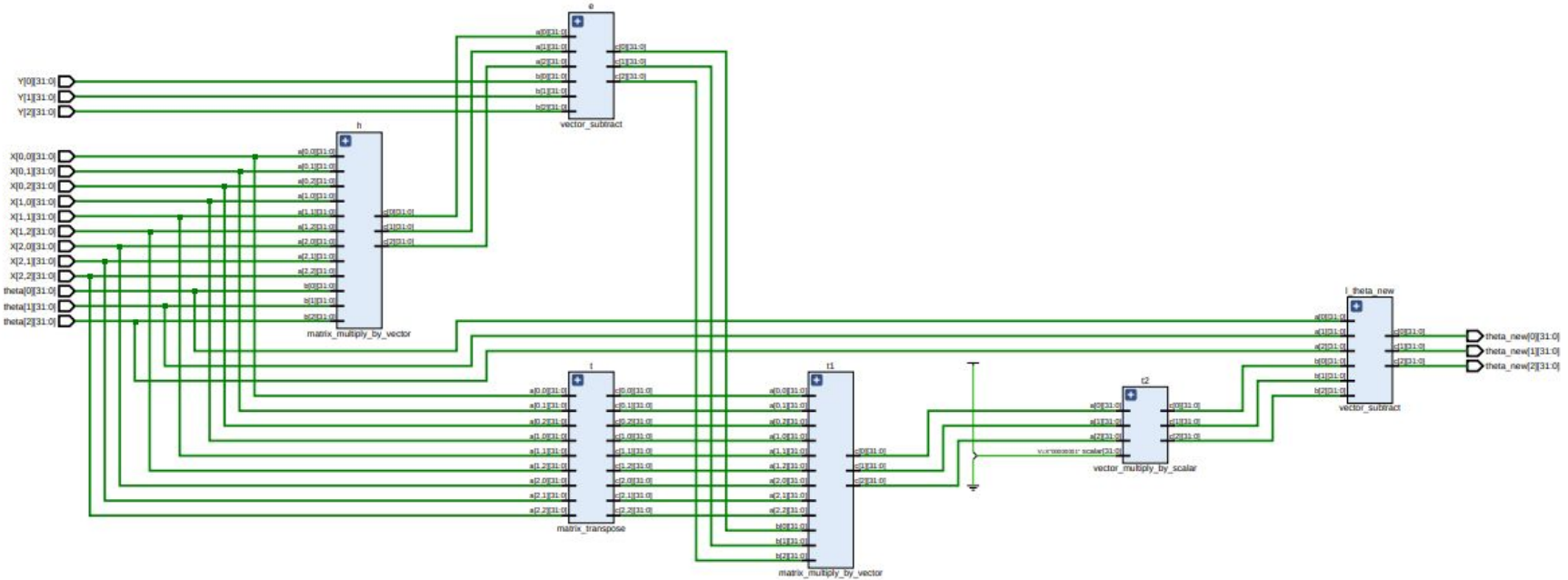
Hardware Accelerator

The vectorized gradient descent is what is implemented in the hardware accelerator. It is a combinatorial block with 4 inputs (parameter vector **θ**, feature matrix **X**, label vector **Y**, and scalar learning rate **α)** and 1 output **θ** vector.

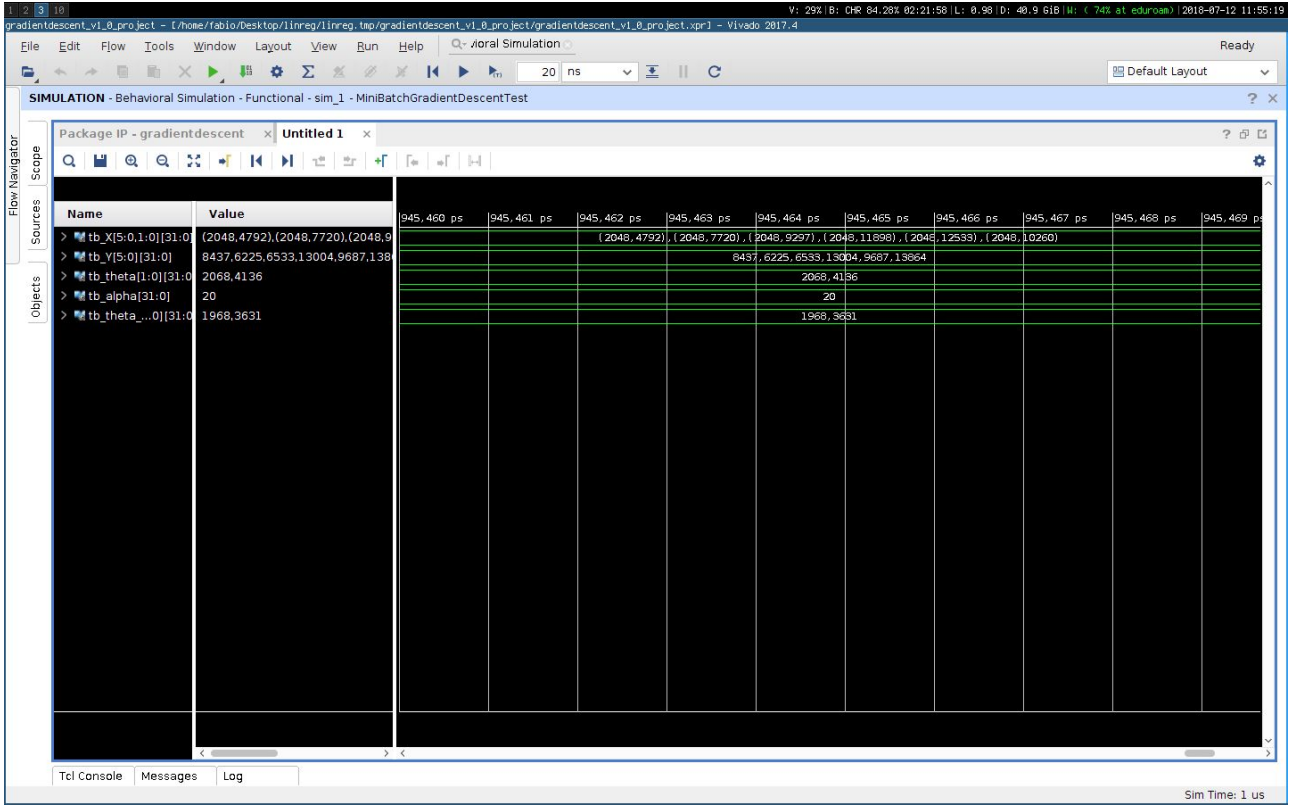
To compute the updated **θ** it must perform

- matrix by vector multiplication
- matrix by scalar multiplication
- vector subtraction and addition
- matrix transpose

Hardware Accelerator - Block Diagram



Hardware Accelerator - Testbench



Approach to Floating Point

Without a FPU one can’t use fractional numbers. More importantly, for the algorithm to converge, the scalar learning rate α has to be a small fractional number (e.g. 0.01). Otherwise the algorithm just won’t converge.

To solve this, one can represent the input fractional numbers as integers by multiplying them by a scalar factor. On the output one can then divide by the same scalar factor to get the fractional numbers back.

Special attention must be paid when performing multiplications between numbers in this representation. A division by the scaling factor must be made in every intermediate multiplication.

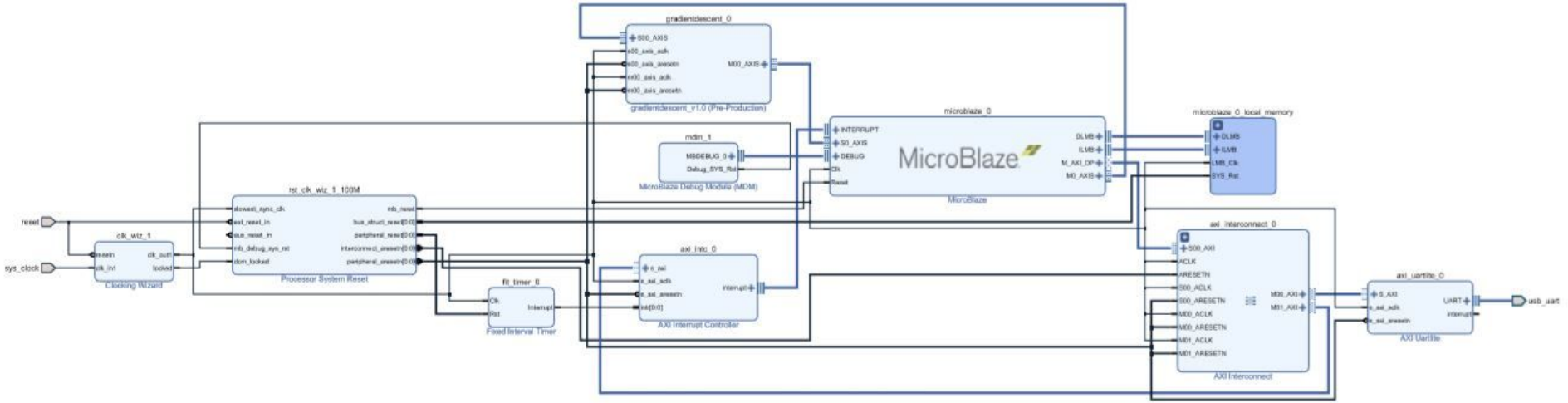
Approach to Floating Point

One could use e.g. 1000 as a scaling factor, but using a power of two like 2048 enables using right-shifting instead of integer division, which requires far more clock cycles.

FPGA Implementation

AXI Stream Interface

The MicroBlaze processor and our hardware accelerator are interfaced by AXI Stream.



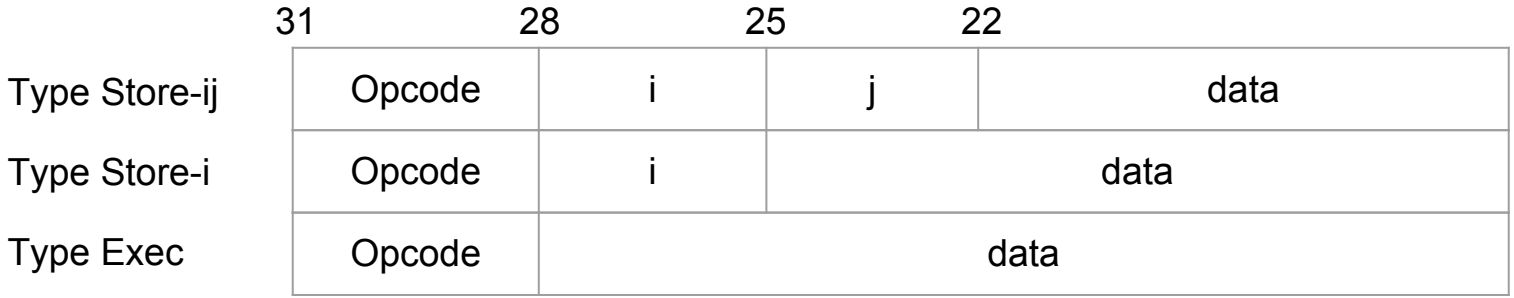
Coprocessor IP Core

Our coprocessor is a coprocessor in the true sense of the word because the MicroBlaze processor issues instructions for it to perform, which results in a fairly clean implementation of the coprocessor's slave.

The MicroBlaze processor's AXI Stream master interface sends instructions to the coprocessor which receives, decodes and processes them on its slave interface. The result (a vector of elements) is buffered to the MicroBlaze processor one element (a 32 bit word) at a time.

Coprocessor Instruction Set

There are 3 types of instructions that the coprocessor can decode. One instruction type can have more than one function associated.



Coprocessor Instruction Set

Function	Opcode	Type
Store element in matrix X	000	Store-ij
Store element in vector y	001	Store-i
Store element in vector θ	010	Store-i
Compute iteration of gradient descent	011	Exec
Reset	100	Exec
Store α (learning rate)	101	Exec

MicroBlaze Processor

Send Instructions

The MicroBlaze processor sends the coprocessor instructions to store each element of the matrix X, vector Y, learning rate α, and parameters θ. After this, in a loop, it instructs the coprocessor to compute iterations of gradient descent.

Note that it is only necessary to send the initial vector θ to the coprocessor. The subsequent θ vectors that are determined by the coprocessor are reutilized to calculate the next ones, without the need for software interference.

Nonetheless, the newly calculated vector θ is retrieved so that it can check if the algorithm has converged.

Convergence Check

The software checks if the algorithm has converged on every iteration.

Convergence can be approximately defined as follows: when the difference between each element of the newly calculated θ vector to each (respective) element on the previous θ vector is below a certain threshold, then the algorithm has converged, otherwise, it hasn’t.

Timing Performance

A fixed interval timer is configured to fire events every 100000-th clock cycle. The MicroBlaze processor is interrupted on each one of these events, upon which an interrupt handler is called to increment a global counter **irqCount**.

We can reset this counter when we want to start timing some task and simply look at the counter when we know said task has finished.

We can estimate the time elapsed **T** as a function of the interrupt counter **irqCount**, the clock's period (**period**) and the frequency of the fixed interval timer (**fit**).

$$\mathbf{T = period \times irqCount \times fit}$$

Demonstration

Conclusion

Hardware implementation yields 8x speedup