HAUTE ÉCOLE D'INGÉNIERIE ET DE GESTION DU CANTON DE VAUD

Acceleration of an AI application

Thursday, 02 February 2023

Professor: Marina Zapater

Kevin Jordil & Olivier D'Ancona

STAGE 1 – CHOOSING AN APPLICATION

1 Multi-Layer Perceptron

Multi-Layer Perceptron (MLP) is a type of artificial neural network commonly used for supervised learning tasks such as classification. It consists of one input layer, some hidden layer and one output layer. In each layer, each neuron is connected to all neurons in the next layer. The architecture that we built have 3 layers:

Architecture our neural network implementation is as follows:

- Input layer I of size 784 (28x28 images)
- Hidden layer H of size 1000 (number of hidden neurons)
- Output layer O of size 10 (number of class of the dataset)

We used a c implementation who consists of a single file with all the functions needed to train and test the neural network. The batch size $B = 10$.

Dataset We used the fashion dataset which is a commonly used dataset in machine learning and computer vision task. He consists of a collection of Zalando shopping items such as shirts, pants and shoes. There are in total 10 categories. he class are T-shirt/top, Trouser, Pullover, Dress, Coat, Sandal, Shirt, Sneaker, Bag and Ankle boot. Each image is 28 pixels in height and 28 pixels in width, for a total of 784 pixels in total. Each pixel has a single pixel-value associated with it, indicating the lightness or darkness of that pixel, with higher numbers meaning darker. This pixel-value is an integer between 0 and 255. The training and test data sets have 785 columns. The first column consists of the class labels (see above), and represents the article of clothing. The rest of the columns contain the pixel-values of the associated image.

2 Grayscale Conversion Program

Definition We ran into troubles with the cuda neural network acceleration so we developped an auxiliary program to be accelerated with cuda. The program converts a color image to a grayscale image which is a good idea to implement with our multi layer perceptron. Because our mlp necessitates a grayscale image as input we can convert rgb images to grayscale images with this program. Furthermore this task is a good example to show the acceleration of a program with cuda.

Principle For each pixel of the image, we calculate the average of all color channels. The result is the grayscale value of the pixel. Then we replace the color channels with the grayscale value in a new image.

STAGE 2 – ANALYSING APPLICATION BOTTLENECKS

1 Execution time

MLP baseline The baseline execution time is showed on table 1 with a total time of 575.676547s.

time	accuracy	error	samples/sec	gflop/s
67.270 s	40.99%	0.045	743.28	2.20
134.516 s	61.63%	0.022	743.53	2.20
201.754 s	74.50%	0.019	743.64	2.20
268.999 s	82.60%	0.016	743.54	2.20
336.225 s	87.68%	0.015	743.77	2.20
403.459 s	90.91%	0.013	743.67	2.20
470.688 s	93.01%	0.013	743.73	2.20
537.911 s	94.43%	0.012	743.79	2.20

Table 1: MLP Baseline execution time

MLP accelerated with OpenMP The accelerated execution time is showed on table 2 with a total time of 393.993599s.

time	accuracy	error	samples/sec	gflop/s
46.092 s	40.99%	0.045	1084.79	3.21
92.241 s	61.63%	0.022	1083.43	3.20
138.231 s	74.50%	0.019	1087.20	3.22
184.198 s	82.59%	0.016	1087.76	3.22
230.153 s	87.68%	0.015	1088.01	3.22
276.135 s	90.91%	0.013	1087.40	3.22
322.124 s	93.01%	0.013	1087.23	3.22
368.109 s	94.43%	0.012	1087.32	3.22

Table 2: MLP OpenMP accelerated execution time

MLP accelerated with OpenMP and CUDA The accelerated execution time is showed on table 3 with a total time of 326.487s.

Grayscale baseline In the following results, the image name contains the resolution in pixels. Here is the baseline execution time:

- ./main chicky_512_512.png -> Time elapsed: 3.431923 ms
- ./main avatar_5000_2381.jpg -> Time elapsed: 151.122006 ms
- ./main winter_10667_6000.jpg -> Time elapsed: 803.954043 ms

time	accuracy	error	samples/sec	gflop/s
64.577 s	36.72%	0.061	774.27	2.29
130.425 s	57.61%	0.034	759.32	2.25
195.328 s	70.85%	0.029	770.39	2.28
261.072 s	79.38%	0.025	760.52	2.25
326.487 s	84.74%	0.024	764.35	2.26
391.392 s	88.17%	0.021	770.37	2.28
455.994 s	90.38%	0.021	773.96	2.29
519.951 s	91.87%	0.019	781.79	2.31
584.651 s	92.86%	0.020	772.79	2.29
649.924 s	93.56%	0.018	766.02	2.27
714.872 s	94.03%	0.018	769.84	2.28
780.575 s	94.48%	0.017	761.00	2.25
845.612 s	94.74%	0.016	768.80	2.27

Table 3: MLP OpenMP and CUDA accelerated execution time

Grayscale OpenMP The following results use the same images but with CUDA acceleration:

- ./main chicky_512_512.png -> Time elapsed: 2.095706 ms
- ./main avatar_5000_2381.jpg -> Time elapsed: 61.278724 ms
- ./main winter_10667_6000.jpg -> Time elapsed: 209.700440 ms

2 Complexity Analysis

MLP The MLP is a feedforward neural network, so the complexity is bounded by the biggest loop, which is the backpropagation loop in $O(\#epoch \cdot H \cdot O \cdot B)$

Grayscale The complexity is one computation for every pixel so $O(m \cdot n)$ where m is the number of rows and n is the number of columns of the image.

3 Theoretical acceleration

Hardware The theoretical limitations depends on the hardware. In our case, we use a jetson nano which has a max threads/block bound of 1024 and 128 CUDA cores. Therefore, if we want to accelerate our application properly, we need want to maximize the number of thread running the same operation on the warps. In our case, we have a lot of small operations in parallel and we therefore we can't use the full power of the GPU because memory transfer will be too long.
If we analyze the GPU time only with nprof on 4, we can use Amdahl's law to calculate the theoretical acceleration. The kernel use 11.69 second, and the memcpy use 15.41 second and the memset use 5.6s. The part we can accelerate is only over those 11.69seconds. Therefore, because the total GPU time takes more than running on the CPU baseline, we can't accelerate the application with the GPU faster than the cpu.

Roofline The max performance is 472 GFLOPs and the max memory bandwidth is 25.6G GB/s. Therefore, the jetson is balanced with CI = 18.43. If we calculate the number of operations on the backpropagation kernel we have: $Y * H * B * 18op = 1.8M$. The number of data transfered is $Y * H * B * 4Bytes = 0.4M$. Therefore the CI of our kernel is 4.5. Using the symetry in the roofline

model our kernel performance is $472/18.43 * 4.5 = 115.24 GFLOPs$. As we can see, we are not limited by the performance but by the memory.

STAGE 3 – ACCELERATION

1 Process

First, we added OpenMP to the base code. As a result, we have a slight performance increase. Then we wanted to add a CUDA kernel function as requested. However, the performance was not as good as the original code. So we spent a lot of time trying to optimize the CUDA code. We tried to make several CUDA kernel functions, to use cuBLAS, to make memory optimizations. Unfortunately, we never managed to improve the code base with CUDA because the transfer between the CPU and the GPU took more time than the CUDA kernel function itself. Therefore, we decided to make a second application and optimize it with a CUDA kernel function.

2 Accelerated parts

Choice We chose to keep the optimization only with OpenMP on MLP because it is more efficient alone than with CUDA optimizations. To respect the instructions and accelerate part of the code on GPU, we used the CUDA runtime library with CUDA kernel functions which allowed us to have good results. We looked at using cuBLAS but it was not adapted because it is intended for matrix calculations and it is not really what is done in our example. We did not use cuDNN because it is not suitable for our needs as it is specifically created for deep neural networks.

Data For MLP, we use the Fashion MINST dataset as explained above. So we will have several large arrays but mostly loops that will go through these arrays. So we found three triple for loops that took a lot of execution time according to us. In a first step, we regenerated a single triple loop with these three loops, which will simplify the transition to CUDA kernel. In a second step, we put all the data on the GPU. Then, we created a CUDA kernel function which allows to replace the generated triple for loop.

Listing 1: "Original code with triple loop"

```
/* dy */
for (int b = 0; b < B; b++)
  for (int k = 0; k < Y; k++)
  dy[b * Y + k] = p[b * Y + k] - t[b * Y + k];
/* dv := h * dy' */
for (int b = 0; b < B; b++)
  for (int j = 0; j < H; j++)
  for (int k = 0; k < Y; k++)
    dv[k * H + j] += h[b * H + j] * dy[b * Y + k];
/* dh := v * dy */
for (int b = 0; b < B; b++)
  for (int j = 0; j < H; j++)
  for (int k = 0; k < Y; k++)
    dh[b * H + j] += v[k * H + j] * dy[b * Y + k];
```

Listing 2: "Edited nested triple loop"

```
/* dy */
for (int b = 0; b < B; b++)
  for (int k = 0; k < Y; k++)
  for (int j = 0; j < H; j++)
  {
```

```
      dy[b * Y + k] = p[b * Y + k] - t[b * Y + k];
      dv[k * H + j] += h[b * H + j] * dy[b * Y + k];
      dh[b * H + j] += v[k * H + j] * dy[b * Y + k];
    }
```

Listing 3: "CUDA kernel function to calculate back_propagation"

```
// Cuda kernel function of loop
__global__ void backprop_kernel(float *p, float *t, float *dv, float *v, float
   *dh, float *h)
{
  int y_idx = threadIdx.x + blockIdx.x * blockDim.x;
  int h_idx = threadIdx.y + blockIdx.y * blockDim.y;

  if (y_idx >= H || h_idx >= Y)
    return;

  float dy, dh_gpu, dv_gpu;

  for(int b_idx = 0; b_idx < B; b_idx++){
    dy = p[b_idx * Y + h_idx] - t[b_idx * Y + h_idx];
    dv_gpu = h[b_idx * H + y_idx] * dy;
    dh_gpu = v[h_idx * H + y_idx] * dy;
    atomicAdd(&dv[h_idx * H + y_idx], dv_gpu);
    atomicAdd(&dh[b_idx * H + y_idx], dh_gpu);
  }
}

dim3 dimBlock(32, 32, 1);
dim3 dimGrid(Y / 32 + 1, H / 32 + 1, 1);
backprop_kernel<<<dimGrid, dimBlock>>>(p_gpu, t_gpu, dv_gpu, v_gpu, dh_gpu,
  h_gpu);
```

As said before, this code is still slower than the original and using OpenMP alone is faster. So there is an OpenMP directive (#pragma omp parallel for) before each for loop. The number of threads is set to 4.
For the application that transforms the image into grayscale, there is a double for loop that runs through the image. So we created a CUDA kernel function that allows to replace the generated double for loop and thus to use the GPU computing power.

Listing 4: "Original grayscale code"

```
for (int y = 0; y < height; y++)
{
  for (int x = 0; x < width; x++)
  {
    Vec3b color = color_image.at<Vec3b>(y, x);
    int gray = (color[0] + color[1] + color[2]) / 3;
    gray_image.at<unsigned char>(y, x) = gray;
  }
}
```

Listing 5: "CUDA kernel function to convert image to grayscale"

```
__global__ void convertToGray(uchar3 *input, unsigned char *output, int width,
   int height)
{
  int x = blockIdx.x * blockDim.x + threadIdx.x;
  int y = blockIdx.y * blockDim.y + threadIdx.y;

  if (x < width && y < height)
```

```
    {
      output[y * width + x] = (input[y * width + x].x + input[y * width + x].y +
     input[y * width + x].z) / 3;
    }
  }

  dim3 block(BLOCK_SIZE, BLOCK_SIZE);
  dim3 grid((width + block.x - 1) / block.x, (height + block.y - 1) / block.y);

  convertToGray<<<grid, block>>>(d_color_image, d_gray_image, width, height);
```

STAGE 4 – ANALYSIS OF RESULTS

Given the work you performed in Stage 3, we want you know to analyse the results obtained and draw some conclusions: • Analysis of the results: what is the performance enhancement achieved? Where does it come from and why? What is the new bottleneck created? • Potential future lines: seeing the acceleration results, what other things should you accelerate? Now that you see the results, should you have done something different?

MLP Analysis As we can see, the OpenMP implementation is the fastest then the baseline and finally the CUDA and OpenMP is the slowest. The CUDA profiler results are showed on table 4. The CUDA profiler shows that the CUDA memcpy is the most time consuming operation, followed by the backprop_kernel and the CUDA memset. So our kernel function is too short to benefit from the GPU acceleration. The cost to move the data to memory are too high.

Grayscale Analysis For the grayscale algorithm, we can see that the acceleration is substantial. For instance, for the winter image the gpu algorithm is 4 time faster.

Type	Time perc.	Time	Calls	Avg	Name
GPU :	47.10%	15.4163s	410754	37.531us	CUDA memcpy HtoH
	35.72%	11.6917s	68459	170.78us	backprop_kernel
	17.18%	5.62353s	136918	41.072us	CUDA memset
API calls:	74.08%	145.662s	410754	354.62us	cudaMemcpy
	13.21%	25.9788s	136918	189.74us	cudaMemset
	9.01%	17.7185s	68459	258.82us	cudaDeviceSynchronize
	3.49%	6.86282s	68459	100.25us	cudaLaunchKernel
	0.18%	344.80ms	7	49.257ms	cudaMallocManaged
	0.04%	72.091ms	68459	1.0530us	cudaGetLastError
	0.00%	953.77us	7	136.25us	cudaFree
	0.00%	108.55us	97	1.1190us	cuDeviceGetAttribute
	0.00%	10.521us	1	10.521us	cuDeviceTotalMem
	0.00%	7.3430us	3	2.4470us	cuDeviceGetCount
	0.00%	3.8540us	2	1.9270us	cuDeviceGet
	0.00%	1.5100us	1	1.5100us	cuDeviceGetName
	0.00%	938ns	1	938ns	cuDeviceGetUuid

Table 4: MLP CUDA accelerated GPU profiling results