### **Literature Review: FP8 Quantization and Its Application in Dataloaders**

1. **FP8 Quantization Strategy**: The FP8 format, as explored in the paper, consists of two main configurations:

* **E4M3**: 1-bit sign, 4-bit exponent, 3-bit mantissa.
* **E5M2**: 1-bit sign, 5-bit exponent, 2-bit mantissa.

These configurations provide distinct trade-offs. **E4M3** offers greater precision but a smaller dynamic range, whereas **E5M2** extends the dynamic range, making it suitable for representing larger values with less precision. The paper argues that the **power of the exponent**—with careful selection between these two configurations—can enable models to retain accuracy while benefiting from substantial reductions in memory and computation

1. **Data Pipeline Efficiency**:
   * **Data Preprocessing**: Use FP8 for memory-intensive steps like normalization and augmentation to reduce compute and memory needs, particularly during batch processing.
   * **Data Storage**: Store datasets in FP8 to save disk space and lower I/O costs, speeding up data retrieval.
   * **Batch Transfer**: Quantize batches to FP8 for multi-GPU setups, optimizing memory usage and transfer efficiency.
2. **Hardware Compatibility**: Ensure compatibility with modern GPUs supporting FP8 to leverage native FP8 operations. On older hardware, assess if the quantization overhead offsets the memory and compute gains.
3. **Precision Management**: Carefully manage quantization and dequantization boundaries, especially for layers needing higher precision, to avoid cumulative rounding errors and maintain accuracy.
4. **Challenges**:
   * **Precision Loss**: Selectively apply FP8 for non-critical data in precision-sensitive tasks (e.g., medical imaging).
   * **Error Propagation**: Address potential error accumulation across pipeline stages with quantization-aware processing.

Using FP8 within dataloaders can significantly streamline data handling in memory-constrained and distributed systems, while managing precision trade-offs for optimal model training and inference performance.

**Question:**

1. Do we have to use Cuda C - I feel like we have to write in python using torch library?

Our code are supposed to manually distribute the data to cuda, pool everything and have an args to define the quantization method

1. How to start writing this from scratch? Any detailed guidance?
2. How to test the correctness of dataloader before testing efficiency?