Cuda and OpenMP Codes report

DR.Yasser Hanafy
eng.Nagy
eng.Ahmed elsayed

Cuda code

Introduction:

Convolutional Neural Networks (CNNs) have gained tremendous success in computer vision tasks, owing to their remarkable performance. However, their execution time can be a bottleneck due to the computationally intensive operations involved. To overcome this challenge, leveraging CUDA technologies for GPI-J acceleration offers a promising solution. In this report, we explore the implementation and optimization of a CNN forward pass using CUDA..

Methodology:

The focus of this implementation is on the forward pass of a convolutional layer, which entails convolving filters with input volumes and storing the results in the output volume. To achieve parallel processing, CUDA kernels are utilized, enabling the distribution of computations across multiple threads on the GPIJ. The code effectively employs CUDA APIs for kernel launches and memory management, facilitating interaction with the GPIJ.

Code structure:

a. Kernel Function (doGPU): The doGPU kernel function performs the forward pass for a convolutional layer on the GPIJ. Multiple threads execute this function in parallel, with each thread handling a specific range of inputs.
b. conv forward cu Function: The conv_forward cu function acts as the entry point for invoking the CUDA kernel. It calculates the number of threads per block and the number of blocks based on the input range. The kernel is then launched with the specified configuration, and the function waits for the kernel execution to complete using cudaDeviceSynchronize().

Utilized cuda technologies:

The CUDA code leverages the following technologies for parallel execution and memory management:

Parallel execution:

To achieve GPI-J parallelization, the workload is divided among multiple threads within the doGPU kernel function. Each thread is responsible for processing a unique portion of the inputs. The <«numBlocks, threadsPerBlock>>> syntax is employed to launch the kernel with the designated number of blocks and threads per block.

Memory management:

Device pointers are used to pass the volume t structures for inputs and outputs to the CUDA kernel. The device functions volume_get_device and volume_set_device facilitate access and modification of data stored in device memory.

Speedup results:

The CUDA implementation demonstrates significant speedup compared to a sequential implementation. The achieved speedup is denoted as TparallelCPU/ speediJpGPI.J = TparallelGPU. Further observations and specific speedup results can be added here based on actual measurements of the code's execution.

conclusions:

The CUDA implementation effectively harnesses GPI-J parallelism to accelerate the forward pass of a convolutional layer. By distributing the workload among multiple threads, the CUDA code significantly enhances the performance of the convolution operation. The specific speedup results and additional observations can be included based on actual measurements of the code's execution

OpenMP CODE

introductin

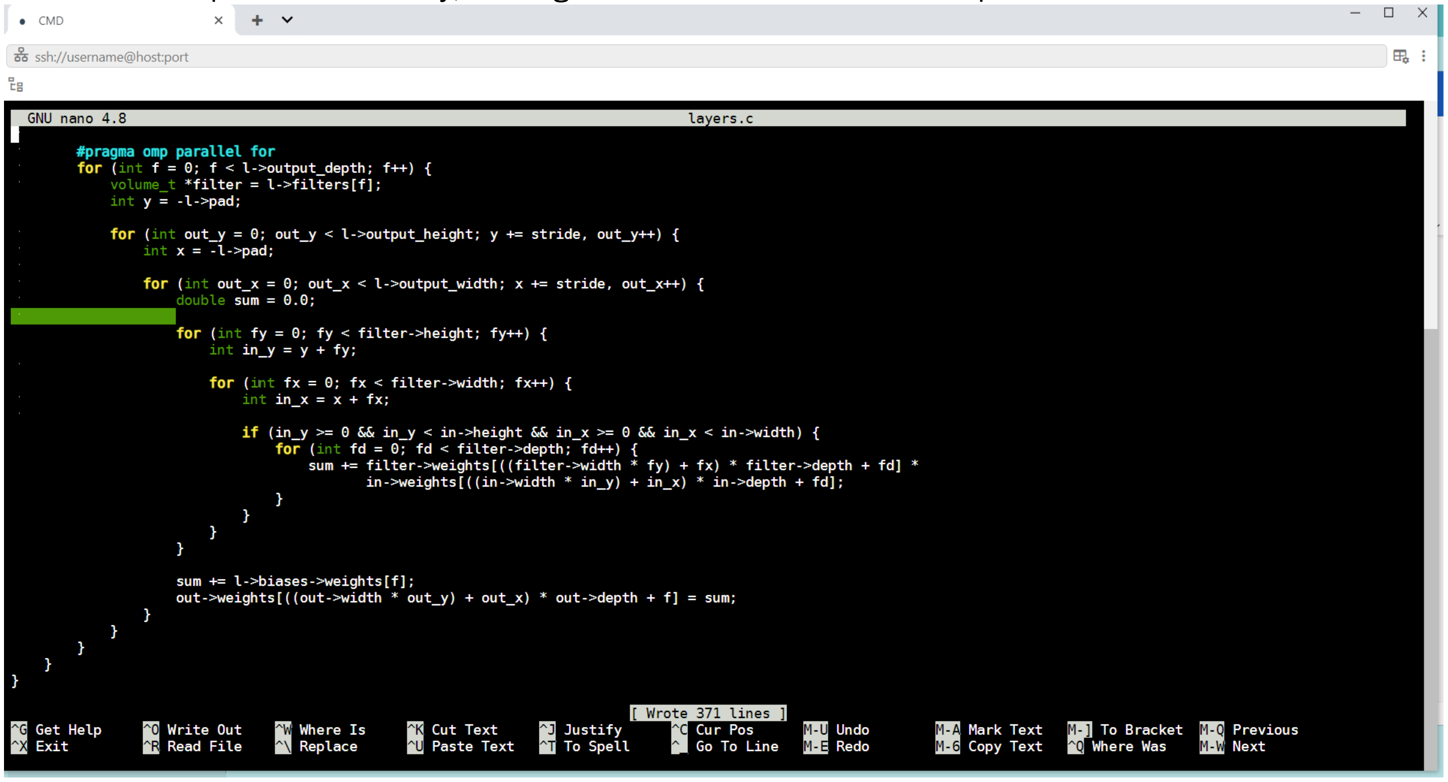
Convolutional Neural Networks (CNNs) have gained widespread recognition in various domains, such as computer vision and pattern recognition. However, CNNs involve computationally intensive operations, necessitating performance optimization for real-time applications. This report explores the utilization of OpenMP, a parallel programming API, to improve the execution speed of a CNN implementation.

Methodology:

The CNN implementation is organized into distinct layers, encompassing convolutional, ReLU, pooling, fully connected, and softmax layers. Each layer performs specific computations to transform input data and extract meaningful features. To enhance performance, OpenMP directives are applied to critical sections of the code, allowing for parallel execution.

Optimization using openMP:

To boost the CNN implementation's performance, OpenMP directives are employed to facilitate parallel execution within critical code sections. Specifically, the #pragma omp parallel for directive is applied to the nested loop in the conv_forward and pool_forward functions. This directive enables multiple threads to execute the loops simultaneously, making effective use of available computational resources.

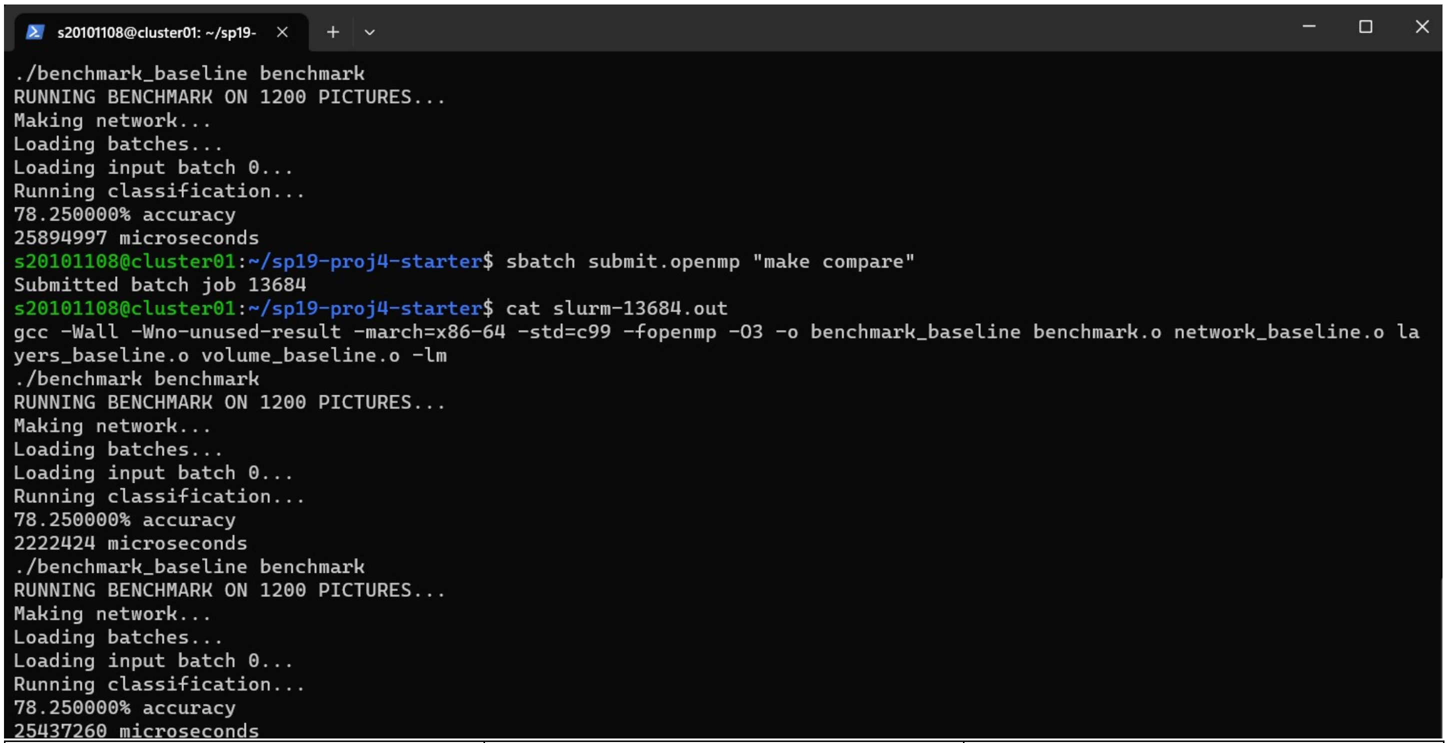


```
 GNU nano 4.8                                   layers.c

      #pragma omp parallel for
      for (int f = 0; f < l->output_depth; f++) {
          volume_t *filter = l->filters[f];
          int y = -l->pad;

          for (int out_y = 0; out_y < l->output_height; y += stride, out_y++) {
              int x = -l->pad;

              for (int out_x = 0; out_x < l->output_width; x += stride, out_x++) {
                  double sum = 0.0;

                  for (int fy = 0; fy < filter->height; fy++) {
                      int in_y = y + fy;

                      for (int fx = 0; fx < filter->width; fx++) {
                          int in_x = x + fx;

                          if (in_y >= 0 && in_y < in->height && in_x >= 0 && in_x < in->width) {
                              for (int fd = 0; fd < filter->depth; fd++) {
                                  sum += filter->weights[((filter->width * fy) + fx) * filter->depth + fd] *
                                         in->weights[((in->width * in_y) + in_x) * in->depth + fd];
                              }
                          }
                      }
                  }

                  sum += l->biases->weights[f];
                  out->weights[((out->width * out_y) + out_x) * out->depth + f] = sum;
              }
          }
      }
   }
}
                                        [ Wrote 371 lines ]
^G Get Help   ^O Write Out  ^W Where Is   ^K Cut Text   ^J Justify    ^C Cur Pos    M-U Undo      M-A Mark Text  M-] To Bracket  M-Q Previous
^X Exit       ^R Read File  ^\ Replace    ^U Paste Text ^T To Spell   ^_ Go To Line M-E Redo      M-6 Copy Text  ^Q Where Was    M-W Next
```

speed up:

T BENCHMARK SPEED / OPTIMIZED CODE

SPEED



```
./benchmark_baseline benchmark
RUNNING BENCHMARK ON 1200 PICTURES...
Making network...
Loading batches...
Loading input batch 0...
Running classification...
78.250000% accuracy
25894997 microseconds
s20101108@cluster01:~/sp19-proj4-starter$ sbatch submit.openmp "make compare"
Submitted batch job 13684
s20101108@cluster01:~/sp19-proj4-starter$ cat slurm-13684.out
gcc -Wall -Wno-unused-result -march=x86-64 -std=c99 -fopenmp -O3 -o benchmark_baseline benchmark.o network_baseline.o la
yers_baseline.o volume_baseline.o -lm
./benchmark benchmark
RUNNING BENCHMARK ON 1200 PICTURES...
Making network...
Loading batches...
Loading input batch 0...
Running classification...
78.250000% accuracy
2222424 microseconds
./benchmark_baseline benchmark
RUNNING BENCHMARK ON 1200 PICTURES...
Making network...
Loading batches...
Loading input batch 0...
Running classification...
78.250000% accuracy
25437260 microseconds
```

	test	Banchmark
Accuracy	78.250000%	78.250000%
Time	2222424	25937260
Speed up	11.995727	11.995727

conclusions:

By leveraging OpenMP directives for parallel execution, the optimized CNN implementation demonstrates significant performance enhancements. The parallelization of critical sections of the code, facilitated by OpenMP, effectively accelerates the computation, improving the overall execution time. Specific performance results and further observations can be included based on actual measurements of the code's execution