GPU-Accelerated Convolutional Kernel Training for vertical edge detection

Massimiliano Giordano Orsini

January 12, 2024

Supervisor: Livia Marcellino
Tutor: Pasquale De Luca

Abstract

Convolutional layers are a key component of several deep learning models, such as Convolutional Neural Networks (CNNs), which have been particularly successful in image and video processing tasks, achieving state-of-the-art results. In this report, we focus on a simplified task, such as vertical edge detection, for training a convolutional kernel on a synthetic dataset of labeled images, in which the convolution is performed on a GPU architecture, according to a proper parallelization strategy.

1 Problem definition and analysis

Edge detection is the process of identifying the boundaries between objects and the background in an image. It is a basic but fundamental task in image processing and computer vision. Finding only vertical edges is a semplification of that task, that we made, for educational purposes. One common approach to edge detection is to use a convolutional kernel, which can be learned through a supervised training procedure.

1.1 Convolution

For semplicity, we restrict our attention to gray-scale images and introduce the idea of convolution to two-dimensional images as follow [1]. For an image $\mathbf{I}$ with pixel intensities $I(j,k)$, and a filter $\mathbf{K}$ with pixel values $K(l,m)$, the feature map $\mathbf{C}$ has activation values given by

$$C(j,k) = \sum_l \sum_m I(j+l, k+m) K(l,m) \tag{1}$$

where we have omitted the nonlinear activation function for clarity. This again is an example of a *convolution* and is sometimes expressed as $\mathbf{C} = \mathbf{I} * \mathbf{K}$ [1].
Note that strictly speaking (1) is called a *cross-correlation*, which differs slightly from the conventional mathematical definition of convolution, but here we follow

common practice in the machine learning literature and refer to (1) as a convolution [1].
The relationship (1) is illustrated in Figure 1 for a 3×3 image and a 2×2 filter.

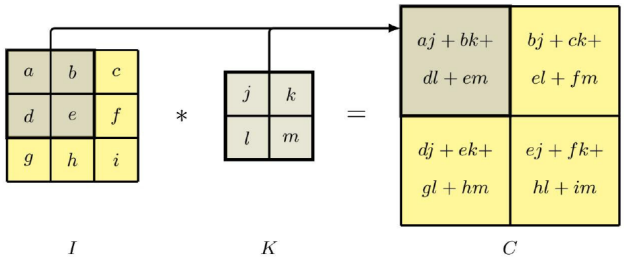


Figure 1: Example of a 3×3 image convolved with a 2×2 filter to give a resulting 2×2 feature map. [1]

We see from Figure 1 that the convolution map is smaller than the original image. If the image has dimensionality $H \times W$ pixels and we convolve with a kernel of dimensionality $K \times K$ (filters are typically chosen to be square) the resulting feature map has dimensionality $(H-K+1) \times (W-K+1)$. In some cases we want the feature map to have the same dimensions as the original image. This can be achieved by padding the original image with additional pixels around the outside, as illustrated in Figure 2. If we pad with P pixels then the output map has dimensionality $(H + 2P - K + 1) \times (W + 2P - K + 1)$ [1].

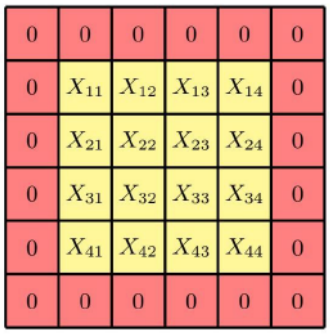


Figure 2: Illustration of a 4×4 image that has been padded with additional pixels to create a 6×6 image. [1]

When the value of P is chosen such that the output array has the same size as the input, corresponding to $P = (K-1)/2$, this is called a *same* convolution, because the image and the feature map have the same dimensions. In computer vision, filters generally use odd values of M, so that the padding can be symmetric on all sides of the image and that there is a well-defined central pixel associated with the location

of the filter. Finally, we have to choose a suitable value for the intensities associated with the padding pixels. A typical choice is to set the padding values to zero [1].

1.1.1 Sequential algorithm

As shown in Figure 1, the convolution can be easily thought as a dot product between a patch of the input image and the convolutional kernel, where they have the same sizes. The pseudocode for the dot product between two arrays is reported in Algorithm 1, where both patch and kernel are represented as monodimensional arrays, instead of 2D matrices.

Algorithm 1 Pseudocode for performing a dot product between two arrays

Require: $patch, kernel, n$
  $value \leftarrow 0$
  for $i \leftarrow 0$ to $n - 1$ **do**
    $value \leftarrow value + patch[i] \times kernel[i]$
  end for
  $result \leftarrow value$

In terms of time complexity, the dot product requires n multiplications and $n - 1$ additions.
Algorithm 2 provides the pseudocode for the convolution between an input image and a convolutional kernel. In this work, we used to consider same convolutions for the forward pass (which will be introduced later), assuming that the input image has been already padded properly, in order to get a feature map as output of the convolution, which has the same shape of the original input image.

Algorithm 2 Pseudocode for the convolution between an input image and a convolutional kernel

Require: $input, inputHeight, inputWidth$
Require: $kernel, kernelHeight, kernelWidth$
Require: $output, outputHeight, outputWidth$
Require: $bias$
  $kernelSize \leftarrow kernelHeight \times kernelWidth$
  for $i \leftarrow 0$ to $outputHeight - 1$ **do**
    for $j \leftarrow 0$ to $outputWidth - 1$ **do**
      $patch \leftarrow$ allocate $kernelSize$-dimensional array
      for $m \leftarrow 0$ to $kernelHeight - 1$ **do**
        for $n \leftarrow 0$ to $kernelWidth - 1$ **do**
          $patch[m][n] \leftarrow input[(i + m)][(j + n)]$
        end for
      end for
      $output[i][j] \leftarrow$ DotProduct($patch, kernel, kernelSize$)+$bias$
    end for
  end for

The *bias* term is typically added in the context of neural networks. However, in this case, it's a non-trainable parameter, which is fixed in advance.
In terms of time complexity, considering only the convolution operation performed as particular dot products between patches of the original image and the kernel, in this context, it takes a number of dot products equals to the number of pixels of the output image. Considering a same convolution, the number of pixels of the output image is equal to the number of pixels of the input image; so, denoting with H and W the number of rows and columns, respectively, of the input image, the convolution operation requires HW dot products, that is HWn multiplications and $HW(n-1)+HW$ additions, where n is the number of pixel of the kernel. Note that the last term is due to the bias additions.

1.2 Convolutional layer

A typical layer of a convolutional layer consists of three stages (see Figure 3). In the first stage, the layer performs several convolutions in parallel to produce a set of linear activations. In the second stage, each linear activation is run through a nonlinear activation function, such as the rectified linear activation function or a sigmoid activation function. This stage is sometimes called the **detector stage**. In the third stage, we use a pooling function to modify the output of the layer further [2].
In this work, we do not consider the so-called *pooling stage*.

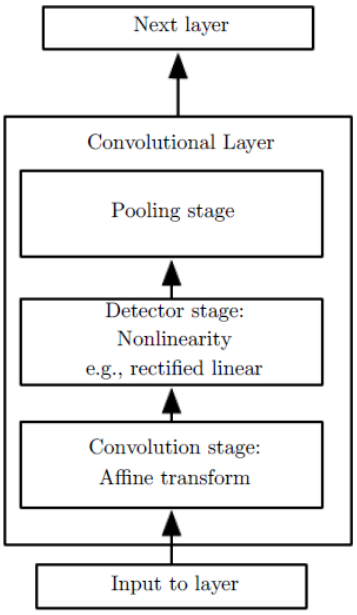


Figure 3: The components of a typical convolutional neural network layer. [2]

1.3 Training

Suppose to have a synthetic dataset composed by a sufficient number of images, where pixels of each image are labeled as edge pixels or not, considering a particular encoding strategy (e.g. 1 for edge pixels, 0 otherwise). This dataset is splitted into two subset: the **training set**, which will be used for training and the **test set**, which will be used for testing how well the convolutional kernel has been trained.

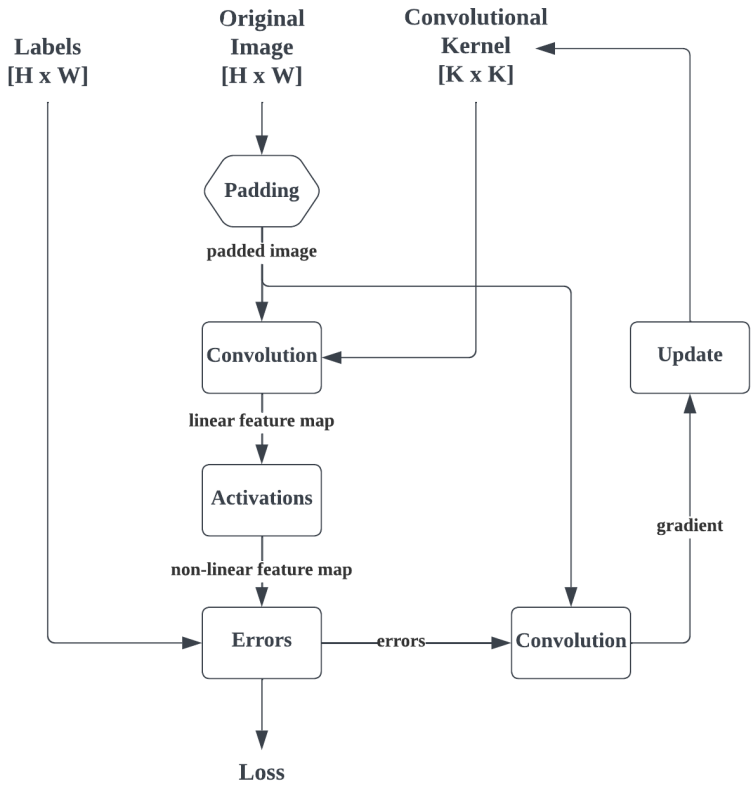


Figure 4: Visual representation of the training loop performed for an image at a given epoch.

Algorithm 3 essentially implements the forward and backward passes of a single-layer CNN for pixel classification (edge pixel or not). The forward pass involves the convolution and activation operations, while the backward pass involves the error and gradient computations, and the kernel update.
Mainly, it involves these macro-operations:

- Initialization: The function takes as input a training dataset, a kernel (or filter) with its dimensions, the number of training epochs, the learning rate, and an epoch step for logging purposes. It also calculates the padding for the convolution operation on the basis of the kernel size.

- Training Loop: The function then enters a loop that runs for the specified number of epochs. For each epoch, it initializes a variable trainLoss to keep track of the training loss.
- Sample loop: Within each epoch, the function loops over all images in the training set. For each image, it performs the following steps:
  - Data Retrieval: Retrieves the image data and its corresponding label.
  - Convolution: Pads the image data and performs a convolution operation between the padded image and the kernel. The result is stored in output.
  - Activation: Applies an activation function (in this case, sigmoid) to the output of the convolution operation.
  - Error calculation: Computes the error between the output of the activation function and the label.
  - Loss Calculation: Computes the loss from errors and adds it to trainLoss.
  - Gradient Calculation: Performs a convolution operation between the padded image and the errors to compute the gradient.
  - Kernel Update: Updates the kernel's weights using the computed gradient and the learning rate, according to an update rule.
- Memory Deallocation: Deallocates the dynamically allocated memory for output, errors, gradient, and paddedData.

Algorithm 3 Pseudocode of the training loop

Require: $trainingSet, epochs, learningRate, epochStep$
Require: $kernel, kernelHeight, kernelWidth$
  Compute padding $padh$, $padw$
  for $epoch \leftarrow 0$ to $epochs$ **do**
    Initialize $trainLoss \leftarrow 0$
    for $i \leftarrow 0$ to $nImages$ of the $trainingSet$ **do**
      Get data, label, width, height from i-th sample of the $trainingSet$
      Pad $data$ to get $paddedData$ using $padh$ and $padw$
      Perform convolution between $paddedData$ and $kernel$ to get $output$
      Apply activation function (e.g sigmoid) to $output$
      Compute errors between $output$ and $label$ to get $errors$
      Compute loss from $errors$ to get $loss$
      Accumulate $loss$ to $trainLoss$
      Perform convolution between $paddedData$ and $errors$ to get $gradient$
      Update $kernel$ using $gradient$ and $learningRate$
    end for
  end for

The *loss function* is involved in the context of backpropagation and optimization algorithms, such as Gradient Descent, to train the model. The process involves adjusting the weights of the filters to minimize the errors in predictions. In this

context, it is used during the training loop, for measuring how well the learning is going.
The gradient points directly uphill, and the negative gradient points directly downhill. We can decrease the loss function L by moving in the direction of the negative gradient. This is known as the method of steepest descent or gradient descent. [2]
Steepest descent proposes a new point

$$w' = w - \epsilon \Delta_w L(w) \tag{2}$$

where ϵ is the *learning rate*, a positive scalar determining the size of the step. We can choose ϵ in several different ways. A popular approach is to set ϵ to a small constant. [2]
Once the training procedure is performed, the convolutional kernel is involved in a testing procedure, for assessing the performance (e.g. accuracy) on the samples of the test set.
Algorithm 4 is used to make predictions on a test dataset using a trained convolutional kernel. In simple words, it implements the forward pass of a single-layer CNN for pixel classification. The forward pass involves the convolution and activation operations. Typically, the accuracy of the predictions is then computed and logged. In constrast to the training loop, the step function is used as activation function and, obviously, the backward pass is avoided.

Algorithm 4 Pseudocode of the Predict function used for testing a trained convolutional kernel

Require: *testSet*
Require: *kernel*, *kernelHeight*, *kernelWidth*
  Initialize accuracy variables
  Compute padding *padh*, *padw*
  for $i = 0$ to *nImages* of the *testSet* **do**
    Get *data* and *label* from *i*-th sample of the *testSet*
    Pad *data* to get *paddedData* using *padh* and *padw*
    Perform convolution between *paddedData* and *kernel* to get *preds*
    Apply activation function (e.g. step function) to *preds*
    Compute accuracy between *preds* and *label*
    Update total accuracy and correct predictions
  end for
  Compute an average *testAccuracy*

1.3.1 Backpropagation

$x^l_{i,j}$ is the convolved input vector at layer l plus the bias b, represented as

$$x^l_{i,j} = \sum_m \sum_n w^l_{m,n} o^l_{i+m,j+n} + b^l \tag{3}$$

where $o^l_{i,j}$ is the output vector at layer l given by

$$o^l_{i,j} = f(x^l_{i,j}) \tag{4}$$

$f(x_{i,j}^l)$ is the application of the activation layer to the convolved input vector at layer l.
For a total of P predictions, the predicted network outputs y_p and their corresponding targeted values t_p the mean squared error is given by

$$E = \frac{1}{2}\sum_p (t_p - y_p)^2 \tag{5}$$

Learning will be achieved by adjusting the weights such that the prediction is as close as possible or equals to corresponding label associated to each pixel of the image [3].
In the classical backpropagation algorithm, the weights are changed according to the gradient descent direction of an error surface E [3].
We are looking to compute the gradient of the error surface E with respect to the weights of the convolutional kernel $w_{m,n}$, which can be interpreted as the measurement of how the change in a single pixel $w_{m,n}$ of the kernel affects the loss function E [3].
During forward propagation, the convolution operation ensures that the pixel $w_{m,n}$ in the weight kernel makes a contribution in all the products and hence affects all the elements in the output feature map [3].
Convolution between the input feature map of dimension $H \times W$ and the weight kernel of dimension $k_1 \times k_2$ produces an output feature map of size $(H-k_1+1)by(W-k_2+1)$ [3].
Gradient computation for individual weights can be obtained by applying the chain rule [3].

$$\begin{aligned}\frac{\partial E}{\partial w_{m,n}^l} &= \sum_{i=0}^{H-k_1}\sum_{j=0}^{W-k_2} \frac{\partial E}{\partial x_{i,j}^l}\frac{\partial x_{i,j}^l}{\partial w_{m,n}^l} \\ &= \sum_{i=0}^{H-k_1}\sum_{j=0}^{W-k_2} \delta_{i,j}^l \frac{\partial x_{i,j}^l}{\partial w_{m,n}^l}\end{aligned} \tag{6}$$

where $\delta_{i,j}^l$is the error committed by the prediction with respect to the correct label.

$$\begin{aligned}\frac{\partial x_{i,j}^l}{\partial w_{m,n}^l} &= \frac{\partial}{\partial w_{m,n}^l}\left(\sum_m\sum_n w_{m,n}^l o_{i+m,j+n}^{l-1} + b^l\right) \\ &= \frac{\partial}{\partial w_{m,n}^l}(w_{0,0}^l o_{i+0,j+0}^{l-1} + \cdots + w_{m',n'}^l o_{i+m',j+n'}^{l-1} + \cdots + b^l) \\ &= \frac{\partial}{\partial w_{m,n}^l}(w_{m',n'}^l o_{i+m',j+n'}^{l-1}) \\ &= o_{i+m',j+n'}^{l-1}\end{aligned} \tag{7}$$

We substitute the term obtained in Equation 7, and we get that

$$\begin{aligned}\frac{\partial E}{\partial w^l_{m,n}} &= \sum_{i=0}^{H-k_1}\sum_{j=0}^{W-k_2} \delta^l_{i,j} o^{l-1}_{i+m',j+n'} \\ &= \text{rot}_{180^\circ}\{\delta^l_{i,j}\} * o^{l-1}_{m',n'}\end{aligned} \tag{8}$$

This demonstrates that all convolutional layers can be trained by means of backward convolutions.

2 Input and Output

The convolutional kernel can be initialized in several ways. For example, weights of the kernel can be drawn from some classical distribution (normal, uniform, etc.). In this work, we arbitrarily consider a random initilization by sampling weigths from a uniform distribution from 0 to 100.

-1	0	1
-1	0	1
-1	0	1

Figure 5: Hand-crafted convolutional filter for detecting vertical edges. (Source: [1])

After training, we expect that the filter has learned how to detect the vertical edge, and so to give strong responses in the corrispondence of a vertical edge. Figure 5 gives an insight of how an hand-crafted vertical edge filter should look like, according to the definition of derivatives. However, as we will see, the outputted convolutional kernel is not properly equal to the kernel in Figure 5 but roughly similar.

3 Algorithm description

Due to the fact that the convolution was the heaviest computational kernel in the training procedure, we adopted a suitable parallelization strategy for computing it. This strategy considers a 1D grid of blocks, where the number of blocks is equal to the height of the output matrix. Each block contains a number of threads equal to the width of the output matrix. This means that each thread in the grid is responsible for computing one element of the output matrix, independently, according to the coalescence rules. Note that, at the forward pass, the size of the output matrix is equal to the size of the input image, instead at the backward pass, it is equal to the size of the kernel.

Generally speaking, in the forward pass, convolution requires the computation of a large number of "small" dot products, due to the fact that the kernel size will be much smaller that the image size, that is typically large; in the backward pass, convolution requires the computation of a small number of "large" dot products, due to the fact that the kernel size will be equal to the size of the original image, resulting in a more computationally-intensive operation.

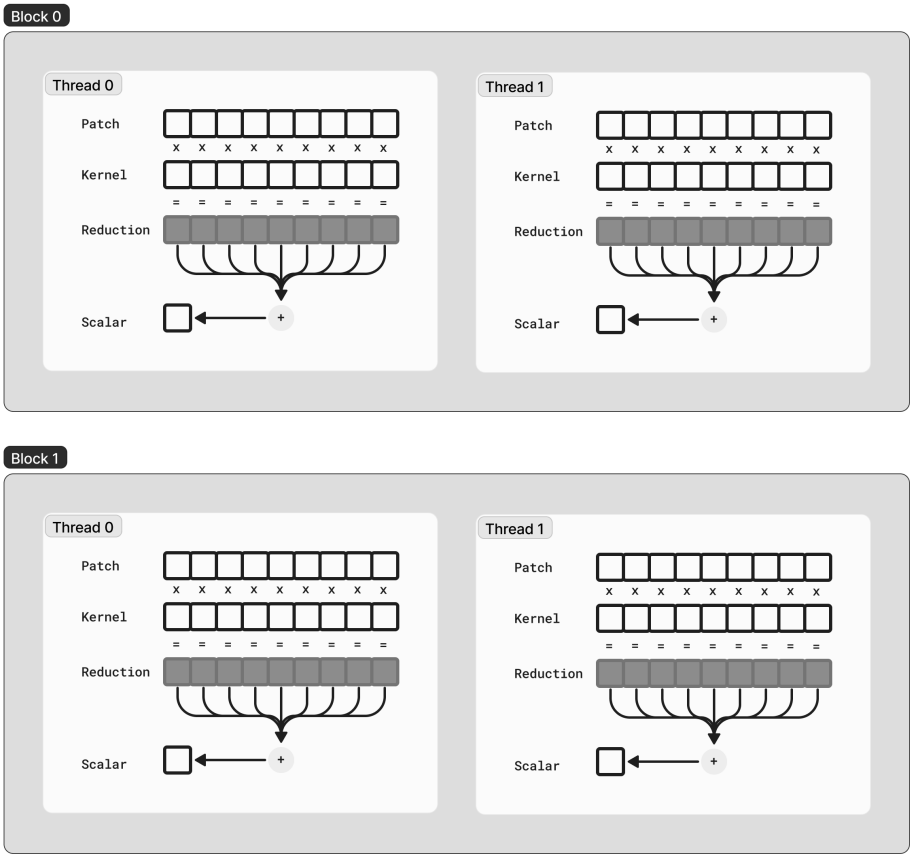


Figure 6: Visual representation of the parallelization strategy for a convolution between a 4×4 image and a 3×3 kernel. The output is given by a 2×2 image, where a particular dot product is performed by each thread, independently.

The use of shared memory seems to be useful for storing the kernel, which is shared for each convolution. However, this is not involved because, at the backward pass, the kernel corresponds to the errors matrix, which has the same size of the input image. This means that, for large input images, storing the kernel into the shared memory is not affordable, due to its limited size.

4 Implemented routines

All the code has been developed from scratch. In this section, we focus on the fundamental routines, which are actually parallelized using CUDA, a parallel computing platform and application programming interface model created by Nvidia.

4.1 cudaConvolve_

The *cudaConvolve_* function is the parallel routine responsible for computing the dot product, according to the parallelization strategy discussed before. In this case, each thread in the kernel computes one element of the output matrix by taking the dot product of the kernel and a submatrix of the input matrix, and then the bias is added.

Algorithm 5 Function *cudaConvolve_* implemented in CUDA C

```
__global__ void cudaConvolve_(double *input, int inputWidth,
  double *kernel, int kernelHeight, int kernelWidth,
  double *output, int outputHeight, int outputWidth,
  double bias) {
  // pay attention that large kernels could not be fit into shared
  // memory (and consequently, large fraction of input)
  int index = blockDim.x * blockIdx.x + threadIdx.x;
  if (index < outputHeight * outputWidth) {
    // perform dot product
    double value = 0.f;
    for(int i = 0; i < kernelHeight; i++) {
      for(int j = 0; j < kernelWidth; j++) {
        value += (input[(blockIdx.x + i) * inputWidth + (threadIdx.x
             + j)] * kernel[i * kernelWidth + j]);
      }
    }
    output[index] = value + bias;
  }
}
```

4.2 cudaConvolve

The *cudaConvolve* function performs a convolution operation on an input matrix using a kernel.

a. The function starts by setting up the grid and block dimensions. It creates a 1D grid of blocks, where the number of blocks is equal to the height of the output matrix. Each block contains a number of threads equal to the width of the output matrix.

b. Next, the function allocates memory on the GPU for the input matrix, the kernel, and the output matrix. It then transfers the input matrix and the kernel from the host (CPU) memory to the device (GPU) memory. It also initializes the output matrix on the device to zero.

c. Once the data are set on the device, the function launches the *cudaConvolve_* kernel for computing elements of the output matrix.

d. After the kernel execution, the function checks for any errors during kernel launch and synchronizes the device to ensure that all threads have finished their computation. It then transfers the output matrix from the device memory back to the host memory.

e. Finally, the function deallocates the memory that was allocated on the GPU. This cleanup step is important to prevent memory leaks and to free up resources for other operations.

gpuErrchk is a macro defined for checking any possible error produced by CUDA routines, ensuring the consistency of their output.

5 Performance analysis

In this section, we will discuss about the performance of the sequential algorithm for performing the convolution and so, the training of the convolutional kernel, which runs on a specific Central Processing Unit (CPU), in comparison to the parallel algorithm proposed, which runs on a specific Graphic Processing Unit (GPU) device and architecture.
All the experiments were performed on PurpleJeans, a high performance computing cluster dedicated to the production of "research products" forms the fourth generation core of RCF's advanced computational infrastructure[1] [4].
Table 1 reports the technical specifications of purpleJeans.

Resource	Partitions	Knots	CPU/Node	Core/Node	Memory/Node	Notes
purpleJeans (2019)	xhicpu	4	2 CPU Intel(R) Xeon(R) Xeon 16-Core 5218 2,3Ghz 22MB	32	192 GB	Cluster dedicated to research in the field of machine learning and big data. Mellanox CX4 VPI SinglePort FDR IB 56Gb/s x16.
	xgpu	4	2 CPU Intel(R) Xeon(R) Xeon 16-Core 5218 2,3Ghz 22MB	32	192 GB	4 GPU (NVLINK) NVIDIA Tesla V100 32GB SXM2. Infiniband Mellanox CX4 VPI SinglePort FDR IB 56Gb/s. x16.

Table 1: Technical specifications of purpleJeans. (Source: [5])

Table 2 reports the device specification of the actual GPU used for the experiment, which is a Tesla V100 GPU.

[1] Resource Computing Facilities (RFC) is a management structure for High Performance Computing, GPU Computing and Cloud Computing computational resources of the Department of Science and Technology of the University of Naples "Parthenope". [4]

Algorithm 6 Function *cudaConvolve* implemented in CUDA C

```
1 void cudaConvolve(double *input, int inputHeight, int inputWidth,
2   double *kernel, int kernelHeight, int kernelWidth,
3   double *output,  int outputHeight, int outputWidth,
4   double bias) {
5
6   // GPU kernel configuration
7   dim3 nBlocks(outputHeight), nThreadsPerBlock(outputWidth);
8   double *deviceInput, *deviceKernel, *deviceOutput;
9
10  // device vars allocation
11  gpuErrchk(cudaMalloc((void **) &deviceInput, inputHeight *
        inputWidth * sizeof(double)));
12  gpuErrchk(cudaMalloc((void **) &deviceKernel, kernelHeight *
        kernelWidth * sizeof(double)));
13  gpuErrchk(cudaMalloc((void **) &deviceOutput, outputHeight *
        outputWidth * sizeof(double)));
14
15  // data transfer from ram (host) to global memory (device)
16  gpuErrchk(cudaMemcpy(deviceInput, input, inputHeight * inputWidth
        * sizeof(double), cudaMemcpyHostToDevice));
17  gpuErrchk(cudaMemcpy(deviceKernel, kernel, kernelHeight *
        kernelWidth * sizeof(double), cudaMemcpyHostToDevice));
18  gpuErrchk(cudaMemset(deviceOutput, 0, outputHeight * outputWidth *
         sizeof(double)));
19
20  // GPU kernel call
21  cudaConvolve_ <<< nBlocks, nThreadsPerBlock >>> (deviceInput,
        inputWidth, deviceKernel, kernelHeight, kernelWidth,
        deviceOutput, outputHeight, outputWidth, bias);
22  gpuErrchk(cudaGetLastError());
23  gpuErrchk(cudaDeviceSynchronize());
24
25  // data transfer from global memory (device) to ram (host)
26  gpuErrchk(cudaMemcpy(output, deviceOutput, outputHeight *
        outputWidth * sizeof(double), cudaMemcpyDeviceToHost));
27
28  // memory deallocation on device
29  gpuErrchk(cudaFree(deviceInput));
30  gpuErrchk(cudaFree(deviceOutput));
31  gpuErrchk(cudaFree(deviceKernel));
32 }
```

Device Property	Value
name	Tesla V100-SXM2-32GB
major/minor	7.0 compute capability
totalGlobalMem	4 024 958 976 bytes
sharedMemPerBlock	49 152 bytes
regsPerBlock	65 536
warpSize	32
maxThreadsPerBlock	1 024
maxThreadsDim[0]	1 024
maxThreadsDim[1]	1 024
maxThreadsDim[2]	64
maxGridSize[0]	2 147 483 647
maxGridSize[1]	65 535
maxGridSize[2]	65 535
clockRate	1 530 000 (kHz)
deviceOverlap	1
multiProcessorCount	80

Table 2: Technical specification of the GPU device used.

For analyzing the performance of our parallelization strategy and the corresponding implementation, we performed several experiments, varying the image and kernel size, from 8×8 to 256×256 and from 3×3 to 32×32, respectively. According to our strategy and the capabilities of the GPU device considered, this limit was given by the maximum number of threads per block, which was 1024. This means that our strategy works up to 1024×1024 images.
All the experiments were conducted under the same initial conditions, which are:

a. The initial dataset was composed by 500 squared images, each associated to the correct label image. The training/test split was 75/25 (a common choice in machine learning). The learning rate was fixed at 0.01. The number of epochs was set to 10.

b. The convolutional kernel was initialized in the same way before being passed to the sequential algorithm and the parallel version. The considered filters were squared and generally odd.

Almost all the experiments reported a loss value between 0 and 1 in the last epoch of the training phase and a 100% accuracy score at the testing phase, except for some of them, which in anyway reached a 99.8% accuracy score. However, these results were omitted from our analysis because we are focused on the capabilities of the proposed parallelization strategy, in terms of execution time, number of blocks/threads

used, and ratio of improvement (with respect to the sequential algorithm).
Table 3 reports the experimental results of the sequential algorithm. As it's possible to observe, execution times seem to be quite reasonable for little computational kernel (fast training and fast inference).

Image Size (scalar)	**Kernel Size** (scalar)	**Maximum execution time for forward convolution** (seconds)	**Maximum execution time for backward convolution** (seconds)	**Total execution time for training [cpu]** (seconds)	**Maximum execution time for forward convolution [predict]** (seconds)	**Total execution time for test [cpu]** (seconds)
8	3	9.0e-6	7.0e-6	0.103	6.0e-6	0.002
	5	3.4e-5	2.9e-5	0.24	1.1e-5	0.003
	11					
	23					
	32					
16	3	4.8e-5	3.3e-5	0.369	2.2e-5	0.004
	5	9.7e-5	7.8e-5	0.720	5.0e-5	0.008
	11	2.7e-4	2.7e-4	2.374	1.61e-4	0.022
	23					
	32					
32	3	1.2e-4	8.5e-5	0.943	1.0e-4	0.015
	5	4.5e-4	3.6e-4	2.122	1.7e-4	0.025
	11	6.2e-4	5.9e-4	6.387	6.58e-4	0.082
	23	2.6e-3	2.7e-3	26.695	2.9e-3	0.347
	32					
64	3	4.3e-4	2.9e-4	3.227	3.9e-4	0.057
	5	9.9e-4	8.0e-4	6.471	1.2e-3	0.159
	11	2.4e-3	2.3e-3	24.739	2.6e-3	0.328
	23	1.0e-2	1.0e-2	103.821	3.6e-2	1.746
	32	2.0e-2	2.0e-2	196.952	4.3e-2	3.261
128	3	1.0e-3	7.0e-4	11.285	1.4e-3	0.220
	5	2.4e-3	1.9e-3	24.171	2.8e-3	0.413
	11	9.7e-3	9.0e-3	95.884	1.0e-2	1.311
	23	4.3e-2	3.9e-2	409.802	4.3e-2	5.230
	32	8.1e-2	7.6e-2	783.628	8.0e-2	10.013
256	3	4.1e-3	2.8e-3	44.884	5.6e-3	0.845
	5	9.7e-3	7.5e-3	96.250	1.1e-2	1.576
	11	3.9e-2	3.6e-2	387.106	4.0e-2	5.198
	23	1.7e-2	1.6e-2	1634.144	1.7e-1	21.333
	32	3.3e-1	3.0e-1	3127.344	3.2e-1	39.972

Table 3: Experimental results of the sequential algorithm, run a CPU device.

Performances start to degradate quite faster considering larger and larger kernels and images, as we expected.
Table 4 shows the experimental results of the parallelization strategy (and the corresponding implementation) on the GPU device, introduced before. For little compu-

tational kernels, performances are even worse than the sequential version (mainly, due to the communication overhead) but quite consistent, also considering huge image and kernel sizes. Execution times measured for forward convolutions seem to be almost constant, in contrast to what we've seen before.

Image Size (scalar)	**Kernel Size** (scalar)	**Maximum execution time for forward convolution** (seconds)	**Maximum execution time for backward convolution** (seconds)	**Total execution time for training [gpu]** (seconds)	**Maximum execution time for forward convolution [predict]** (seconds)	**Total execution time for test [gpu]** (seconds)
8	3	4.9e-4	4.9e-4	3.918	5.4e-4	0.047
	5	5.3e-4	5.3e-4	4.149	5.0e-4	0.045
	11					
	23					
	32					
16	3	5.5e-4	4.7e-4	4.275	6.1e-4	0.042
	5	5.8e-4	8.1e-4	4.517	4.7e-4	0.042
	11	5.9e-4	5.2e-4	4.015	5.6e-4	0.044
	23					
	32					
32	3	7.5e-4	7.3e-4	4.805	6.3e-4	0.048
	5	5.5e-4	5.8e-4	4.904	5.0e-4	0.048
	11	4.2e-4	7.6e-4	4.028	4.8e-4	0.042
	23	3.9e-4	4.8e-4	4.512	5.3e-4	0.047
	32					
64	3	4.9e-4	7.0e-4	6.062	4.8e-4	0.047
	5	3.7e-4	7.5e-4	5.477	5.5e-4	0.060
	11	3.9e-4	7.1e-4	5.807	4.9e-4	0.048
	23	5.1e-4	7.4e-4	6.287	6.0e-4	0.061
	32	4.6e-4	6.6e-4	5.712	4.9e-4	0.053
128	3	4.1e-4	1.5e-3	11.985	4.1e-4	0.068
	5	4.7e-4	1.3e-3	10.567	4.4e-4	0.069
	11	4.0e-4	1.5e-3	12.268	5.2e-4	0.071
	23	4.1e-4	1.4e-3	12.103	4.6e-4	0.076
	32	5.4e-4	1.3e-3	11.183	5.5e-4	0.079
256	3	6.0e-4	4.8e-3	36.862	6.7e-4	0.156
	5	5.9e-4	3.9e-3	32.117	8.4e-4	0.200
	11	6.8e-4	4.9e-3	36.984	8.1e-4	0.156
	23	7.3e-4	5.0e-3	37.218	1.3e-3	0.165
	32	7.4e-4	3.9e-3	32.803	9.7e-4	0.191

Table 4: Experimental results of the parallel algorithm, run on a GPU device.

Please, note that measurements are strictly dependent from the initial dataset or the parameters of the training procedure (e.g. considering a larger dataset will correspond necessarily to a larger computational effort, and consequently, higher execution times. Same considerations for the number of epochs).

Table 5 highlights the number of threads involved in the computation of the forward and backward convolutions and, at the same time, it puts the focus on the improvement obtained by using the proposed parallel algorithm for a GPU device, with respect to the sequential version executed on a CPU device, for training a convolutional kernel.

Image Size (scalar)	**Kernel Size** (scalar)	**Total no. of threads for forward convolution** (scalar)	**Total no. of threads for backward convolution** (scalar)	**Ratio of improvement for training [cpu/gpu]** (scalar)	**Ratio of improvement for test [cpu/gpu]** (scalar)
8	3	64	9	0.03	0.04
	5	64	25	0.06	0.07
	11	64	121		
	23	64	529		
	32	64	1 024		
16	3	256	9	0.09	0.09
	5	256	25	0.16	0.19
	11	256	121	0.59	0.5
	23	256	529		
	32	256	1 024		
32	3	1 024	9	0.20	0.31
	5	1 024	25	0.43	0.52
	11	1 024	121	1.58	1.95
	23	1 024	529	5.92	7.4
	32	1 024	1 024		
64	3	4 096	9	0.53	1.21
	5	4 096	25	1.18	2.65
	11	4 096	121	4.26	6.83
	23	4 096	529	16.51	28.62
	32	4 096	1 024	34.48	61.53
128	3	16 384	9	0.94	3.23
	5	16 384	25	2.29	5.98
	11	16 384	121	7.81	18.46
	23	16 384	529	33.86	68.81
	32	16 384	1 024	70.07	126.75
256	3	65 536	9	1.22	5.42
	5	65 536	25	3.00	7.88
	11	65 536	121	10.47	33.32
	23	65 536	529	43.90	129.29
	32	65 536	1 024	95.34	209.28

Table 5: Comparison between the sequential and parallel versions for training and testing a convolutional kernel, executed on CPU and GPU devices, respectively.

The overall training and testing phases result to be up to **95 times** and **209 times faster**, respectively, in comparison to what we measured initially.

6 Usage examples

This section will provide compilation instructions and usage examples of the software developed in order to perform the training of a convolutional kernel for detecting vertical edges.

- **convGradientDescent.c**: this file contains the code of the sequential algorithm.
- **cudaConvGradientDescent.cu**: this file contains the code of the parallel algorithm.

Table 6 resumes the options, in common between all the files, for tweaking the hyperparameters of the training.

Parameter	Description
-h	Specifies the height of the images. The argument must be a positive integer (default: 12)
-w	Specifies the width of the images. The argument must be a positive integer (default: 12)
-k	Specifies the size of the kernel. The argument must be a positive integer and smaller than the image size. (default: 3)
-n	Specifies the number of images. The argument must be a positive integer (default: 500)
-t	Specifies the test size. The argument must be a floating-point number between 0 and 1 (default: 0.75)
-e	Specifies the number of epochs. The argument must be a positive integer (default: 100)
-l	Specifies the learning rate. The argument can be any floating-point number (default: 0.01)
-s	Specifies the epoch step for logging. The argument must be a positive integer (default: 10)
-i	Specifies whether the program runs in interactive mode. The argument must be either 0 or 1 (default: 0)
-H	Prints the usage of the program and exits

Table 6: Options table for executing the training of a convolutional kernel.

6.1 convGradientDescent.c

The following code snippet contains the instructions for compiling and executing the training of a convolutional kernel (sequential version).

```
$ gcc -std=c99 -lm convGradientDescent.c -o
 ↪  convGradientDescent
$ ./convGradientDescent [-n nImages] [-h height] [-w
 ↪  width] [-k kernelSize] [-e epochs] [-s epochStep]
 ↪  [-l lr] [-t testSize] [-i interactive]
```

Here we show an execution of the sequential training, with 10 epochs, 1 epoch step for printing information. We set the *interactive* mode to 1 in order to look at the test predictions given by the trained convolutional kernel.
The convolutional kernel is displayed before and after the training for looking at how it is changed. For each epoch, it's displayed the elapsed time taken to perform the fundamental steps of the training loop, and the overall elapsed time for training.

Other relevant insights are shown in Figure 7.

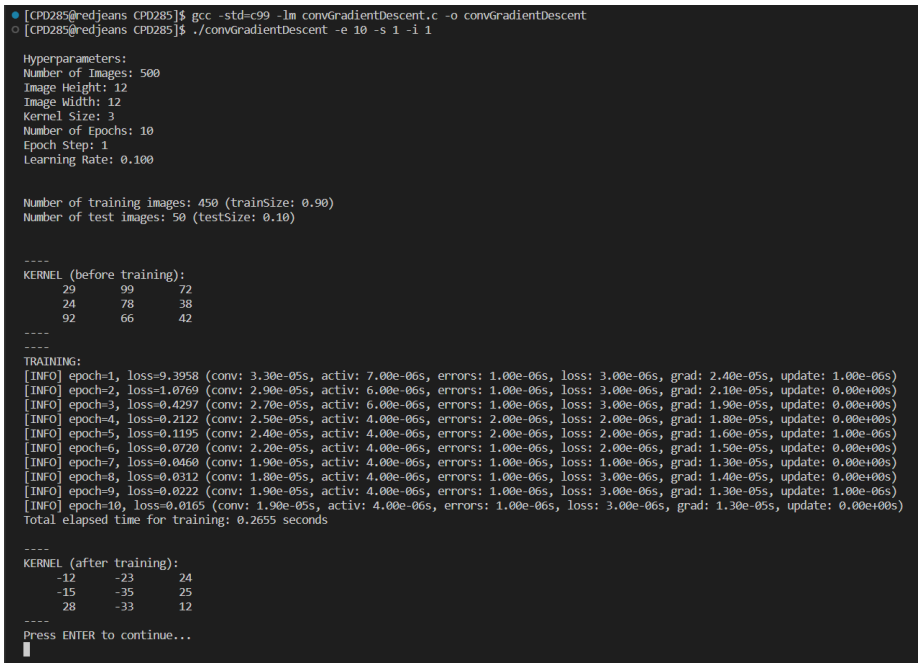
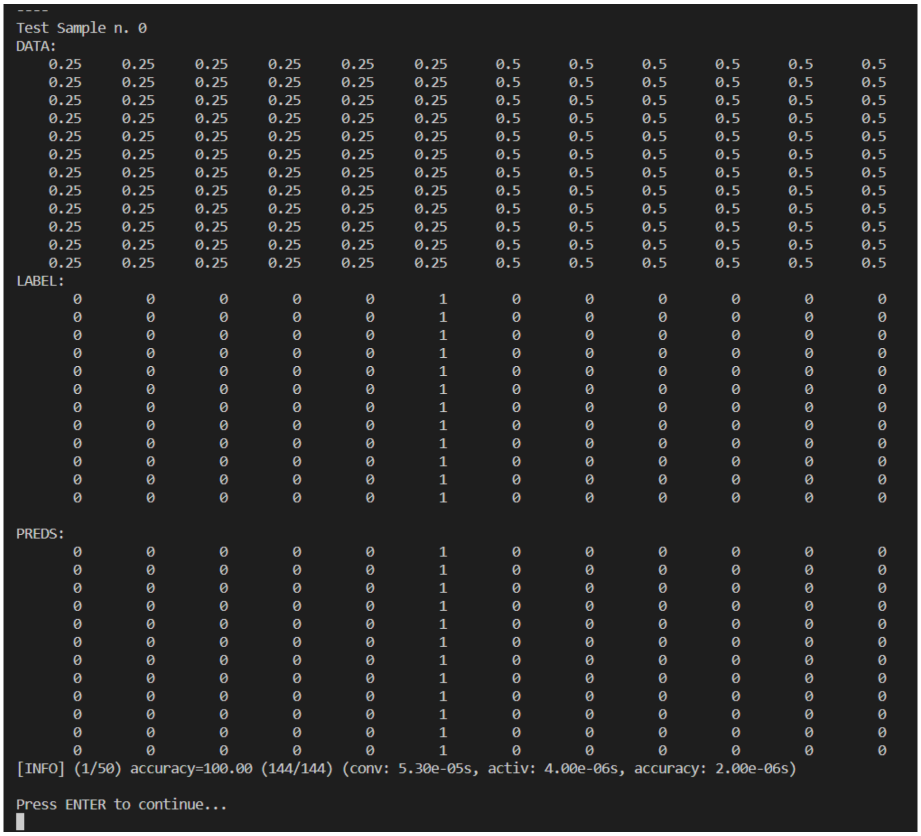


Figure 7: Usage example of *convGradientDescent.c* (part 1)

Thanks to the *interactive* option, we are able to see test images (and corresponding label images) and the predictions produced by the convolutional kernel, after training, as Figure 8 shows. Correctly, it outputs edge pixels in correspondence of the actual edge pixels.



```
----
Test Sample n. 0
DATA:
    0.25    0.25    0.25    0.25    0.25    0.25    0.5     0.5     0.5     0.5     0.5     0.5
    0.25    0.25    0.25    0.25    0.25    0.25    0.5     0.5     0.5     0.5     0.5     0.5
    0.25    0.25    0.25    0.25    0.25    0.25    0.5     0.5     0.5     0.5     0.5     0.5
    0.25    0.25    0.25    0.25    0.25    0.25    0.5     0.5     0.5     0.5     0.5     0.5
    0.25    0.25    0.25    0.25    0.25    0.25    0.5     0.5     0.5     0.5     0.5     0.5
    0.25    0.25    0.25    0.25    0.25    0.25    0.5     0.5     0.5     0.5     0.5     0.5
    0.25    0.25    0.25    0.25    0.25    0.25    0.5     0.5     0.5     0.5     0.5     0.5
    0.25    0.25    0.25    0.25    0.25    0.25    0.5     0.5     0.5     0.5     0.5     0.5
    0.25    0.25    0.25    0.25    0.25    0.25    0.5     0.5     0.5     0.5     0.5     0.5
    0.25    0.25    0.25    0.25    0.25    0.25    0.5     0.5     0.5     0.5     0.5     0.5
    0.25    0.25    0.25    0.25    0.25    0.25    0.5     0.5     0.5     0.5     0.5     0.5
    0.25    0.25    0.25    0.25    0.25    0.25    0.5     0.5     0.5     0.5     0.5     0.5
LABEL:
       0       0       0       0       0       1       0       0       0       0       0       0
       0       0       0       0       0       1       0       0       0       0       0       0
       0       0       0       0       0       1       0       0       0       0       0       0
       0       0       0       0       0       1       0       0       0       0       0       0
       0       0       0       0       0       1       0       0       0       0       0       0
       0       0       0       0       0       1       0       0       0       0       0       0
       0       0       0       0       0       1       0       0       0       0       0       0
       0       0       0       0       0       1       0       0       0       0       0       0
       0       0       0       0       0       1       0       0       0       0       0       0
       0       0       0       0       0       1       0       0       0       0       0       0
       0       0       0       0       0       1       0       0       0       0       0       0
       0       0       0       0       0       1       0       0       0       0       0       0

PREDS:
       0       0       0       0       0       1       0       0       0       0       0       0
       0       0       0       0       0       1       0       0       0       0       0       0
       0       0       0       0       0       1       0       0       0       0       0       0
       0       0       0       0       0       1       0       0       0       0       0       0
       0       0       0       0       0       1       0       0       0       0       0       0
       0       0       0       0       0       1       0       0       0       0       0       0
       0       0       0       0       0       1       0       0       0       0       0       0
       0       0       0       0       0       1       0       0       0       0       0       0
       0       0       0       0       0       1       0       0       0       0       0       0
       0       0       0       0       0       1       0       0       0       0       0       0
       0       0       0       0       0       1       0       0       0       0       0       0
       0       0       0       0       0       1       0       0       0       0       0       0
[INFO] (1/50) accuracy=100.00 (144/144) (conv: 5.30e-05s, activ: 4.00e-06s, accuracy: 2.00e-06s)

Press ENTER to continue...
```

Figure 8: Usage example of *convGradientDescent.c* (part 2)

6.2 cudaConvGradientDescent.cu

The following code snippet contains the instructions for compiling and executing the training of a convolutional kernel (parallel version).

```
$ nvcc cudaConvGradientDescent.cu -o
 ↪  cudaConvGradientDescent
$ ./cudaConvGradientDescent [-n nImages] [-h height]
 ↪  [-w width] [-k kernelSize] [-e epochs] [-s
 ↪  epochStep] [-l lr] [-t testSize] [-i interactive]
```

Similarly to what we've done before, Figure 9 shows the execution of a training using the "parallel" convolution. In particular, we set the number of epochs, the epoch step equal to 10 and 1, respectively. Moreover, in this particular case, we consider images of size 16×16 and a kernel of size 5×5. We set the *interactive* mode to 1 in order to look at the test predictions given by the trained convolutional kernel.
The convolutional kernel is displayed before and afterthe training for looking at how

it is changed. For each epoch, it is displayed the elapsed time taken to perform the fundamental steps of the training loop, and the overall elapsed time for training. Other relevant insights are shown in Figure 9.

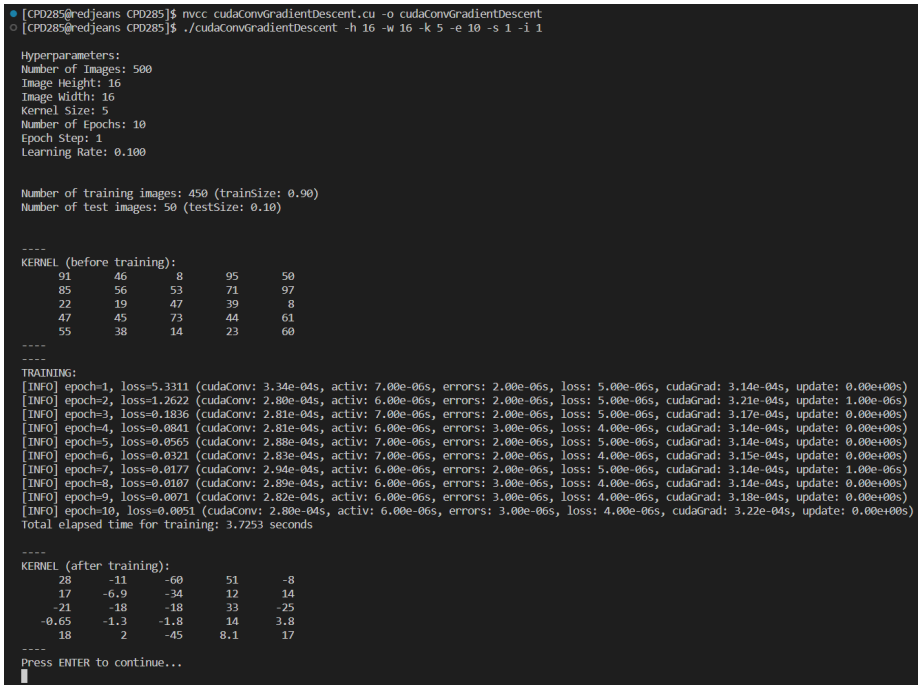


Figure 9: Usage example of *cudaConvGradientDescent.cu* (part 1)

As in sequential training, we can see test images (and corresponding label images) and the predictions produced by the convolutional kernel, after training (see Figure 10). Correctly, it outputs edge pixels in correspondence to the actual edge pixels.

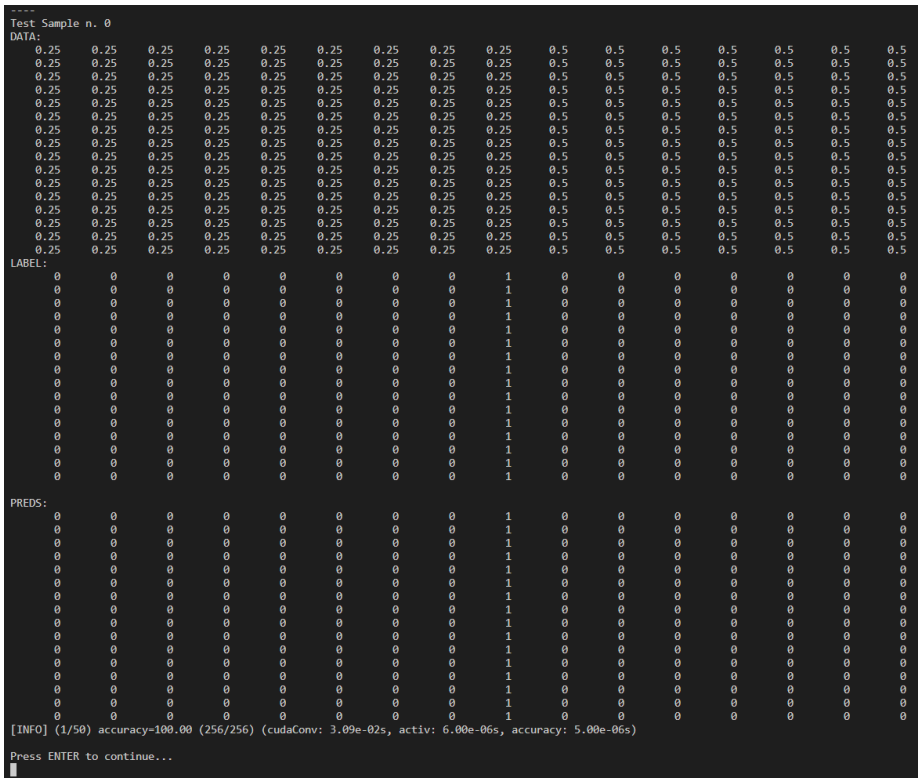


Figure 10: Usage example of *cudaConvGradientDescent.cu* (part 2)

References

[1] Christopher M. Bishop and Hugh Bishop. *Deep Learning: Foundations and Concepts*. Cham: Springer International Publishing, 2024. ISBN: 978-3-031-45467-7 978-3-031-45468-4. DOI: 10.1007/978-3-031-45468-4. URL: https://link.springer.com/10.1007/978-3-031-45468-4 (visited on 01/05/2024).

[2] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[3] Alessio Ferone. "Computer Vision - Recognition". Slides of the lecture.

[4] *Research Computing Facilities*. Research Computing Facilities. URL: https://rcf.uniparthenope.it/en/about/ (visited on 01/11/2024).

[5] *Research Computing Facilities*. Research Computing Facilities. URL: https://rcf.uniparthenope.it/en/resources/ (visited on 01/11/2024).