CEN/CSE 524 Machine Learning Acceleration
Lab Assignment #2

Megan Kuo, Ameya Gurjar

June 16, 2025

1 Understanding GPU A100 (Part 1)

1.1 A100 Specifications

1. **Memory Capacity, Type, and Bandwidth:**

GPU Model	Memory Capacity	Memory Type	Memory Bandwidth
NVIDIA A100 80GB PCIe	80 GB	HBM2e	1935 GB/s

Table 1: NVIDIA A100 80GB PCIe Memory Specifications

2. **TDP:** TDP stands for Thermal Design Power. The NVIDIA A100 80GB PCIe version has a TDP of 300W, meaning its cooling system must be able to dissipate up to 300 watts of heat under normal use.

1.2 A100 Compute Capabilities

1. **Number of SMs:** 108 SMs.
FP32 CUDA Cores: Each SM contains 64 FP32 CUDA cores, yielding a total of $108 \times 64 = 6912$ FP32 CUDA cores.

2. **Supported Precisions and Peak Performance, and Tensor Core v.s. CUDA Core:**

Precision	Peak Performance	Unit	Description
FP64	9.7	TFLOPS	Double Precision (CUDA cores)
FP64 (Tensor Core)	19.5	TFLOPS	Double Precision (Tensor cores)
FP32	19.5	TFLOPS	Single Precision (CUDA cores)
FP16	78	TFLOPS	Half Precision (CUDA cores)
BF16	39	TFLOPS	Brain Float 16 (CUDA cores)
TF32 (Tensor Core)	156	TFLOPS	Tensor Float 32 (Tensor cores)
FP16 (Tensor Core)	312	TFLOPS	Half Precision (Tensor cores)
BF16 (Tensor Core)	312	TFLOPS	Brain Float 16 (Tensor cores)
INT8 (Tensor Core)	624	TOPS	8-bit Integer (Tensor cores)
INT4 (Tensor Core)	1248	TOPS	4-bit Integer (Tensor cores)

Table 2: Peak Performance Metrics for the NVIDIA A100

- **CUDA Cores:** CUDA Cores are the fundamental processing units within NVIDIA's SMs. They are designed to handle general-purpose parallel computations, including FP32 and integer operations. Each CUDA Core can execute one operation per clock cycle per thread, making them highly versatile for a wide range of parallel computing tasks, such as graphics rendering, scientific simulations, and general-purpose GPU (GPGPU) workloads.
- **Tensor Cores:** Tensor Cores are specialized hardware units designed to accelerate deep learning and AI workloads. Unlike CUDA Cores, which process scalar operations, Tensor Cores are optimized for matrix operations, particularly matrix multiply-and-accumulate (MMA) operations. They can process small matrix tiles (e.g., 4x4 or 8x8) in a single clock cycle.

3. **Number of exponent and mantissa bits in each precision:**

Precision	Signed Bit	Exponent Bits	Mantissa Bits
FP64 (Double Precision)	1	11	52
FP32 (Single Precision)	1	8	23
TF32 (Tensor Float 32)	1	8	10
FP16 (Half Precision)	1	5	10
BF16 (Bfloat16)	1	8	7
INT8 / INT4	NA	NA	NA

Table 3: Exponent and Mantissa Bits for Different Precisions in NVIDIA A100

2 Training and Inference on MLP and CNN (Part 2)

```
# To find loadable module
ls -al /packages/apps/spack/21/opt/spack/linux-rocky8-zen3/gcc-12.1.0/

module load cuda-12.6.1-gcc-12.1.0
module load gcc-13.2.0-gcc-12.1.0 # we need to use updated GLIBCXX for parse_ncu_xxx.py script
```

2.1 CPU vs GPU Time Comparison

Settings: workload: [training, inference], epochs = 5, batch size = [1,64,128]

Batch Size	CPU Train (s)	CPU Infer (μ s)	GPU Train (s)	GPU Infer (μ s)	Model Arch
1	201.54	0.08	202.71	0.14	MLP
64	6.38	0.28	4.53	0.16	MLP
128	5.71	0.43	2.49	0.16	MLP
1	925.44	0.06	280.10	0.23	CNN
64	11.78	0.62	6.73	0.23	CNN
128	9.94	0.86	3.20	0.24	CNN

Table 4: Comparison of CPU and GPU Training/Inference Time

2.1.1 SimpleMLP Architecture:

```
class SimpleMLP(nn.Module):
    def __init__(self):
        super(SimpleMLP, self).__init__()

        self.fc1 = nn.Linear(784, 256)
        self.fc2 = nn.Linear(256, 128)
        self.fc3 = nn.Linear(128, 10)
        self.softmax = nn.Softmax(dim=1)

    def forward(self, x):
        x = x.view(-1, 784)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = F.relu(self.fc3(x))
        logits = self.softmax(x)
```

```
        return logits
```

2.1.2 SimpleCNN Architecture:

```
class SimpleCNN(nn.Module):
    def __init__(self):
        super(SimpleCNN, self).__init__()
        self.conv1 = nn.Conv2d(1, 1, 5) # input channels = 1, output channels = 1, kernel_size = 5
        self.pool1 = nn.MaxPool2d(2, 2) # kernel_size = 2, stride = 2
        self.conv2 = nn.Conv2d(1, 1, 5)
        self.pool2 = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(1 * 4 * 4, 100)
        self.fc2 = nn.Linear(100, 10)


    def forward(self, x):
      x = self.pool1(self.conv1(x))
      x = self.pool2(self.conv2(x))
      # x = self.pool3(self.conv3(x))
      x = torch.flatten(x, 1)
      x = F.relu(self.fc1(x))
      x = F.softmax(self.fc2(x), dim = -1)
      return x
```

2.1.3 Comments:

From the experiment, we observe that when the batch size is 1, the difference in training and inference time between the CPU and GPU is not significant. However, as the batch size increases, the GPU starts to show a clear advantage. Specifically, for both training and inference, the GPU processes batches more efficiently, resulting in lower execution time compared to the CPU (This effect is even more obvious for CNN). Additionally, while the CPU's inference time increases significantly with larger batch sizes, the GPU's inference time scales more efficiently, showing a much smaller increase.

2.1.4 Snapshots:

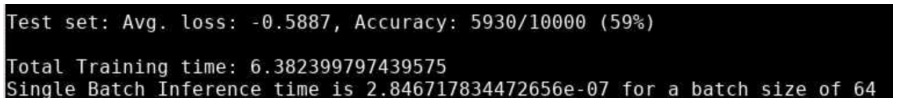
```
Test set: Avg. loss: -0.7278, Accuracy: 7277/10000 (73%)

Total Training time: 201.54271602630615
Single Batch Inference time is 8.392333984375e-08 for a batch size of 1
```

(a) Simple MLP w/ 1 batch on CPU

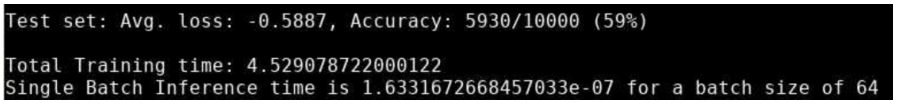
```
Test set: Avg. loss: -0.7294, Accuracy: 7297/10000 (73%)

Total Training time: 202.71081805229187
Single Batch Inference time is 1.4591217041015625e-07 for a batch size of 1
```

(b) Simple MLP w/ 1 batch on GPU

Figure 1: Comparison of Simple MLP execution on CPU and GPU with batch size 1

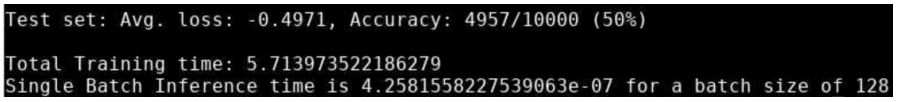


(a) Simple MLP w/ 64 batch on CPU

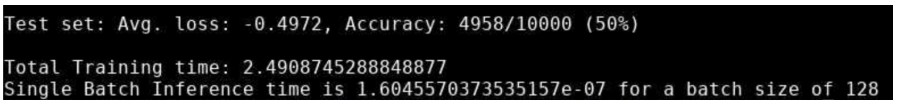


(b) Simple MLP w/ 64 batch on GPU

Figure 2: Comparison of Simple MLP execution on CPU and GPU with batch size 64

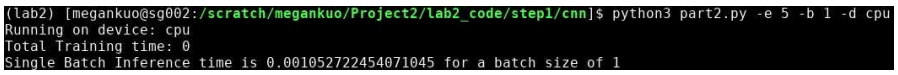


(a) Simple MLP w/ 128 batch on CPU

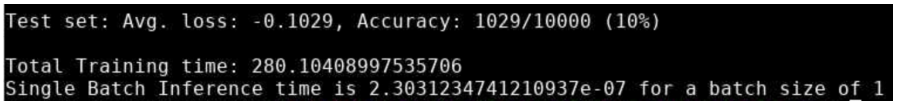


(b) Simple MLP w/ 128 batch on GPU

Figure 3: Comparison of Simple MLP execution on CPU and GPU with batch size 128

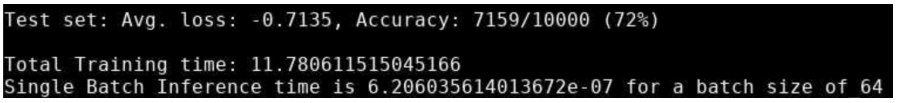


(a) Simple CNN w/ 1 batch on CPU

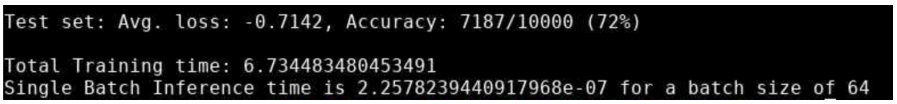


(b) Simple CNN w/ 1 batch on GPU

Figure 4: Comparison of Simple CNN execution on CPU and GPU with batch size 1

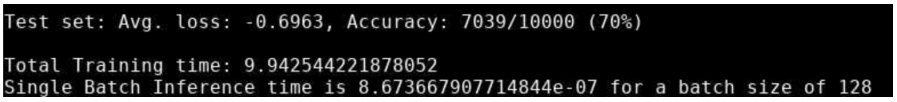


(a) Simple CNN w/ 64 batch on CPU

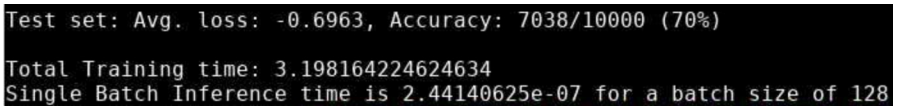


(b) Simple CNN w/ 64 batch on GPU

Figure 5: Comparison of Simple CNN execution on CPU and GPU with batch size 64



(a) Simple CNN w/ 128 batch on CPU



(b) Simple CNN w/ 128 batch on GPU

Figure 6: Comparison of Simple CNN execution on CPU and GPU with batch size 128

2.2 Profiling MLP on A100

Settings: workload: training, epochs = 1, batch size = 128
GPU profiling was done using `nsys, ncu`

Profile commands:

```
nsys profile --gpu-metrics-device=all --force-overwrite true -o mlp_1000.nsys-rep python3 part2.py -b 128
↪  -d cuda -w inference -e 1000

nsys profile --gpu-metrics-device=all --force-overwrite true -o mlp_part2.nsys-rep python3 part2.py -b 128
↪  -d cuda -w train -e 1

nsys export -t sqlite --force-overwrite true -o mlp_1000.sqlite mlp_1000.nsys-rep
nsys export -t sqlite --force-overwrite true -o mlp_part2.sqlite mlp_part2.nsys-rep

nsys stats mlp_1000.nsys-rep &> mlp_1000.stats
nsys stats mlp_part2.nsys-rep &> mlp_part2.stats
```

Commands for generating statistics:

```
sqlite3 mlp_1000.sqlite ".tables"
sqlite3 mlp_part2.sqlite ".tables"

sqlite3 mlp_part2.sqlite "
  SELECT
    MIN(start) / 1e9 AS earliest_start_sec,
    MAX(end) / 1e9   AS latest_end_sec
  FROM (
    SELECT start, end FROM CUPTI_ACTIVITY_KIND_KERNEL
  );
"
# alternate CUPTI_ACTIVITY_KIND_KERNEL with CUPTI_ACTIVITY_KIND_MEMCPY
# outputs start time (ST) and end time (ET)

python3 parse_nsys_sqlite.py --sql_db mlp_1000.sqlite --start_time $ST --end_time $ET
python3 parse_nsys_sqlite.py --sql_db mlp_part2.sqlite --start_time $ST --end_time $ET
```

2.2.1 Profile results for training with 1 epoch:

Metric	Model config (epoch/workload)	Utilization	Start time	End time
GPU	(1/train)	0.17%	9.64	19.45
DRAM bandwidth - r	(1/train)	0.09 GB/s	9.53	19.45
DRAM bandwidth - w	(1/train)	0.96 GB/s	9.53	19.45

- **Average GPU Utilization:** 0.17%
- **Total DRAM bandwidth**: 1.05 GB/s for model train with 1 epoch.

Snapshots for training with 1 epoch:

```
(lab2) [megankuo@sg041:/scratch/megankuo/Project2/lab2_code/step1/mlp]$ sqlite3 mlp_part2.sqlite "
SELECT
MIN(start) / 1e9 AS earliest_start_sec,
MAX(end) / 1e9 AS latest_end_sec
FROM (
SELECT start, end FROM CUPTI_ACTIVITY_KIND_KERNEL
);
"
9.644169424|19.447882187
(lab2) [megankuo@sg041:/scratch/megankuo/Project2/lab2_code/step1/mlp]$ python3 ../../parse_nsys_sqlite.py --sql_db mlp_part2.sqlite
--start_time 9.644 --end_time 19.448
Select data from 9644000000.0 timestamp to 19448000000.0 timestamp
GPU utilization 0.17 %
DRAM read bw 0.07 GB/s (0.00 % of Total DRAM bw)
DRAM write bw 0.93 GB/s (0.05 % of Total DRAM bw)
Total DRAM bw: 1.00 GB/s
```

(a) SimpleMLP training with 1 epoch, profile with CUDA kernel

```
(lab2) [megankuo@sg041:/scratch/megankuo/Project2/lab2_code/step1/mlp]$ sqlite3 mlp_part2.sqlite "
SELECT
MIN(start) / 1e9 AS earliest_start_sec,
MAX(end) / 1e9 AS latest_end_sec
FROM (
SELECT start, end FROM CUPTI_ACTIVITY_KIND_MEMCPY
);
"
9.5316707|19.447603756
(lab2) [megankuo@sg041:/scratch/megankuo/Project2/lab2_code/step1/mlp]$ python3 ../../parse_nsys_sqlite.py --sql_db mlp_part2.sqlite
--start_time 9.5314 --end_time 19.448
Select data from 9531400000.0 timestamp to 19448000000.0 timestamp
GPU utilization 0.17 %
DRAM read bw 0.09 GB/s (0.00 % of Total DRAM bw)
DRAM write bw 0.96 GB/s (0.05 % of Total DRAM bw)
Total DRAM bw: 1.05 GB/s
```

(b) SimpleMLP training with 1 epoch, profile with CUDA memcpy

Figure 7: SimpleMLP profile when training using 1 epoch

2.2.2 Analysis:

Table 5: CUDA GPU Kernel Summary

Time %	Total Time	Instances	Avg	Med)	Min	Max	Name
20.5	17880191	1644	10876.0	12192.0	7328	15072	ampere_sgemm_32x32_sliced1x4_tn
16.1	14104665	937	15053.0	15584.0	14208	16128	multi_tensor_apply_kernel
8.4	7346991	939	7824.3	7808.0	7137	8832	ampere_sgemm_32x32_sliced1x4_nt
7.9	6900418	938	7356.5	7536.0	6112	8096	reduce_kernel
6.4	5551353	468	11861.9	11871.0	11775	12000	multi_tensor_apply_kernel
6.0	5257664	469	11210.4	11232.0	9728	11456	reduce_kernel
4.8	4226481	468	9030.9	9024.0	8960	9120	ampere_sgemm_64x32_sliced1x4_nt
4.3	3759001	1647	2282.3	2304.0	2111	2624	vectorized_elementwise_kernel
4.2	3660680	469	7805.3	7808.0	7775	8128	ampere_sgemm_32x32_sliced1x4_nn
3.7	3233758	1407	2298.3	2304.0	2175	2432	vectorized_elementwise_kernel
2.9	2549612	469	5436.3	5440.0	5407	5473	ampere_sgemm_128x32_nn
2.8	2421968	548	4419.6	4352.5	2751	5184	nll_loss_forward_reduce_cuda_kernel
2.4	2086008	549	3799.7	3809.0	3615	3872	splitKreduce_kernel
2.0	1738506	938	1853.4	1856.0	1759	1921	vectorized_elementwise_kernel
1.9	1641810	469	3500.7	3520.0	3137	3807	nll_loss_backward_reduce_cuda_kernel
1.7	1456379	549	2652.8	2688.0	2528	2721	softmax_warp_forward
1.3	1103898	469	2353.7	2368.0	2240	2464	softmax_warp_backward

Continued on next page

Time %	Total Time	Instances	Avg	Med	Min	Max	Name
1.2	1009715	469	2152.9	2144.0	2111	2272	vectorized_elementwise_kernel
0.6	499647	79	6324.6	6336.0	5824	6496	reduce_kernel
0.4	365952	79	4632.3	4609.0	4256	4768	reduce_kernel
0.2	205793	79	2605.0	2592.0	2528	2720	unrolled_elementwise_kernel
0.2	166782	79	2111.2	2080.0	2048	2209	vectorized_elementwise_kernel
0.2	164256	78	2105.8	2080.0	2016	2177	vectorized_elementwise_kernel
0.0	11488	1	11488.0	11488.0	11488	11488	ampere_sgemm_32x128_tn
0.0	9184	1	9184.0	9184.0	9184	9184	ampere_sgemm_64x32_sliced1x4_tn
0.0	6368	1	6368.0	6368.0	6368	6368	gemmSN_TN_kernel
0.0	2464	1	2464.0	2464.0	2464	2464	unrolled_elementwise_kernel
0.0	2048	1	2048.0	2048.0	2048	2048	vectorized_elementwise_kernel

Table 5: Profile result of SimpleMLP with less than 3 hidden layers

2.2.3 Profile results for Inference with 1000 times:

Metric	Model config (epoch/workload)	Utilization	Start time	End time
GPU	(1000/infer)	0.04%	9.62	9.67
DRAM bandwidth - r	(1000/infer)	2.47 GB/s	9.51	9.54
DRAM bandwidth - w	(1000/infer)	5.15 GB/s	9.51	9.54

Table 6: Exponent and Mantissa Bits for Different Precisions in NVIDIA A100

- **Average GPU Utilization:** 0.04%
- **Total DRAM bandwidth**: 7.62 GB/s for model inference with 1000 epochs.

Snapshots for inference with 1000 epochs:

```
(lab2) [megankuo@sg041:/scratch/megankuo/Project2/lab2_code/step1/mlp]$ sqlite3 mlp_1000_inf.sqlite "
> SELECT
> MIN(start) / 1e9 AS earliest_start_sec,
> MAX(end) / 1e9 AS latest_end_sec
> FROM (
> SELECT start, end FROM CUPTI_ACTIVITY_KIND_KERNEL
> );
> "
9.622852869|9.666457045
(lab2) [megankuo@sg041:/scratch/megankuo/Project2/lab2_code/step1/mlp]$ python3 ../../parse_nsys_sqlite.py --sql_db mlp_1000_inf.sqli
te --start_time 9.6228 --end_time 9.667
Select data from 9622800000.0 timestamp to 9667000000.0 timestamp
GPU utilization 0.04 %
DRAM read bw 1.24 GB/s (0.06 % of Total DRAM bw)
DRAM write bw 10.62 GB/s (0.55 % of Total DRAM bw)
Total DRAM bw: 11.86 GB/s
```

(a) SimpleMLP inference with 1000 epochs, profile with CUDA kernel

```
(lab2) [megankuo@sg041:/scratch/megankuo/Project2/lab2_code/step1/mlp]$ sqlite3 mlp_1000_inf.sqlite "
SELECT
MIN(start) / 1e9 AS earliest_start_sec,
MAX(end) / 1e9 AS latest_end_sec
FROM (
SELECT start, end FROM CUPTI_ACTIVITY_KIND_MEMCPY
);
"
9.514966022|9.536404576
(lab2) [megankuo@sg041:/scratch/megankuo/Project2/lab2_code/step1/mlp]$ python3 ../../parse_nsys_sqlite.py --sql_db mlp_1000_inf.sqli
te --start_time 9.5149 --end_time 9.5361
Select data from 9514900000.0 timestamp to 9536100000.0 timestamp
GPU utilization 0.00 %
DRAM read bw 2.47 GB/s (0.13 % of Total DRAM bw)
DRAM write bw 5.15 GB/s (0.27 % of Total DRAM bw)
Total DRAM bw: 7.62 GB/s
```

(b) SimpleMLP inference with 1000 epochs, profile with CUDA memcpy

Figure 8: SimpleMLP profile when inferencing using 1000 epochs

2.2.4 Analysis:

Table 7: CUDA GPU Kernel Summary

Time %	Total Time	Instances	Avg	Med)	Min	Max	Name
72.3	35808	3	11936.0	12736.0	8032	15040	ampere_sgemm_32x32_sliced1x4_tn
14.2	7040	3	2346.7	2304.0	2144	2592	vectorized_elementwise_kernel_launch_clamp_scalar
8.0	3967	1	3967.0	3967.0	3967	3967	splitKreduce_kernel
5.5	2720	1	2720.0	2720.0	2720	2720	softmax_warp_forward

Table 7: Profile result of SimpleMLP inferencing 1000 times

2.2.5 Profile results comparison:

- **Average GPU and DRAM bandwidth Utilization:**

Metric	Model config (epoch/workload)	Utilization	Start time	End time
GPU	(1/train)	0.17%	9.64	19.45
GPU	(1000/infer)	0.04%	9.62	9.67
DRAM bandwidth - r	(1/train)	0.09 GB/s	9.53	19.45
DRAM bandwidth - w	(1/train)	0.96 GB/s	9.53	19.45
DRAM bandwidth - r	(1000/infer)	2.47 GB/s	9.51	9.54
DRAM bandwidth - w	(1000/infer)	5.15 GB/s	9.51	9.54

Table 8: Profile result of training-1 and inference-1000

- **Total DRAM bandwidth**: 1.05 GB/s for model train with 1 epoch, 7.62 GB/s for model inference with 1000 epochs.

2.3 Kernel Name Analysis

2.3.1 - Ampere SGEMM Kernel

```
ampere_sgemm_32x32_sliced1x4_tn
```

- **ampere** - Indicates the kernel is optimized for NVIDIA's Ampere architecture.
- **sgemm** - Refers to **Single-precision General Matrix Multiplication (SGEMM)**.
- **32x32** - Likely represents the **tile size**, meaning that each thread block processes a 32×32 sub-matrix.
- **sliced1x4** - Suggests a **partitioning strategy** for dividing work across the GPU.
- **tn** - Indicates the layout of the matrices:
  - **T** (Transpose) for matrix A
  - **N** (Non-Transpose) for matrix B

Use Case: This kernel is used in **cuBLAS** for accelerating matrix multiplications in deep learning and scientific computing.

2.3.2 - Softmax Warp Forward Kernel

```
void <unnamed>::softmax_warp_forward<float, float, float, 4, 0, 0> (T2 *, const T1 *,
    int, int, int, const bool *, int, bool)
```

- **softmax_warp_forward** - Computes the **softmax activation function** using **warp-level parallelism**.
- **Template Parameters:**
  - **float, float, float** - Likely represents **input type, intermediate type, and output type**.
  - **4** - May indicate that each warp processes **4 elements per thread**.
  - **0, 0** - Additional tuning parameters (e.g., memory alignment or shared memory usage).
- **Function Arguments:**
  - T_2^* - Output tensor pointer.
  - T_1^* - Input tensor pointer.
  - Several `int` and `bool` values, likely representing **batch size, mask flags, and activation properties**.

2.3.3 - Vectorized Elementwise Kernel

```
void at::vectorized_elementwise_kernel<4, at::<unnamed>::launch_clamp_scalar (at::
    TensorIteratorBase &, c10::Scalar, c10::Scalar, at::ClampLimits)::[lambda() (instance
     1)]::operator ()() const::[lambda() (instance 7)]::operator ()() const::[lambda(
    float) (instance 1)], std::array<char *, 2>>(int, T2, T3)
```

- **vectorized_elementwise_kernel** - Executes an **elementwise operation** across a tensor using SIMD for efficiency.
- **Template Parameter 4** - Likely processes **4 elements per loop iteration** to optimize vectorized computation.
- **launch_clamp_scalar** - Suggests this kernel is responsible for **clamping** values within a given range.
- $std :: array < char*, 2 >$ - Likely holds two pointers for **input and output arrays**.

Use Case: Used in **PyTorch** for `torch.clamp()` and other elementwise tensor operations.

2.3.4 - Split-K Reduction Kernel

```
void cublasLt::splitKreduce_kernel<32, 16, int, float, float, float, 0,
    float, float, float, 1, 1, 0>(cublasLt::cublasSplitKParams<T6>,
    const T4 *, const T9 *, T8 *, T5 *, const T6 *, const T6 *,
    const T10 *, const T4 *, T10 *, void *, long, T6 *, int *,
    T6 *, const T6 *, const T6 *, const T6 *, const T6 *)
```

- **cublasLt** - Belongs to **cuBLASLt**, a low-level CUDA library for optimized GEMM computations.
- **splitKreduce_kernel** - Implements **Split-K Reduction**, a technique for parallel matrix multiplication.
- **Template Parameters:**
  - $32, 16$ - Likely refers to **tile sizes** in the matrix multiplication (each block handles a 32×16 sub-matrix).
  - Several `float` and `int` parameters representing **data types and tuning factors**.
- **Function Arguments:**
  - **cuBLAS Split-K Parameters** - Structure defining how work is distributed across multiple GPU blocks.
  - **Multiple Float Pointers** - Representing matrices A, B, C and scaling factors α and β.

2.3.5 Comments:

In our profiling, we noticed that all the *sm__ops_path_tensor...* metrics reported 0; while the *sm__sass_thread_inst_executed_ops_fadd_fmul_ffma_pred_on* metric showed non-zero values. This typically indicates that the workload isn't utilizing tensor core-specific operations—hence, the tensor path metrics remain at zero, and the operations are being executed via standard instructions.

2.4 Raw Metrics, Roofline, Units

In our profiling data, we extract several key metrics [1]:

- **Execution Time:**

$$T = \frac{\texttt{sm__cycles\_elapsed.avg}}{\texttt{sm__cycles\_elapsed.avg.per\_second}},$$

  where:
  - `sm__cycles_elapsed.avg` is the average number of cycles elapsed during execution.
  - `sm__cycles_elapsed.avg.per_second` is the number of cycles per second (i.e. the GPU's clock rate in Hz).

  This gives the execution time T in seconds.

- **Total Floating-Point Operations (FLOPs):** The total FLOPs are computed from two sources:

[1] reference: https://www.nersc.gov/assets/Uploads/RooflineHack-2020-mechanism-v2.pdf

1. **CUDA Core Operations:**

$$\begin{aligned}\text{FLOPs (CUDA)} = &\texttt{sm\_\_sass\_thread\_inst\_executed\_op\_xadd\_pred\_on.sum} \\ &+ 2 \times \texttt{sm\_\_sass\_thread\_inst\_executed\_op\_xfma\_pred\_on.sum} \\ &+ \texttt{sm\_\_sass\_thread\_inst\_executed\_op\_xmul\_pred\_on.sum}. \end{aligned} \tag{1}$$

Note1: Replace `x` with `d` for double precision (DP), `f` for single precision (SP), or `h` for half precision (HP) as appropriate. *Note2:* We are considering ffma as we are using single precision.

2. **Tensor Core Operations:**

$$\text{FLOPs (Tensor)} = \texttt{sm\_\_inst\_executed\_pipe\_tensor.sum} \times 512.$$

Each tensor-core instruction is assumed to perform 512 floating-point operations.

Thus, the **Total FLOPs** is:

$$\text{Total FLOPs} = \text{FLOPs (CUDA)} + \text{FLOPs (Tensor)}.$$

- **Memory Traffic (Bytes):** We use the metric:

$$\texttt{dram\_\_bytes\_read.sum + dram\_\_bytes\_write.sum}$$

which gives the total DRAM bytes accessed.

2.4.1 Deriving Roofline Parameters

Once the above quantities are known, we compute:

Performance (FLOPs/s) The performance is defined as:

$$\text{Performance (FLOPs/s)} = \frac{\text{Total FLOPs}}{T}.$$

Here, T is the execution time in seconds computed from the cycle counts.

Arithmetic Intensity (Ops/Byte) Arithmetic intensity is the ratio of the total number of operations to the total memory bytes accessed:

$$\text{Arithmetic Intensity (Ops/Byte)} = \frac{\text{Total FLOPs}}{\text{Memory Traffic}}.$$

2.4.2 Calculation

- `sm__cycles_elapsed.avg` = 72,581 cycles,
- `sm__cycles_elapsed.avg.per_second` = 10,083,762,361 cycles/s,
- `sm__sass_thread_inst_executed_op_ffma_pred_on_x2.sum` = 30,968,064 FLOPS (fused mul and add together, hence x2 times 2 operations for us),
- `sm__sass_thread_inst_executed_op_fadd_pred_on.sum` = 763,904 FLOPS,
- `sm__sass_thread_inst_executed_op_fmul_pred_on.sum` = 772,096 FLOPS,
- `sm__inst_executed_pipe_tensor.sum` = 0 cycles (no tensor core is used),
- `dram__bytes_read.sum` = 2,481,024 bytes.
- `dram__bytes_write.sum` = 0 bytes.

Note: We can find the sum of fadd, ffma, and fmul, is also given by metric : (2)

$$\texttt{sm\_\_sass\_thread\_inst\_executed\_op\_fadd\_fmul\_ffma\_pred\_on.sum}$$

Compute Execution Time

$$T = \frac{72{,}581}{10{,}083{,}762{,}361} \approx 7.2 \times 10^{-6} \text{ seconds}.$$

Compute Total FLOPs For CUDA cores:

$$\text{FLOPs (CUDA)} = 32,504,064.$$

For Tensor cores:

$$\text{FLOPs (Tensor)} = 0.$$

Thus,

$$\text{Total FLOPs} = 32,504,064.$$

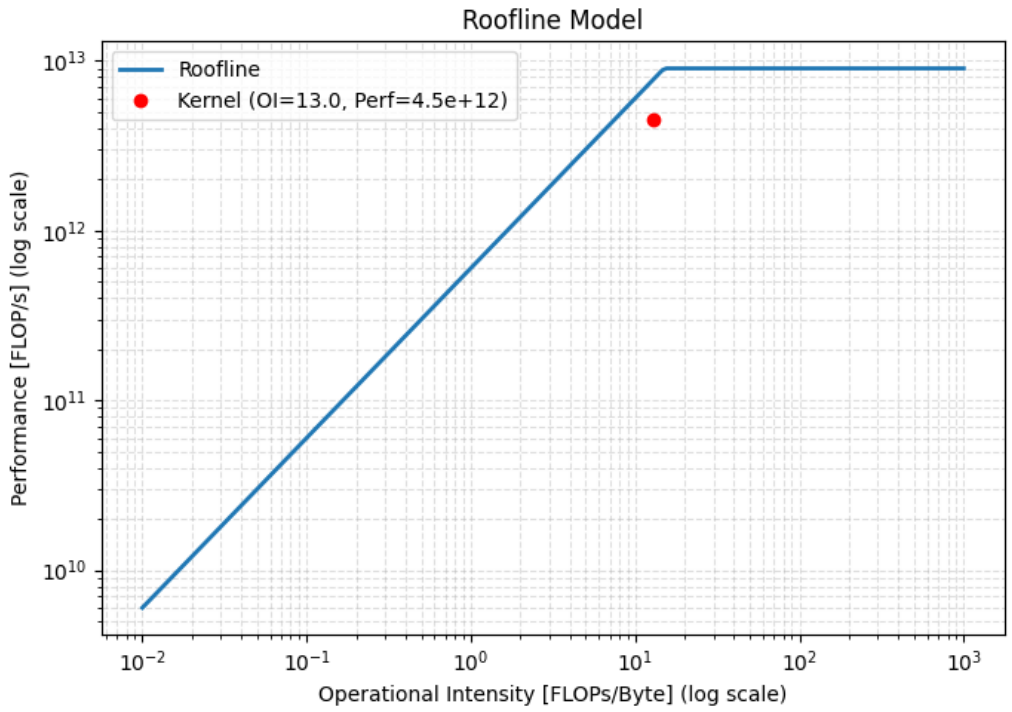
Compute Performance

$$\text{Performance (FLOPs/s)} = \frac{32,504,064}{7.2 \times 10^{-6}} = 4.5 \times 10^{12}$$

Compute Arithmetic Intensity

$$\text{Arithmetic Intensity (Ops/Byte)} = \frac{32,504,064}{2{,}481{,}024} \approx 13$$

Roofline plot



(a) Roofline

2.4.3 Conclusion

- The performance in 4.5 TFLOPs/s.
- The arithmetic intensity 13 bytes/s.

We observe a performance of 4.5 TFLOP/s and an arithmetic intensity of 13 in our batch inference experiment, and we attribute these values to the characteristics of our MLP kernel. Specifically, the high throughput stems from the fact that our kernel is short-lived, yet it executes with high concurrency, maximizing the utilization of available GPU cores within its brief execution window. We think the use of batch inference significantly contributes to the arithmetic intensity. By processing multiple inputs simultaneously, the kernel can efficiently reuse weights and intermediate activations across multiple data elements within the batch. This reduces the need for repeated memory accesses from DRAM, thereby increasing the ratio of FLOPs to bytes transferred. The high arithmetic intensity of 13 suggests that a substantial portion of the computation is performed using data that resides in registers or cache, minimizing costly memory transactions. Additionally, the short-lived nature of the kernel enhances its ability to achieve peak-like performance. Since overheads such as kernel launch latency and memory transactions are amortized over a high number of computations, the kernel achieves a high sustained FLOP rate despite its short duration. The combination of efficient data reuse, high concurrency, and short execution time leads to a scenario where the kernel appears to operate close to the peak performance of the GPU. And we conclude from the roofline model that our inference is indeed memory-bound computation.

Tables:

Table 9: SASS Thread Instruction Execution Metrics

Metric	SGEMM	Softmax Warp	Vectorized	Split-K Reduce	Total
sm_sass_thread_inst_executed_ops_fadd_fmul_ffma_pred_on.avg	298552.90	286.81	0	2123.85	300963.56
sm_sass_thread_inst_executed_ops_fadd_fmul_ffma_pred_on.max	751616	3872	0	3584	759072
sm_sass_thread_inst_executed_ops_fadd_fmul_ffma_pred_on.min	171008	0	0	0	171008
sm_sass_thread_inst_executed_ops_fadd_fmul_ffma_pred_on.sum	32243712	30976	0	229376	32504064
Grand Total	33464888.9	35134.81	0	235083.85	33735107.56

Table 10: DRAM Performance Metrics

Metric	SGEMM	Softmax Warp	Vectorized	Split-K Reduce	Total
Total DRAM Elapsed Cycles	2,496,640	219,136	613,120	307,840	3,636,736
dram__bytes_write.sum	0	0	0	0	0
dram__bytes_read.sum	367.43	11.65	216.70	665.86	1,261.64
DRAM Throughput	5.69	0.17	3.27	6.76	15.89
DRAM Frequency	4.77	1.59	4.71	1.57	12.64
Grand Total	2,497,017.89	219,149.41	613,344.68	308,514.19	3,638,026.17

Snapshots:

Kernel Name	ampere_sgemm	void <unnamed	void at::'	void cublasLt	Grand Total
Metric Name	Metric Value (Sum)				
sm__sass_thread_inst_executed_ops_fadd_fmul_ffma_pred_on.sum	32243712	30976	0	229376	**32504064**
sm__sass_thread_inst_executed_ops_fadd_fmul_ffma_pred_on.min	171008	0	0	0	**171008**
sm__sass_thread_inst_executed_ops_fadd_fmul_ffma_pred_on.max	751616	3872	0	3584	**759072**
sm__sass_thread_inst_executed_ops_fadd_fmul_ffma_pred_on.avg	298552.90	286.81	0	2123.85	**300963.56**
Grand Total	**33464888.9**	**35134.81**	**0**	**235083.85**	**33735107.56**

(a) thread instruction execution metrics

Kernel Name	ampere_sgemm	void <unnamed	void at::'	void cublasLt	Grand Total
Metric Name	Metric Value (Sum)				
Total DRAM Elapsed Cycles	2496640	219136	613120	307840	**3636736**
dram__bytes_write.sum	0	0	0	0	**0**
dram__bytes_read.sum	367.43	11.65	216.70	665.86	**1261.64**
DRAM Throughput	5.69	0.17	3.27	6.76	**15.89**
DRAM Frequency	4.77	1.59	4.71	1.57	**12.64**
Grand Total	**2497017.89**	**219149.41**	**613344.(**	**308514.19**	**3638026.17**

(b) dram__bytes_read_write.sum

Figure 10: SimpleMLP profile when inferencing using 1000 epochs

2.4.4 Kernel performance metrics - ncu

```
ncu -f --set full --metrics \
sm__ops_path_tensor_src_bf16_dst_fp32,sm__ops_path_tensor_src_bf16_dst_fp32_sparsity_off, \
sm__ops_path_tensor_src_bf16_dst_fp32_sparsity_on,sm__ops_path_tensor_src_fp16_dst_fp16,\
sm__ops_path_tensor_src_fp16_dst_fp16_sparsity_off,sm__ops_path_tensor_src_fp16_dst_fp16_sparsity_on,\
sm__ops_path_tensor_src_fp16_dst_fp32,sm__ops_path_tensor_src_fp16_dst_fp32_sparsity_off,\
sm__ops_path_tensor_src_fp16_dst_fp32_sparsity_on,sm__ops_path_tensor_src_fp64,\
sm__ops_path_tensor_src_int1,sm__ops_path_tensor_src_int4,\
sm__ops_path_tensor_src_int4_sparsity_off,sm__ops_path_tensor_src_int4_sparsity_on,\
sm__ops_path_tensor_src_int8,sm__ops_path_tensor_src_int8_sparsity_off,\
sm__ops_path_tensor_src_int8_sparsity_on,sm__ops_path_tensor_src_tf32_dst_fp32,\
sm__ops_path_tensor_src_tf32_dst_fp32_sparsity_off,sm__ops_path_tensor_src_tf32_dst_fp32_sparsity_on,\
sm__sass_thread_inst_executed_ops_fadd_fmul_ffma_pred_on,smsp__sass_thread_inst_executed_op_fadd_pred_on.sum,\
smsp__sass_thread_inst_executed_op_fmul_pred_on.sum,smsp__sass_thread_inst_executed_op_ffma_pred_on.sum,\
dram__bytes_read.sum,dram__bytes_write.sum,\
sm__cycles_elapsed.avg --csv -o output_profile python part2.py -e 1 -d cuda -w inference -b 128

ncu --import output_profile.ncu-rep --csv > output_profile.csv

# for debug
ncu --metrics dram__bytes_write.sum,l1tex__t_bytes.sum,lts__t_bytes.sum,smsp__sass_data_bytes.sum -o debug
↪  python part2.py -e 1 -d cuda -w inference -b 128
==PROF== Connected to process 3799129 (/usr/bin/python3.11)
Running on device: cuda
Total Training time: 0
==PROF== Profiling "ampere_sgemm_32x32_sliced1x4_tn" - 0: 0%....50%....100% - 1 pass
==PROF== Profiling "splitKreduce_kernel" - 1: 0%....50%....100% - 1 pass
==PROF== Profiling "vectorized_elementwise_kernel" - 2: 0%....50%....100% - 1 pass
==PROF== Profiling "ampere_sgemm_32x32_sliced1x4_tn" - 3: 0%....50%....100% - 1 pass
==PROF== Profiling "vectorized_elementwise_kernel" - 4: 0%....50%....100% - 1 pass
==PROF== Profiling "ampere_sgemm_32x32_sliced1x4_tn" - 5: 0%....50%....100% - 1 pass
==PROF== Profiling "vectorized_elementwise_kernel" - 6: 0%....50%....100% - 1 pass
==PROF== Profiling "softmax_warp_forward" - 7: 0%....50%....100% - 1 pass
Single Batch Inference time is 0.0023584647178649903 for a batch size of 128
==PROF== Disconnected from process 3799129
```

3 Training and Inference on ResNet-18 (Part 3)

3.1 Training and Inference Times

Batch Size	Training Time (GPU)	Inference Time (GPU)
1	6283.2402 sec	0.0053 sec
64	625.0386 sec	0.0093 sec
128	614.9121 sec	0.0144 sec

Table 11: ResNet-18 Training and Inference Time

3.2 Inference Times on GPU and CPU

Batch Size	Inference Time (CPU)	Inference Time (GPU)
1	0.02 sec	0.0053 sec
64	0.95 sec	0.0093 sec
128	1.94 sec	0.0144 sec

Table 12: ResNet-18 Inference Time on CPU and GPU

3.2.1 Compare inference time on CPU vs. A100 GPU for Resnet18:

The inference time with a GPU for a batch size of 1 is approximately 3.77 times faster than the inference time with a CPU; for a batch size of 64, it is 102.15 times faster; and for a batch size of 128, it is 134.72 times faster.

Snapshots:

```
(myenv) [agurjar2@sg003:/scratch/agurjar2/lab2_code]$ python part3_resnet18_inf.py
Using cuda for inference
Using cache found in /home/agurjar2/.cache/torch/hub/NVIDIA_DeepLearningExamples_torchhub
Inference time for batchsize 1 of device cuda: 0.005322594642639161
```

(a) Resnet18 inference with batch size 1

```
(myenv) [agurjar2@sg003:/scratch/agurjar2/lab2_code]$ python part3_resnet18_inf.py
Using cuda for inference
Using cache found in /home/agurjar2/.cache/torch/hub/NVIDIA_DeepLearningExamples_torchhub
Inference time for batchsize 64 of device cuda: 0.009327802658081055
```

(b) Resnet18 inference with batch size 64

```
(myenv) [agurjar2@sg003:/scratch/agurjar2/lab2_code]$ python part3_resnet18_inf.py
Using cuda for inference
Using cache found in /home/agurjar2/.cache/torch/hub/NVIDIA_DeepLearningExamples_torchhub
Inference time for batchsize 128 of device cuda: 0.014488756656646729
```

(c) Resnet18 inference with batch size 128

```
Train Epoch: 1 [832100/832571] Loss: 7.548227
Train Epoch: 1 [832200/832571] Loss: 7.296202
Train Epoch: 1 [832300/832571] Loss: 6.049806
Train Epoch: 1 [832400/832571] Loss: 5.809541
Train Epoch: 1 [832500/832571] Loss: 6.544328
Epoch: 1, Train Loss: 6.766327%
Total training time for 1 epoch for 1 batch size and device cuda: 6283.24022436142
```

(d) Resnet18 training with batch size 1

```
Train Epoch: 1 [806400/832571] Loss: 4.250562
Train Epoch: 1 [812800/832571] Loss: 4.398875
Train Epoch: 1 [819200/832571] Loss: 4.153772
Train Epoch: 1 [825600/832571] Loss: 4.263725
Train Epoch: 1 [832000/832571] Loss: 4.254420
Epoch: 1, Train Loss: 4.934634%
Total training time for 1 epoch for 64 batch size and device cuda: 625.0386245250702
```

(e) Resnet18 training with batch size 64

```
Train Epoch: 1 [768000/832571] Loss: 2.641210
Train Epoch: 1 [780800/832571] Loss: 3.068695
Train Epoch: 1 [793600/832571] Loss: 3.098099
Train Epoch: 1 [806400/832571] Loss: 2.949887
Train Epoch: 1 [819200/832571] Loss: 2.574875
Train Epoch: 1 [832000/832571] Loss: 2.635711
Epoch: 1, Train Loss: 3.286167%
Total training time for 1 epoch for 128 batch size and device cuda: 614.9121587276459
```

(f) Resnet18 training with batch size 128

3.3 Profile training on Resnet18 on A100

3.3.1 GPU utilization

The GPU utilization for Resnet18 training on the A100 GPU is 14.41%. The GPU utilization is low because the SMs are idle for a significant period. From the third snapshot, we can see that no kernels are being issued during this time, which is why the SMs are inactive.

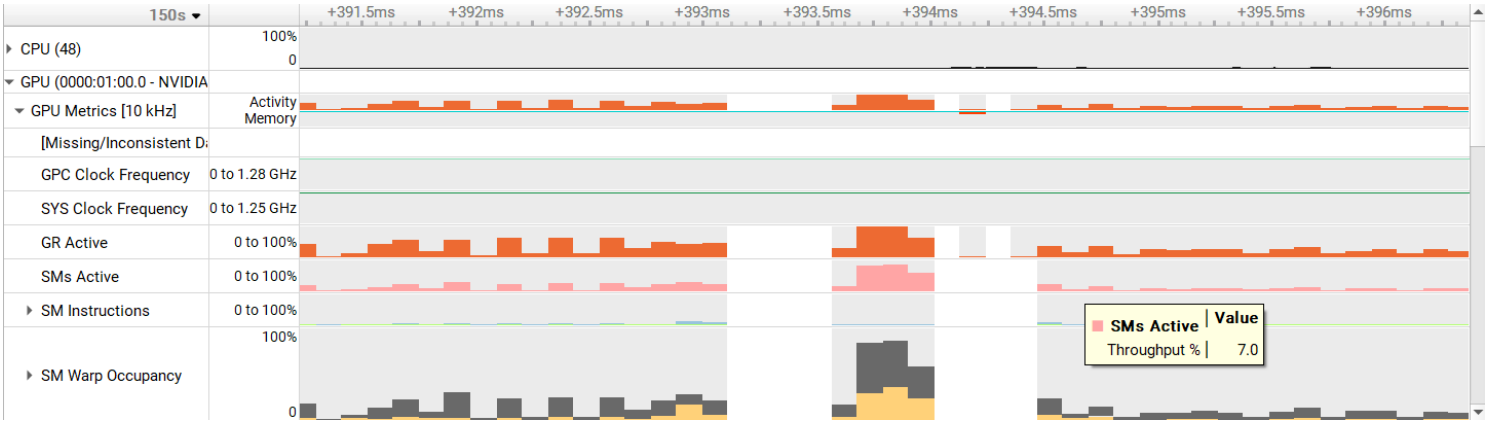
3.3.2 DRAM bandwidth utilization

The read bandwidth of the DRAM is 47.98 GB/s (2.48% of the total DRAM bandwidth), and the write bandwidth of the DRAM is 34.97 GB/s (1.81% of the total DRAM bandwidth), and the total DRAM bandwidth is 82.95 GB/s.

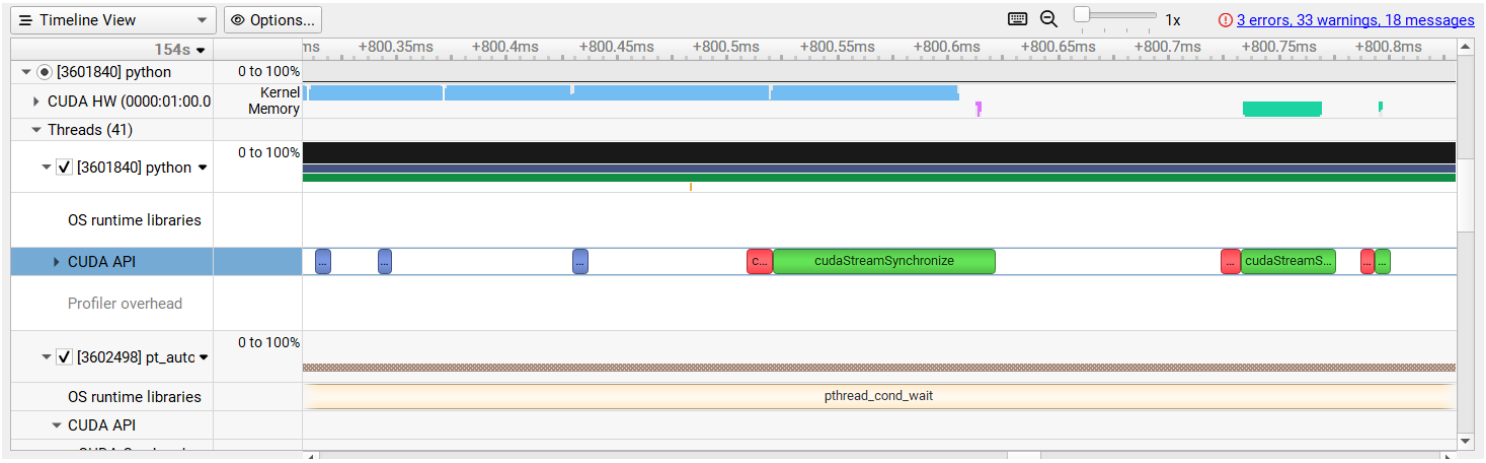
Snapshots:

```
(myenv) [agurjar2@sg019:/scratch/agurjar2/lab2_code]$ python parse_nsys_sqlite.py --sql_db part3_train.sqlite --start_time 130 --
end_time 240
Select data from 130000000000.0 timestamp to 240000000000.0 timestamp
GPU utilization 14.41 %
DRAM read bw 47.98 GB/s (2.48 % of Total DRAM bw)
DRAM write bw 34.97 GB/s (1.81 % of Total DRAM bw)
Total DRAM bw: 82.95 GB/s
```

(a) Resnet training profile



(b) Resnet SMs active



(c) Resnet Kernels active

- **Top 3 fprop kernels:**
  - `sm80_xmma_fprop_implicit_gemm_indexed_tf32f32_tf32f32_f32_nhwckrsc_nchw_tilesize64x32x64_stage5_war...`
  - `sm80_xmma_fprop_implicit_gemm_tf32f32_tf32f32_f32_nhwckrsc_nchw_tilesize64x32x64_stage5_warpsize2x2...`
  - `sm80_xmma_fprop_implicit_gemm_indexed_wo_smem_tf32f32_tf32f32_f32_nhwckrsc_nhwc_tilesize128x32x16_s...`

- **Top 3 dgrad kernels:**
  - `void cudnn::detail::dgrad2d_alg1_1<float, (int)0, (int)6, (int)6, (int)5, (int)4, (int)4, (bool)0, ...>`
  - `void cutlass__5x_cudnn::Kernel<cutlass_tensorop_s1688dgrad_optimized_tf32_64x64_32x5_nhwc_unity_str...>`
  - `sm80_xmma_dgrad_implicit_gemm_tf32f32_tf32f32_f32_nhwckrsc_nhwc_tilesize64x32x64_stage5_warpsize2x2...`
- **Top 3 wgrad kernels:**
  - `void cudnn::cnn::wgrad_alg1_engine<float, float, (int)128, (int)5, (int)5, (int)3, (int)3, (int)3, ...>`
  - `sm80_xmma_wgrad_implicit_gemm_indexed_tf32f32_tf32f32_f32_nhwckrsc_nhwc_tilesize64x32x64_stage5_war...`
  - `void cutlass__5x_cudnn::Kernel<cutlass_tensorop_s1688wgrad_optimized_tf32_128x64_16x6_nhwc_align4>(...>`

3.4 Profile inference on Resnet18 on A100

3.4.1 GPU utilization

The GPU utilization for Resnet18 inference on the A100 GPU is 3.23%. The GPU utilization is low because the SMs are idle for a significant period due to the small Resnet model.

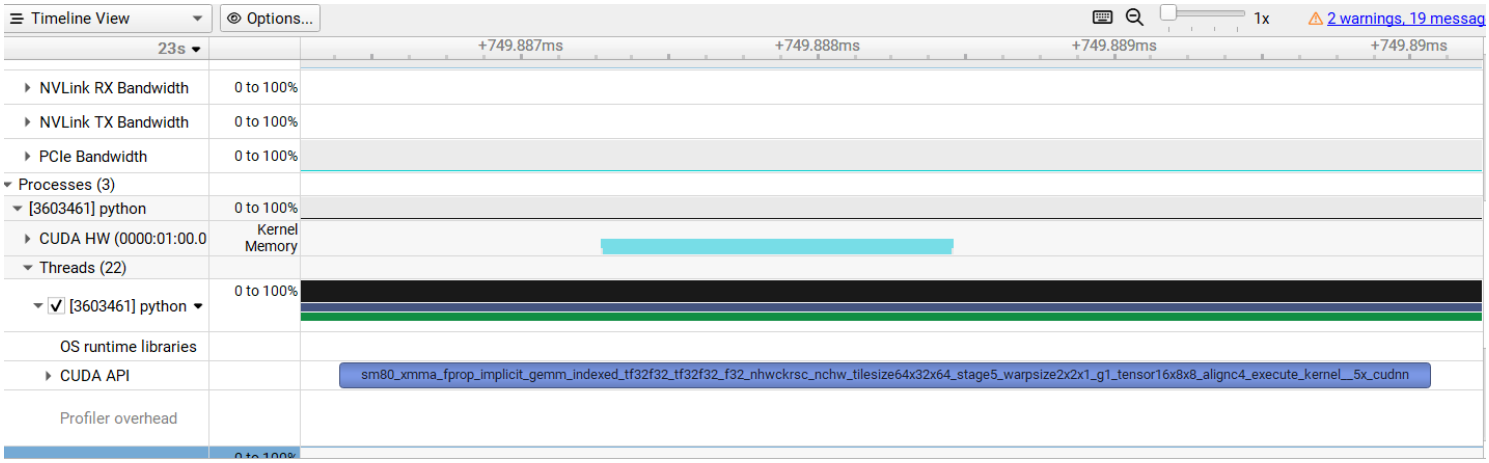
3.4.2 DRAM bandwidth utilization

The read bandwidth of the DRAM is 8.02 GB/s (0.41% of the total DRAM bandwidth), and the write bandwidth of the DRAM is 7.58 GB/s (0.39% of the total DRAM bandwidth), and the total DRAM bandwidth is 15.6 GB/s.

Snapshots:

```
(myenv) [agurjar2@sg009:/scratch/agurjar2/lab2_code]$ python parse_nsys_sqlite.py --sql_db part3_inf.sqlite --start_time 23 --end
_time 24
Select data from 23000000000.0 timestamp to 24000000000.0 timestamp
GPU utilization 4.23 %
DRAM read bw 8.02 GB/s (0.41 % of Total DRAM bw)
DRAM write bw 7.58 GB/s (0.39 % of Total DRAM bw)
Total DRAM bw: 15.60 GB/s
```

(a) Resnet training profile



(b) Resnet first iteration kernel

3.4.3 DRAM read and write bytes

DRAM read bytes = 137.21 MB DRAM write bytes = 242.43 KB

Snapshots:

ID	dram__bytes_read.sum [Mb (137.21 Mbyte)	dram__bytes_read.sum.pct_	dram__bytes_read.sum.per_
0	**0.61**	**4.52**	**91.77**
1	0.05	0.55	11.17
2	0.66	1.57	31.85
3	3.22	23.65	479.81
4	3.22	8.84	180.11
5	3.22	26.71	543.29

(a) DRAM Read Sum

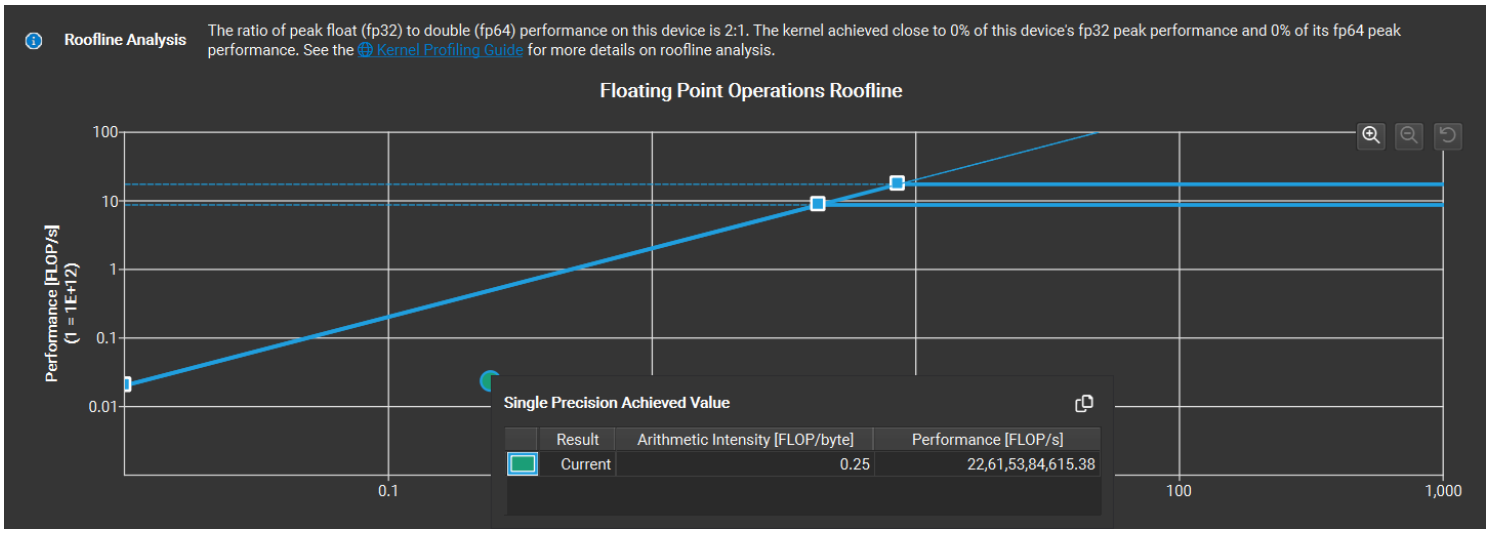
ID	dram__bytes_write.sum [Kb (242.43 Kbyte)	dram__bytes_write.sum.pct	dram__bytes_write.sum.per
0	**0**	**0**	**0**
1	0	0	0
2	0	0	0
3	0	0	0
4	0	0	0

(b) DRAM Write Sum

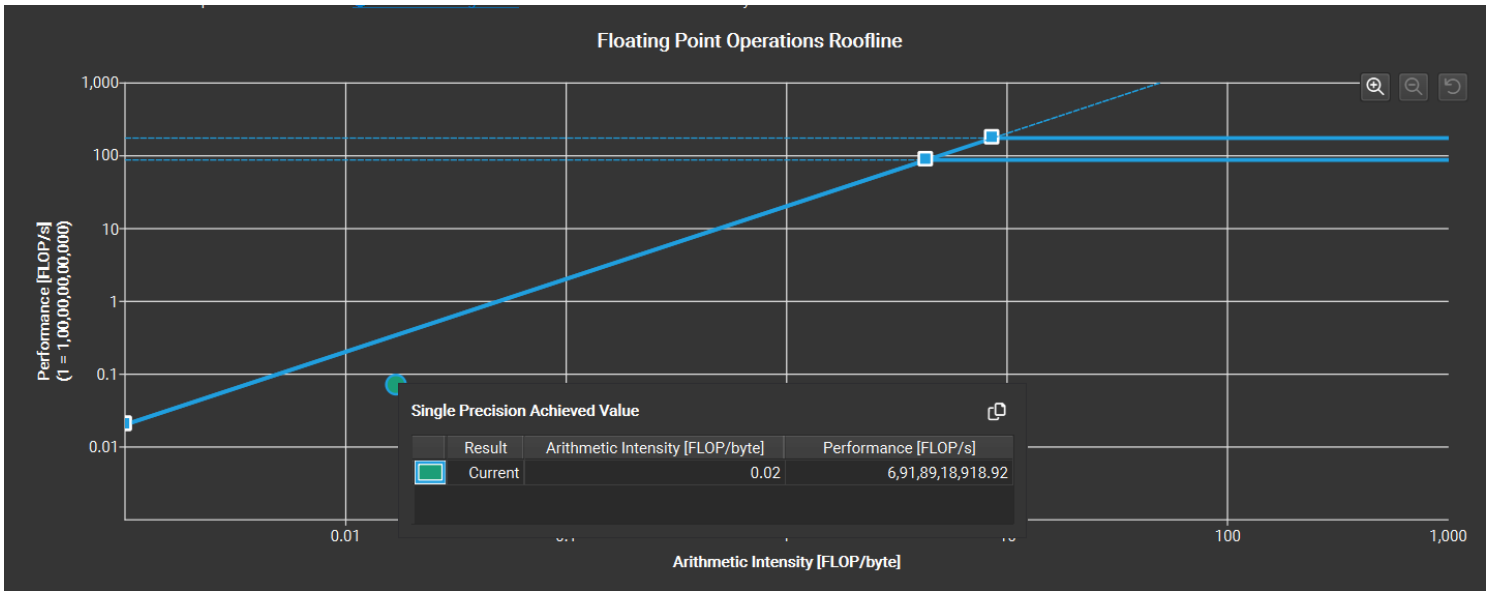
3.4.4 Top 3 kernels which consume most of the execution time in the inference.

- sm80_xmma_fprop_implicit_gemm_indexed_tf32f32_tf32f32_f32_nhwckrsc_nchw_tilesize64x32x64_stage5_warpsize2x2x1_g1_tensor16
- sm80_xmma_fprop_implicit_gemm_tf32f32_tf32f32_f32_nhwckrsc_nchw_tilesize64x32x64_stage5_warpsize2x2x1_g1_tensor16x8x8_alig
- sm80_xmma_fprop_implicit_gemm_indexed_tf32f32_tf32f32_f32_nhwckrsc_nchw_tilesize64x32x64_stage5_warpsize2x2x1_g1_tensor16

Rooflines:



(a) Roofline of the first kernel



(b) Roofline of the longest kernel

4 Inference on BERT (Part 4)

4.1 CPU vs GPU Inference Time

4.1.1 Compare inference time on CPU vs. A100 GPU for BERT-Base:

The inference time with a GPU for a batch size of 1 is approximately 6.8 times faster than the inference time with a CPU; for a batch size of 8, it is 21.96 times faster; for a batch size of 16, it is 22.89 times faster; and for a batch size of 32, it is 25.95 times faster. The GPU is a chip designed for parallel processing, and the Nvidia A100 is a state-of-the-art chip intended for training and inferencing models; hence, it outperforms the CPU.

Batch Size	Inference Time (CPU)	Inference Time (GPU)
1	0.0645 sec	0.0094 sec
8	0.3734 sec	0.0170 sec
16	0.6388 sec	0.0279 sec
32	1.2615 sec	0.0486 sec

Table 13: BERT Inference Time Comparison

Snapshots for GPU inference:

```
(myenv) [agurjar2@sg011:/scratch/agurjar2/lab2_code]$ python part4.py
Some weights of BertForQuestionAnswering were not initialized from the model checkpoint at bert-base-cased and a
re newly initialized: ['qa_outputs.bias', 'qa_outputs.weight']
You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.
100%|| 48/48 [00:01<00:00, 31.04it/s]
convert squad examples to features: 100%|| 100/100 [00:00<00:00, 260.65it/s]
add example index and unique id: 100%|| 100/100 [00:00<00:00, 1118481.07it/s]
Batch size: 1
Device: cuda
Latency :  0.0094331693649292
```

(a) Bert inference with batch size 1

```
(myenv) [agurjar2@sg011:/scratch/agurjar2/lab2_code]$ python part4.py
Some weights of BertForQuestionAnswering were not initialized from the model checkpoint at bert-base-cased and a
re newly initialized: ['qa_outputs.bias', 'qa_outputs.weight']
You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.
100%|| 48/48 [00:01<00:00, 30.61it/s]
convert squad examples to features: 100%|| 100/100 [00:00<00:00, 259.72it/s]
add example index and unique id: 100%|| 100/100 [00:00<00:00, 1115506.38it/s]
Batch size: 8
Device: cuda
Latency :  0.01702481269836426
```

(b) Bert inference with batch size 8

```
(myenv) [agurjar2@sg011:/scratch/agurjar2/lab2_code]$ python part4.py
Some weights of BertForQuestionAnswering were not initialized from the model checkpoint at bert-base-cased and a
re newly initialized: ['qa_outputs.bias', 'qa_outputs.weight']
You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.
100%|| 48/48 [00:01<00:00, 30.63it/s]
convert squad examples to features: 100%|| 100/100 [00:00<00:00, 258.31it/s]
add example index and unique id: 100%|| 100/100 [00:00<00:00, 886745.03it/s]
Batch size: 16
Device: cuda
Latency :  0.027854666709899903
```

(c) Bert inference with batch size 16

```
(myenv) [agurjar2@sg011:/scratch/agurjar2/lab2_code]$ python part4.py
Some weights of BertForQuestionAnswering were not initialized from the model checkpoint at bert-base-cased and a
re newly initialized: ['qa_outputs.bias', 'qa_outputs.weight']
You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.
100%|| 48/48 [00:01<00:00, 30.92it/s]
convert squad examples to features: 100%|| 100/100 [00:00<00:00, 258.63it/s]
add example index and unique id: 100%|| 100/100 [00:00<00:00, 890510.40it/s]
Batch size: 32
Device: cuda
Latency :  0.04863374710083008
```

(d) Bert inference with batch size 32

5 Conclusion

For SimpleMLP, we combined CLI-based profiling tools (e.g., nsys, SQLite queries) with manual calculations to analyze GPU utilization, DRAM bandwidth, FLOPs, and arithmetic intensity. Our findings indicate extremely low GPU utilization (0.17% for training, 0.04% for inference), which we suspect is caused by short-lived kernels and insufficient parallelism. Despite this, the high arithmetic intensity (13 FLOPs/byte) suggests efficient data reuse but limited performance due to DRAM bandwidth constraints (1.05 GB/s for training, 7.62 GB/s for inference). Kernel-level analysis showed that SGEMM (matrix multiplication) and elementwise operations dominated execution, with no Tensor Core utilization, indicating potential opportunities for precision tuning or architectural optimizations. For ResNet-18, we leveraged NVIDIA profiling tools (nsys, ncu) for detailed kernel-level analysis. Compared to SimpleMLP, we observed higher GPU utilization (14.41% for training) but still suboptimal efficiency, with SMs frequently idle due to kernel scheduling gaps and data transfer bottlenecks. DRAM bandwidth usage peaked at 82.95 GB/s, which is only 4.3% of the A100's theoretical peak of 1,935 GB/s, highlighting underutilization of memory bandwidth. Kernel analysis revealed dominance of Tensor Core operations, including sm80_xmma_fprop, dgrad, and wgrad, confirming that mixed-precision (TF32) computation is well-optimized in ResNet.

Overall,

1. MLP models suffer from GPU underutilization due to short-lived kernels and limited parallelism.
2. CNN models like ResNet exhibit moderate GPU usage, with underutilized memory bandwidth and kernel scheduling inefficiencies.
3. ResNet benefits from Tensor Core acceleration, unlike MLP, which lacks architectural optimizations.
4. The Roofline Model analysis highlights computational bottlenecks and memory constraints, indicating a gap between achievable and peak heoretical performance.
5. GPUs perform well for large matrix operations (e.g., BERT, ResNet), but smaller tasks or limited batch sizes result in inefficient CUDA core execution.