HEIG-VD | REDS Institut Reconfigurable and Embedded Digital Systems

Calcul Numérique et accélération Matérielle (CNM)

Prof Marina Zapater
Assistants : Mehdi Akeddar

Session 2 – Neural Network training with OpenMP

lab03 (21.11.2024)

Objectives of the laboratory

The objectives of Lab03 are to exercise the content regarding acceleration using multithreading. Lab03 is structured in two sessions. This manual refers to "Session 2" and is devoted to understanding OpenMP basics and optimizing the performance of a Neural Network by using OpenMP.

Laboratory validation and grading

- We will grade this laboratory based on the work we ask to be delivered.
  The parts in *light blue* are what you are expected to reply in the readme file.
- We expect (at least) three different commits to the git repository, one per session, clearly indicating the session submission (for example: git commit -m "submission session 1 lab02")
- This commit should be done at the latest the night before the next lab session.
- Therefore, you have a full week to submit the lab.

Stage 1 – Neural Networks

Artificial neural networks (**ANNs)**, also shortened to neural networks (NNs), are a branch of machine learning models that are built using principles of neuronal organization discovered by connectionism in the biological neural networks constituting animal brains.

During this lab, we will build a simple neural network structure and train it on dummy data. Then, we will optimize the different parts of the training using openMP.

A training of a NN is usually divided in 3 parts: weights initialization, forward pass and backpropagation. In what follows you will be required to accelerate these three main parts.

Stage 2 – Weights initialization

This is a procedure to set the weights of the neural network to small random values that define the starting point for the optimization. Different initialization techniques have shown positive effects on model trainings. For this lab, we will focus on a simple initialization, referred to as ***Xavier uniform***:

$$Weights = random \cdot \sqrt{\left(\frac{2}{\#\ Input\ units + \#\ output\ units}\right)}$$

Implement this initialization in the corresponding function. What would happen if all the weights were initialized to 0 instead?

Stage 3 – Forward pass

The forward pass in a neural network is the process of computing the output of the network for a given input. During the forward pass, **the input data is multiplied by the weights connecting the input layer to the hidden layer, and the result is passed through an activation function**. This intermediate result becomes the input for the next layer. The process is repeated for subsequent layers until the final output layer is reached. In each layer, the network computes a **weighted sum of the inputs** and applies a non-linear activation function, allowing the model to capture complex relationships in the data.

For this part, we will use a sigmoid activation function, where:

$$output_{layer\ activation} = \frac{1}{1+\exp(-output_{layer-1})}$$

Implement the forward pass in the corresponding function. Provide examples of other activation functions that we could use in this context.

Stage 4 – Backpropagation:

Backpropagation is the core algorithm for training neural networks. In the backward pass, the difference between the predicted output and the actual target, often referred to as the "loss" or "error", is calculated. Backpropagation then systematically adjusts the weights of the network by propagating this error backward through the layers. The gradient of the loss with respect to each weight is computed using the chain rule of calculus. These gradients guide the optimization algorithm (e.g., stochastic gradient descent) to update the weights in a way that minimizes the error. This iterative process is repeated for multiple epochs, refining the model's ability to make accurate predictions. Backpropagation enables neural networks to learn from data, discovering patterns and features that optimize their performance for specific tasks.

In our specific case, the backward pass can be achieved by following the steps:

- For each neuron in the output layer, calculate the gradient of the loss with respect to the output using the formula:

$$output_{gradient\ k} = (target_k - output_k) \cdot output_k \cdot (1 - output_k)$$

- For each weight connecting the hidden layer to the output layer, update the weight using the formula:

$$weight_{j\ k} \mathrel{+}= Learning\ rate \cdot output_{gradient\ k} \cdot neuron_j$$

- For each neuron in the hidden layer, compute the gradient of the loss with respect to the hidden layer using the formula:

$$hidden\ gradients_j = \sum^{k} output_{gradient\ k} \cdot weights_{j\ k} \cdot neuron_j \times (1 - neuron_j)$$

- For each weight connecting the input layer to the hidden layer, update the weight using the formula:

$$weights_{i\,j} += Learning\,rate \times hidden\,gradients_j \cdot input_i$$

Implement the backward pass in the corresponding function.

Stage 5 – Record time:

Record the time taken by the training for 1 epoch.

Stage 6 – OpenMP:

Use the OpenMP directives and functions learned in lab01 to improve the training performance. Indicate in the README where and you included OpenMP, what you wanted to parallelize and how. Report in README the time taken by the training using OpenMP. Do you see any improvement? Use different number of threads to evaluate the performance of your code.

Stage 7 – Task affinity:

Task affinity in OpenMP refers to the ability to bind a thread to a specific processing unit or core. This can potentially improve cache locality and overall performance. To set task affinity, you can export OMP_PROC_BIND and OMP_PLACES environment variables of OpenMP.

- **OMP_PLACES:** This environment variable specifies the available places for thread execution. The value can be a list of hardware places, such as "cores", "sockets", or "NUMA nodes". You can also specify a range of cores using the format "<lowerbound>:<length>:<stride>", where <lowerbound> is the starting core, <length> is the number of cores to use, and <stride> is the spacing between cores. For example, "0:4:2" specifies using four cores starting from the first core, with a stride of two.

- **OMP_PROC_BIND:** This environment variable determines the thread affinity policy, controlling how threads are bound to places. The possible values are:
  - *true*: Threads can be moved between places.
  - *false*: Threads are pinned to the place where they are created.
  - *master*: The master thread is pinned to the first place, and other threads can be moved between places.
  - *close*: Threads are pinned to the closest place that has available logical CPUs.
  - *spread*: Threads are evenly distributed across the available places.

Explore different combinations of number of threads, places and affinity policies and record the different improvements you observe. Provide an explanation for the results you observe. You can use the htop command to see the use of cores during the task.

Before you leave…

Make sure you have all the requests *in light blue* of this manual, that you have implemented all the functions required and filled-in the answers in the *README* file in the repo. Then commit all files to your own “cnm24-yourname” repository.

We expect (at least) one commit for this session, clearly indicating the session submission (for example: git commit -m “submission session 2 lab03”)

This commit should be done at the latest the day before the next lab session at 23:59h.