# SPP Assignment 2

## 2023101045

**Note: Throughout the assignment, only the official `detect.py` was used to make any inference, and it was modified for optimization.**

## System Specs

I chose to use google colab for bench marking the models as my laptop lacked a dedicated GPU.

**Specs:**

→ CPU : Intel(R) Xeon(R) CPU @ 2.00GHz

→ GPU : Tesla T4

→ RAM: 12.7 GB

## Data Set Used

I used 100 images from the COCO dataset.

Link: http://images.cocodataset.org/zips/val2017.zip

## Latency - FPS Output

Performance comparison:

| | model | latency | fps |
|---|---|---|---|
| 0 | yolov5n | 14.1 | 7.993569 |
| 1 | yolov5s | 15.0 | 9.390652 |
| 2 | yolov5m | 23.7 | 8.512493 |
| 3 | yolov5l | 25.7 | 8.265063 |
| 4 | yolov5x | 41.2 | 6.480341 |

## GFLOPS Calculation

```
# Print the results
print("The GPU used has a peak flop performance of 8.1 TFLOPS/s - (T4 GPU on google colab)")
print(model_stats_df)
```

```
The GPU used has a peak flop performance of 8.1 TFLOPS/s - (T4 GPU on google colab)
     Model  Parameters (M)  Model Size (MB)  GFLOPS per inference  GFLOPS/sec  \
0  yolov5n            1.87             7.12                  4.50      318.87
1  yolov5s            7.23            27.56                 16.49     1099.01
2  yolov5m           21.17            80.77                 48.97     2066.13
3  yolov5l           46.53           177.51                109.15     4246.89
4  yolov5x           86.71           330.75                205.67     4991.97

   Utilization (%) Bound Type
0             0.04     memory
1             0.14     memory
2             0.26     memory
3             0.52     memory
4             0.62     memory
```

Clearly, the maximum utilization is not achieved and hence the inference process is memory bound.

I have done the per layer granularity evaluation for yolov5s model, and here are my results:
The layers with highest utlisation are:

- *model.model.20.m.0.cv2.conv (42.92%)*
  - This convolutional layer has the highest utilization at nearly 43%
  - It achieves 128.75 GFLOPS/s performance.
  - This indicates efficient hardware usage for this particular layer
- *model.model.4.m.1.cv2.conv (40.83%)*
  - Close second with ~41% utilization
  - Shows strong compute efficiency with 122.50 GFLOPS/s
- *model.model.8.m.0.cv2.conv (40.31%)*
  - Maintains high efficiency at ~40% utilization
  - Performs at 120.92 GFLOPS/s

The layers with least utilisation are:

- *model.model.11 (0.16%)*

    - Extremely low utilization of just 0.16%
    - Very poor performance at only 0.47 GFLOPS/s
- And many more...

Several layers show 0.00% utilization, particularly activation layers (with .act suffix) which are likely implemented using operations that don't require significant compute and might be fused with other operations.

*Performance Insights*:

- Convolutional layers show high compute utilization (30-43%), while activation layers are memory-bound with negligible utilization.
- *Bottlenecks*: Overall model utilization is just 16.85%, indicating significant optimization potential.
- Memory bandwidth is likely the limiting factor, with uneven workload distribution (utilization ranging from 0.16% to 42.92%).
- Throughput: The current inference time of 162.61ms gives 6.15 FPS, which is far below the potential of the hardware. Average performance is only 50.54 GFLOPS/s, suggesting room for better utilization.

# Task 3

For yolov5s

1. **Torch.Profiler**

| Name | Self CPU % | Self CPU | CPU total % | CPU total | CPU time avg | Self CUDA | Self CUDA % | CUDA total | CUDA time avg | CPU Mem | Self CPU Mem | CUDA Mem | Self CUDA Mem | # of Calls |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| model_inference | 35.39% | 6.488ms | 99.91% | 18.315ms | 18.315ms | 0.000us | 0.00% | 11.833ms | 11.833ms | 0 b | 0 b | -92.50 Kb | -179.55 Mb | 1 |
| aten::conv2d | 1.37% | 251.790us | 48.60% | 8.908ms | 148.466us | 0.000us | 0.00% | 10.005ms | 166.754us | 0 b | 0 b | 98.47 Mb | 0 b | 60 |
| aten::convolution | 1.28% | 235.063us | 47.22% | 8.656ms | 144.270us | 0.000us | 0.00% | 10.005ms | 166.754us | 0 b | 0 b | 98.47 Mb | 0 b | 60 |
| aten::_convolution | 4.44% | 814.420us | 45.94% | 8.421ms | 140.352us | 0.000us | 0.00% | 10.005ms | 166.754us | 0 b | 0 b | 98.47 Mb | 0 b | 60 |
| aten::cudnn_convolution | 26.49% | 4.859ms | 33.18% | 6.082ms | 101.363us | 8.091ms | 75.14% | 8.891ms | 148.186us | 0 b | 0 b | 98.47 Mb | 98.47 Mb | 60 |
| cudaLaunchKernel | 11.86% | 2.174ms | 11.86% | 2.174ms | 7.764us | 0.000us | 0.00% | 0.000us | 0.000us | 0 b | 0 b | 0 b | 0 b | 280 |
| aten::add_ | 4.05% | 741.965us | 6.29% | 1.152ms | 19.203us | 1.114ms | 9.41% | 1.114ms | 18.568us | 0 b | 0 b | 0 b | 0 b | 60 |
| aten::silu_ | 2.25% | 411.647us | 4.40% | 807.256us | 14.162us | 647.831us | 5.47% | 647.831us | 11.365us | 0 b | 0 b | 0 b | 0 b | 57 |
| aten::cat | 2.37% | 434.889us | 3.76% | 690.157us | 34.508us | 531.226us | 4.49% | 531.226us | 26.561us | 0 b | 0 b | 48.44 Mb | 48.44 Mb | 20 |
| aten::reshape | 0.63% | 115.992us | 2.03% | 372.801us | 6.213us | 0.000us | 0.00% | 0.000us | 0.000us | 0 b | 0 b | 0 b | 0 b | 60 |
| aten::arange | 0.55% | 101.208us | 1.88% | 345.072us | 28.756us | 14.880us | 0.13% | 29.760us | 2.480us | 0 b | 0 b | 6.00 Kb | 0 b | 12 |
| aten::mul | 0.99% | 182.326us | 1.56% | 286.109us | 19.074us | 72.511us | 0.61% | 72.511us | 4.834us | 0 b | 0 b | 789.50 Kb | 789.50 Kb | 15 |
| aten::view | 1.49% | 273.628us | 1.49% | 273.628us | 3.648us | 0.000us | 0.00% | 0.000us | 0.000us | 0 b | 0 b | 0 b | 0 b | 75 |
| aten::add | 0.57% | 105.038us | 0.90% | 165.421us | 16.542us | 99.901us | 0.84% | 99.901us | 9.990us | 0 b | 0 b | 9.18 Mb | 9.18 Mb | 10 |
| cudaEventRecord | 0.90% | 165.076us | 0.90% | 165.076us | 2.751us | 0.000us | 0.00% | 0.000us | 0.000us | 0 b | 0 b | 0 b | 0 b | 60 |
| aten::stack | 0.10% | 18.148us | 0.77% | 140.436us | 46.812us | 0.000us | 0.00% | 19.007us | 6.336us | 0 b | 0 b | 66.00 Kb | 0 b | 3 |
| aten::contiguous | 0.04% | 8.051us | 0.73% | 133.702us | 44.567us | 0.000us | 0.00% | 208.701us | 69.567us | 0 b | 0 b | 8.18 Mb | 0 b | 3 |
| aten::clone | 0.14% | 25.283us | 0.69% | 125.651us | 41.884us | 0.000us | 0.00% | 208.701us | 69.567us | 0 b | 0 b | 8.18 Mb | 0 b | 3 |
| cudaMemsetAsync | 0.52% | 95.929us | 0.52% | 95.929us | 5.643us | 0.000us | 0.00% | 0.000us | 0.000us | 0 b | 0 b | 0 b | 0 b | 17 |
| aten::max_pool2d | 0.07% | 13.218us | 0.47% | 86.427us | 28.809us | 0.000us | 0.00% | 112.287us | 37.429us | 0 b | 0 b | 1.17 Mb | -2.34 Mb | 3 |
| aten::select | 0.38% | 70.112us | 0.45% | 82.944us | 5.530us | 0.000us | 0.00% | 0.000us | 0.000us | 0 b | 0 b | 0 b | 0 b | 15 |
| aten::sub | 0.30% | 54.354us | 0.43% | 78.023us | 26.008us | 13.280us | 0.11% | 13.280us | 4.427us | 0 b | 0 b | 197.00 Kb | 197.00 Kb | 3 |
| aten::max_pool2d_with_indices | 0.26% | 46.914us | 0.40% | 73.211us | 24.404us | 112.287us | 0.95% | 112.287us | 37.429us | 0 b | 0 b | 3.52 Mb | 3.52 Mb | 3 |
| aten::pow | 0.26% | 47.461us | 0.39% | 71.328us | 23.776us | 9.312us | 0.08% | 9.312us | 3.104us | 0 b | 0 b | 197.00 Kb | 197.00 Kb | 3 |
| aten::upsample_nearest2d | 0.21% | 38.989us | 0.37% | 68.010us | 34.005us | 49.824us | 0.42% | 49.824us | 24.912us | 0 b | 0 b | 4.69 Mb | 4.69 Mb | 2 |

Self CPU time total: 18.331ms
Self CUDA time total: 11.833ms

## 2. Torch.utils.bottleneck

```
-----------------------  ------------  ------------  ------------  ------------  ------------  ------------
                   Name    Self CPU %      Self CPU   CPU total %     CPU total  CPU time avg    # of Calls
-----------------------  ------------  ------------  ------------  ------------  ------------  ------------
         aten::uniform_        33.14%      11.454ms        33.14%      11.454ms      11.454ms             1
         aten::uniform_        12.99%       4.490ms        12.99%       4.490ms       4.490ms             1
         aten::uniform_        11.34%       3.920ms        11.34%       3.920ms       3.920ms             1
         aten::uniform_         9.49%       3.282ms         9.49%       3.282ms       3.282ms             1
         aten::uniform_         8.45%       2.919ms         8.45%       2.919ms       2.919ms             1
              aten::to          0.03%       9.814us         7.63%       2.637ms       2.637ms             1
         aten::_to_copy         0.05%      18.286us         7.60%       2.628ms       2.628ms             1
    aten::empty_strided         7.34%       2.536ms         7.34%       2.536ms       2.536ms             1
         aten::uniform_         5.56%       1.920ms         5.56%       1.920ms       1.920ms             1
         aten::uniform_         4.44%       1.535ms         4.44%       1.535ms       1.535ms             1
         aten::uniform_         4.36%       1.507ms         4.36%       1.507ms       1.507ms             1
              aten::to          0.01%       3.726us         3.00%       1.038ms       1.038ms             1
         aten::_to_copy         0.02%       8.283us         2.99%       1.034ms       1.034ms             1
            aten::copy_         0.04%      14.551us         2.91%       1.006ms       1.006ms             1
       cudaMemcpyAsync         2.74%     946.961us         2.74%     946.961us     946.961us             1
-----------------------  ------------  ------------  ------------  ------------  ------------  ------------
Self CPU time total: 34.564ms

-------------------------------------------------------------------------------
  autograd profiler output (CUDA mode)
-------------------------------------------------------------------------------
        top 15 events sorted by cpu_time_total

        Because the autograd profiler uses the CUDA event API,
        the CUDA time column reports approximately max(cuda_time, cpu_time).
        Please ignore this output if your code does not use CUDA.

-------------------  ------------  ------------  ------------  ------------  ------------  ------------  ------------  ------------  ------------  -----------
                Name    Self CPU %      Self CPU   CPU total %     CPU total  CPU time avg     Self CUDA   Self CUDA %    CUDA total  CUDA time a
-------------------  ------------  ------------  ------------  ------------  ------------  ------------  ------------  ------------  -----------
      aten::uniform_        33.05%       9.193ms        33.05%       9.193ms       9.193ms       9.208ms        31.86%       9.208ms       9.208m
      aten::uniform_        11.92%       3.315ms        11.92%       3.315ms       3.315ms       3.330ms        11.52%       3.330ms       3.330m
      aten::uniform         11.75%       3.269ms        11.75%       3.269ms       3.269ms       3.285ms        11.36%       3.285ms       3.285m
```

(FULL OUTPUTS CAN BE SEEN IN THE NOTEBOOK)

### Time Breakdown across Profiling Stages

Obtained by running `line_profiler` on the custom `detect.py` file provided by the yolov5 official repository.

| Percent of time | Line | Stage |
|---|---|---|
| 2.3 | device = select_device(device) | Loading (model) |
| 84.0 | model = DetectMultiBackend(weights, device=device, dnn=dnn, data=data, fp16=half) | Loading (model) |
| 2.4 | model.warmup(imgsz=(1 if pt or model.triton else bs, 3, *imgsz)) | Loading (Image) |
| 0.4 | *** | Pre-Processing |
| 1.4 | pred = model(im, augment=augment, visualize=visualize) | Inference |
| 6.1 | pred = non_max_suppression(pred, conf_thres, iou_thres, classes, agnostic_nms, | Post-Processing |

| Percent of time | Line | Stage |
|---|---|---|
| | max_det=max_det) | |
| Rest | | Post-Processing (Printing Results, saving to file etc) |

***

```
for path, im, im0s, vid_cap, s in dataset:
    with dt[0]:
        im = torch.from_numpy(im).to(model.device)
        im = im.half() if model.fp16 else im.float()  # uint8 to fp16/32
        im /= 255  # 0 - 255 to 0.0 - 1.0
        if len(im.shape) == 3:
            im = im[None]  # expand for batch dim
        if model.xml and im.shape[0] > 1:
            ims = torch.chunk(im, im.shape[0], 0)
```

Most of the time was spent in loading the model. Therefore, `detect.py` was modified to print the times after the model loaded and the following result was obtained:

```
--- Pipeline Breakdown (after model load) ---
Pre-processing:  0.20%
Inference:       14.51%
Post-processing: 85.29%
```

### CPU vs GPU Time

From the output of the torch.profiler, we see that the result is as follows for a particular run on a random test image:

```
Self CPU time total: 17.536ms
Self CUDA time total: 11.835ms
```

## Memory or threading issues

**TLDR: For YOLOV5s , no memory issues observed.**

→ We used `torch.utils.bottleneck` to profile the entire inference pipeline on `yolov5s` (For the full report, see above under task 3)

→ If there were threading bottlenecks (e.g., excessive time in Python's `threading` module or contention between threads), it would be reported in the cProfile section — it's not.

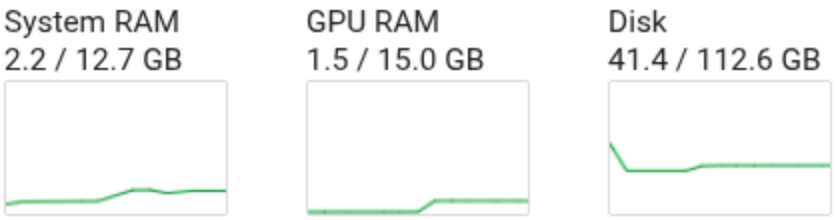
→ The tool did not report any threading bottlenecks or contention, suggesting the current pipeline does not suffer from Python GIL limitations or data loader inefficiencies.

→ Autograd and CUDA profiler outputs confirmed that major operations are dominated by `aten::uniform_` and convolution layers, with no evidence of memory leaks or synchronization stalls.

→ CUDA uses 1.5GB out of 15.36 GB of GPU memory available.

```
+-----------------------------------------------------------------------------------------+
| NVIDIA-SMI 550.54.15              Driver Version: 550.54.15      CUDA Version: 12.4     |
|-----------------------------------------+------------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id          Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util  Compute M. |
|                                         |                        |               MIG M. |
|=========================================+========================+======================|
|   0  Tesla T4                       Off |   00000000:00:04.0 Off |                    0 |
| N/A   75C    P0             32W /   70W |    1526MiB /  15360MiB |      0%      Default |
|                                         |                        |                  N/A |
+-----------------------------------------+------------------------+----------------------+

+-----------------------------------------------------------------------------------------+
| Processes:                                                                              |
|  GPU   GI   CI        PID   Type   Process name                              GPU Memory |
|        ID   ID                                                               Usage      |
|=========================================================================================|
+-----------------------------------------------------------------------------------------+
```



# Task 4

## Approach 1: FP16

Mixed precision (FP16) inference is a well-known optimization technique that reduces model size and computational cost by using 16-bit floating point numbers instead of the standard 32-bit format. The key expectation is improved throughput (FPS) and reduced latency due to faster memory transfers and increased parallelism, especially on GPUs with Tensor Cores like the NVIDIA T4.

## Analysis

Changing to FP16 definitely reduces the size of the model; however the expected performance gain by reduced computation time is not observed **for yolov5s**.

```
[35] # FP32 s
     print(benchmark('yolov5s'))
```

```
{'model': 'yolov5s', 'latency': 14.8, 'fps': 10.452216485549163}
```

```
# Modify Source code to use FP16

#FP16 s
print(benchmark('yolov5s'))
```

```
{'model': 'yolov5s', 'latency': 18.6, 'fps': 9.639478825594844}
```

A potential cause might be that YOLOv5s is already small and fast — FP16 might not bring noticeable benefit. In fact, the worse performance could be due to the overhead that conversion to FP16 brings.

Hence we try for larger models and indeed we see a **massive latency improvement of ~ 50%**. In fact, the latency here also includes the time for loading the model at the beginning. Therefore, the performance gain is slightly better than 50%.

```
[52] #FP32 X
     print(benchmark('yolov5x'))
```

```
{'model': 'yolov5x', 'latency': 42.8, 'fps': 7.2051700580333184}
```

```
# fp16 X
print(benchmark('yolov5x'))
```

```
{'model': 'yolov5x', 'latency': 25.6, 'fps': 7.952850937818522}
```

**CONCLUSION: FP16 shines with large batches and complex models.**

| Model | Latency with FP16 (ms) | Latency with FP32 (ms) |
|---|---|---|
| yolov5s | 18.6 | 14.8 |
| yolov5x | 25.6 | 42.8 |

## Approach 2: Batch Processing

| Component | Approx. Memory Usage |
|---|---|
| YOLOv5s model | ~28MB |
| One 640×640 FP32 image | ~4.7MB |
| Intermediate tensors | Depends on batch |
| Overhead (CUDA, others) | ~1.5–2GB |

So with ~14GB available after overhead, a **safe batch size estimate**:

- **FP32:** `batch_size ≈ 14,000 / 4.7 ≈ 297` → but realistically, 32–64 is safe due to tensor growth.

For the smaller model, batching did lead to an improvement both in latency and to some extent even for throughput in some runs **however this improvement was not reproducible; the subsequent runs did not produce improvement. Similarly, no improvement was observed for larger models.**

```
[65] # yolov5s Batch size: 1
     print(benchmark('yolov5s'))
```

```
{'model': 'yolov5s', 'latency': 24.2, 'fps': 8.98612761578806}
```

```
     # Modify source code
     # yolov5s Batch Size: 32
     print(benchmark('yolov5s'))
```

```
{'model': 'yolov5s', 'latency': 19.3, 'fps': 9.585152053050635}
```

```
[75] # yolov5s Batch size 100
     print(benchmark('yolov5s'))
```

```
{'model': 'yolov5s', 'latency': 15.1, 'fps': 10.759021666929808}
```

```
     # yolov5x Batch size 1
     print(benchmark('yolov5x.pt'))
```

```
{'model': 'yolov5x.pt', 'latency': 42.0, 'fps': 7.179382335781906}
```

```
[26] # yolov5x Batch size 32
     print(benchmark('yolov5x.pt'))
```

```
{'model': 'yolov5x.pt', 'latency': 41.5, 'fps': 7.255651637438981}
```

```
[ ] # yolov5x Batch size 100
    print(benchmark('yolov5x'))
```

```
{'model': 'yolov5x', 'latency': 44.0, 'fps': 7.343453094493132}
```

The potential reason is that the official script used for running the inference `detect.py` in YOLOv5 **loads and infers images one-by-one**, even if you point it to a folder with 100 images.

Therefore, I wrote a custom script to completely bypass the `detect.py.` (Note that this does not include model load times like `detect.py`).

The goal of batching is to perform inference on multiple images simultaneously in a single forward pass through the neural network. The code below implements

this by manually constructing a batch tensor and passing it to the YOLOv5 model in one go:

```
batch = torch.stack(imgs).to('cuda').half()
output = model(batch)
```

The performance still did not improve. The results are as follows:

| model | Batch Size | Pruned Latency |
|---|---|---|
| s | 1 | 8.96 |
| s | 16 | 7.75 |
| s | 32 | 7.85 |
| s | 64 | 8.2 |
| s | 100 | 8.12 |

Batching is most effective when dealing with large workloads where the computational cost of inference dominates. In our case, we processed a relatively small dataset (e.g., 128 images), and used a model ( `yolov5s` ) that is already highly optimized for fast, per-image inference. As a result, the total inference time is already quite low, and the additional gains from batching are limited or hidden under other forms of overhead.

Additionally, batching introduces some fixed overhead — such as memory allocation, GPU scheduling, and setup time — which only becomes worthwhile when the batch size is large enough to amortize these costs. For small to moderate batch sizes (e.g., 4 or 8), this overhead may cancel out the theoretical speedup from parallel processing.

In summary: **Batching helps more at scale**.

## Approach 3: Using ONNX

Deploying YOLOv5 models using the ONNX format with ONNX Runtime can yield better inference performance compared to running models directly in PyTorch. This is due to several key factors:

1. **Graph-Level Optimizations**: ONNX Runtime performs optimizations such as operator fusion and constant folding, reducing computational overhead and improving execution speed.
2. **Hardware Acceleration**: ONNX supports execution providers like TensorRT and CUDA, which leverage highly optimized, low-level libraries for GPU inference, often outperforming PyTorch's general-purpose engine.
3. **Reduced Overhead**: ONNX models are statically defined and require less runtime graph construction, resulting in faster initialization and lower latency, particularly for repeated inferences.
4. **Efficient Batch Processing**: ONNX Runtime scales better with larger batch sizes, improving throughput in batch inference scenarios.

In summary, ONNX enables faster and more efficient inference by optimizing the execution graph, reducing overhead, and leveraging hardware-specific acceleration, making it well-suited for high-performance deployment.

## Results

Without onnx ~ 40ms

With onnx ~ 24 ms.. The results can be observed on the notebook.