1. How to Reproduce Our Results (Markov Cluster, H100)

This section walks from an empty shell on Markov all the way to plots and JSON result files.

Assumptions

- You are on the `markov_gpu` partition.
- Repo is cloned at: `~/csds451_final_project`
- GPU: H100 MIG (`NVIDIA H100 NVL MIG 1g.12gb`)

1.1. Start an interactive GPU session :

```
srun -A csds451 -p markov_gpu --gres=gpu:1 -c 4 --mem=32G --time=03:00:00 --pty bash
```

All commands below are assumed to be run **inside** this interactive session.

1.2. Load required modules :

```
module purge
module avail CUDA      # just to show available CUDA versions
module load CUDA/12.1.1
(CUDA 12.1.1 is what we tested with)
```

1.3. Create & activate Python virtual environment

If you don't already have `flash_env`:

cd ~

```
python3 -m venv flash_env

source ~/flash_env/bin/activate

pip install --upgrade pip

pip install torch torchvision torchaudio --index-url https://download.pytorch.org/whl/cu121

pip install matplotlib numpy
```

On future runs you only need: source ~/flash_env/bin/activate (You should see `(flash_env)` in your prompt.

1.4. Clone / enter the project :

```
cd ~
cd csds451_final_project
```

Project layout (relevant part):

```
csds451_final_project/
  flash-attention/        # reference FlashAttention repo
  experiments/
    baseline_cuda/
    flash_cuda/
    results/
    plots/
```

1.5. H100 compatibility: set architecture

On H100 we must target compute capability 9.0:

```
export TORCH_CUDA_ARCH_LIST="9.0"
```

We also explicitly verified the device:

```
python3 - << 'EOF'

import torch

print("Device 0:", torch.cuda.get_device_name(0))

print("Capability:", torch.cuda.get_device_capability(0))

EOF

# Device 0: NVIDIA H100 NVL MIG 1g.12gb

# Capability: (9, 0)
```

2. Phase-by-Phase: Running All Experiments

Below is the exact sequence to reproduce our experiments from Phase 0 to Phase 7.

Phase 0: Environment & Sanity Checks

From the project root:

```
cd ~/csds451_final_project

source ~/flash_env/bin/activate

module load CUDA/12.1.1

export TORCH_CUDA_ARCH_LIST="9.0"
```

Quick sanity check that PyTorch sees the GPU:

```
python3 - << 'EOF'

import torch

print("CUDA available:", torch.cuda.is_available())

print("Device:", torch.cuda.get_device_name(0))

EOF
```

Phase 1: Baseline Multi-Head Attention (PyTorch)

We treat this as the "vanilla" implementation, mainly limited by memory bandwidth and not fully optimized like FlashAttention.

```
cd ~/csds451_final_project/experiments/baseline_cuda

python3 baseline_attention.py
```

Expected console output (numbers may differ slightly):

```
[BASELINE] L=512, B=16, d_model=512, H=8
[BASELINE RESULT]
  L: 512
  B: 16
  D: 512
  H: 8
  avg_latency_ms: ...
  avg_power_watts: ...
  max_power_watts: ...
[BASELINE] L=1024, B=16, d_model=512, H=8
[BASELINE RESULT]
  L: 1024
  B: 16
  D: 512
  H: 8
  avg_latency_ms: ...
  avg_power_watts: ...
  max_power_watts: ...
Saved baseline results to results/baseline_results.json
```

The JSON file ends up at:

```
ls ~/csds451_final_project/experiments/results
# baseline_results.json
```

Phase 2: FlashAttention CUDA Reference Kernel

This phase uses a FlashAttention-style kernel on the GPU, wrapped from the reference project (`flash-attention/Flash_py`) and measured with our power monitor.

```
cd ~/csds451_final_project/experiments/flash_cuda

python3 flash_attention_cuda.py
```

Expected output pattern:

```
[FLASH] L=512, B=16, d_model=512, H=8

[FLASH RESULT]

  ...

[FLASH] L=1024, B=16, d_model=512, H=8

[FLASH RESULT]

  ...

Saved flash attention results to results/flash_cuda_results.json
```

JSON file:

```
ls ~/csds451_final_project/experiments/results

# flash_cuda_results.json
```

Phase 3: Custom CUDA Kernel (Our Kernel): This is our own CUDA kernel wired as a PyTorch extension. It exercises the same Q/K/V tensors but with a simplified kernel so we can focus on:

- GPU launch configuration
- Block size / grid size
- Power behavior of a tiled kernel

3.1. Build & test the CUDA extension

Make sure we are in the right directory:

```
cd ~/csds451_final_project/experiments/flash_cuda
```

Then:

```
python3 - << 'EOF'
import flash_custom
print("Custom CUDA extension built and loaded!")
print("forward() available:", hasattr(flash_custom,
"forward"))
EOF
```

On success you should see:

```
Custom CUDA extension built and loaded!
forward() available: True
```

If this is the first time on a node, PyTorch will JIT-compile the extension using nvcc. That may take a little bit the first time.

3.2. Run the custom kernel experiment :

```
cd ~/csds451_final_project/experiments/flash_cuda
python3 run_custom_flash_experiment.py
```

Example H100 output we obtained:

```
[CUSTOM CUDA] L=512, B=16, d_model=512, H=8
[CUSTOM RESULT]
  L: 512
```

```
  B: 16
  D: 512
  H: 8
  avg_latency_ms: 0.12589693069458008
  avg_power_watts: 70.94945999999999
  max_power_watts: 77.685

[CUSTOM CUDA] L=1024, B=16, d_model=512, H=8
[CUSTOM RESULT]
  L: 1024
  B: 16
  D: 512
  H: 8
  avg_latency_ms: 0.2353668212890625
  avg_power_watts: 86.91312000000003
  max_power_watts: 92.899

Saved custom CUDA results to ../results/custom_cuda_results.json
```

Phase 4: Inspecting Raw Results (JSON)

```
cd ~/csds451_final_project/experiments/results
python3 - << 'EOF'
import json, glob, pprint

for f in glob.glob("*.json"):
    print("\n======", f, "======")
    with open(f) as fp:
        data = json.load(fp)
        pprint.pprint(data)
EOF
```

You should see three files:

- **baseline_results.json**
- **flash_cuda_results.json**
- **custom_cuda_results.json**

Each contains entries for L=512 and L=1024 with:

- **avg_latency_ms**
- **avg_power_watts**
- **max_power_watts**

Phase 5: Plotting (Latency, Power, Speedup, Energy)

Plots live under experiments/plots.

```
cd ~/csds451_final_project/experiments/plots
```

We generate multiple plots:

1. Combined plots (latency, power, speedup)

```
python3 generate_plots.py

#Produces: latency_plot.png, power_plot.png, speedup_plot.png
```

2.Max power comparison:

```
python3 plot_max_power.py

#Produces: max_power_comparison.png
```

3.Average power comparison:

```
python3 plot_avg_power.py

#Produces: avg_power_comparison.png
```

4.Energy per run :

```
python3 plot_energy.py

#Produces: energy_comparison.png (per kernel, per sequence length)
```

5.Speedup plot :

```
python3 plot_speedup.py

#Produces: speedup_custom_vs_baseline.png
```

All PNGs are saved in experiments/plots/.

Phase 6 / 7: End-to-End Repro Summary :

```
# 0. Get GPU shell

srun -A csds451 -p markov_gpu --gres=gpu:1 -c 4 --mem=32G
--time=03:00:00 --pty bash

# 1. Load modules + env

module purge

module load CUDA/12.1.1

source ~/flash_env/bin/activate

export TORCH_CUDA_ARCH_LIST="9.0"

# 2. Baseline

cd ~/csds451_final_project/experiments/baseline_cuda

python3 baseline_attention.py

# 3. FlashAttention CUDA reference

cd ~/csds451_final_project/experiments/flash_cuda

python3 flash_attention_cuda.py

# 4. Custom CUDA kernel

cd ~/csds451_final_project/experiments/flash_cuda

python3 run_custom_flash_experiment.py

# 5. Plots

cd ~/csds451_final_project/experiments/plots

python3 generate_plots.py
```

```
python3 plot_max_power.py

python3 plot_avg_power.py

python3 plot_energy.py

python3 plot_speedup.py
```

3. Experimental Results & Patterns We Observed

Below are **representative H100 measurements** for B=16, d_model=512, H=8.

Kernel	L = 512	L = 1024
Baseline PyTorch	1.56 ms	4.88 ms
FlashAttention CUDA	1.57 ms	5.10 ms
Custom CUDA	0.13 ms	0.24 ms

Speedups of custom vs baseline:

- `L = 512`: ≈ **12.4× faster**
- `L = 1024`: ≈ **20.7× faster**

The larger sequence length amplifies the speedup, which matches the theoretical $O(L^2)$ cost of attention: as L doubles, the amount of work quadruples, but our tiled kernel benefits more from better GPU utilization at higher arithmetic intensity.

Kernel	L = 512 (avg W)	L = 512 (max W)	L = 1024 (avg W)	L = 1024 (max W)
Baseline PyTorch	≈ 86 W	≈ 116 W	≈ 160 W	≈ 250 W
FlashAttention CUDA	≈ 83 W	≈ 112 W	≈ 155 W	≈ 216 W
Custom CUDA	≈ 71 W	≈ 78 W	≈ 87 W	≈ 93 W

Patterns we saw:

- **Baseline vs FlashAttention:**
  FlashAttention slightly **reduces both average and peak power** vs baseline for the same sequence length, which we interpret as better cache use and less memory thrashing. The kernel finishes in a similar time, but it accesses memory more "politely," so instantaneous power is a bit lower.

- **Custom kernel:**
  Our custom kernel has **significantly lower max power**. That's partly because it does less total arithmetic (we focused on a simplified kernel), but also because we tuned the block/grid launch to avoid oversubscribing the SMs. The GPU is busy, but not "saturating" all power rails like the baseline L^2 attention.

3.3. Energy per Run (Joules)

We approximate energy as:

Energy ≈ avg_power_watts × latency_seconds

Representative numbers:

Kernel	L = 512	L = 1024
Baseline PyTorch	≈ 0.134 J	≈ 0.78 J
FlashAttention CUDA	≈0.134 J	≈ 0.79 J
Custom CUDA	≈ 0.009 J	≈ 0.020 J

So for L = 1024, our custom kernel uses **~38× less energy** than the baseline (0.78 J → 0.02 J) on the H100 MIG slice.

Why this pattern makes sense:

- The baseline and FlashAttention kernels both do full attention math, which is expensive and memory heavy.
- Our custom kernel intentionally simplifies the computation while respecting the same tensor shapes, so the GPU spends far less time in high-power regions.

- Even though average power of the custom kernel is not negligible, the **runtime is so short** that the area under the power–time curve (energy) drops dramatically.

This directly connects to the professor's three metrics:

1. **Maximum power**: our plots show that baseline has the highest peaks, FlashAttention reduces them, and our custom kernel keeps them lowest.

2. **Energy (area under power–time curve)**: we explicitly compute and plot this, and it is where the custom kernel shines.

3. **Total time**: captured via avg_latency_ms and visualized in the latency and speedup plots.

4. What We Learned About Parallel Algorithms & FlashAttention

This is the "10-point discussion" part, where we go beyond "A is better than B."

4.1. Tiling, Blocking, and Memory Locality Matter More Than Raw FLOPs

Baseline attention is easy to write but **memory-bound**: Q/K/V and attention matrices bounce in and out of DRAM repeatedly.

FlashAttention's reference kernel and our experiments highlight:

- Tiling Q/K/V into **on-chip memory (shared/registers)** reduces DRAM traffic.

- Fused softmax + matmul means fewer kernel launches and less round-trip to global memory.

- Even when runtime is similar, the **power profile** improves because the GPU isn't constantly stalling on memory.

Takeaway for us: **"Fast" on GPU usually means "fewer painful global memory trips," not just "more threads."**

4.2. Block Size Is a Speed–Power Dial, Not Just a Speed Dial

By writing our own kernel and tuning block dimensions, we saw that:

- Very large blocks can maximize occupancy but also spike instantaneous power and sometimes increase energy if they cause contention or register pressure.

- Moderately sized blocks gave us **good latency with lower peaks** and smoother power traces.

- On H100 MIG, we are power-limited; the GPU will happily draw more power if you let it. Being aware of block size lets us trade off **throughput vs. energy**.

This was not obvious from reading FlashAttention papers alone; we had to actually tweak grid/block sizes and watch the power monitor.

4.3. Hardware Differences Change the Story

We ran earlier versions on an RTX 2080 Ti, then on H100 MIG:

- The **absolute** numbers changed a lot (H100 is far faster and has different power behavior).

- But **relative trends stayed consistent**:
    - Baseline: highest energy & power.
    - FlashAttention: slightly better power behavior through tiling/fusion.
    - Our custom kernel: shortest runtime and lowest energy for the simplified workload.

This taught us that **"shape of curves" is more portable than raw numbers**. When optimizing parallel algorithms, we should think in terms of:

- Scaling with sequence length L
- Scaling with model dimension D
- Trade-offs between compute and memory on each architecture

4.4. Debugging & Integration Is Half the Battle

Non-trivial but important lessons we picked up:

- Getting PyTorch extensions to compile with the **right** `TORCH_CUDA_ARCH_LIST` is critical, especially moving between 2080 Ti (`7.5`) and H100 (`9.0`).
- C++/CUDA compilation errors about `half`, `dim3`, or GCC versions forced us to understand:
    - How PyTorch passes in include paths for CUDA.

    - Why CUDA 11.1 doesn't like GCC 12 without `-allow-unsupported-compiler`.

- The exercise was less about beating NVIDIA's own optimized kernels and more about **understanding the entire stack**:

    - From Python scripts → C++ binding → CUDA kernels → H100 hardware + NVML power monitoring.

Phase 8: Extended Energy & Efficiency Analysis

8.1 Energy Trends Across Kernels

By combining latency and average power:

$$E = P_{\text{avg}} \times t$$

we observe:

- Baseline attention is the **most energy-consuming**, especially at larger sequence lengths.

- FlashAttention reduces unnecessary memory movement, resulting in **slightly lower energy** for similar runtimes.

- Our custom CUDA kernel has **drastically lower energy**, not because it performs more work, but because it completes work *extremely quickly*.

Why this matters

Energy efficiency is increasingly a design constraint in large-scale AI systems. Our results show that:

- Kernel design choices impact energy more than raw speed.
- Short runtimes multiply energy savings, even if power draw remains moderate.

This aligns with modern GPU algorithm design, where minimizing “active time” yields better ROI than simply reducing wattage.

Phase 9: GFLOPs-per-Watt Efficiency Discussion

This section explains how and why efficiency curves behave the way they do.

9.1 Efficiency Patterns

We estimated FLOPs using FlashAttention’s theoretical complexity:

O(B·H·L2·DH)\mathcal{O}(B \cdot H \cdot L^2 \cdot \frac{D}{H})O(B·H·L2·HD)

and normalized to power consumption:

GFLOPs/W=FLOPs109·Power\text{GFLOPs/W} = \frac{\text{FLOPs}}{10^9 \cdot \text{Power}}GFLOPs/W=109·PowerFLOPs

Observations

1. **Baseline attention achieves the lowest efficiency.**
   Its O(L²) operations rely heavily on memory bandwidth, creating stalls that reduce actual FLOP utilization per watt.
2. **FlashAttention significantly improves efficiency.**
   Its tiling allows work to stay on-chip longer, reducing

DRAM access and unlocking higher SM utilization.

3. **Custom CUDA kernel shows the *highest* GFLOPs/W.**
   This is expected because our simplified kernel removes:
   - masking
   - softmax
   - normalization
   - matrix multiplications
4. It performs less arithmetic but optimizes the execution path extremely well, making efficiency (FLOPs/W) appear high.

Why this matters

Our comparison helps illustrate that:

- **Efficiency should be interpreted relative to algorithmic workload**, not only hardware capability.
- Simplified kernels can appear extremely efficient; real FlashAttention trades some efficiency for correctness and stability.

Phase 10: Combined 3-Panel Figure Summary

We generated a publication-style 3-panel PDF figure containing:

1. **Energy per run** (baseline vs FlashAttention vs custom)
2. **GFLOPs/W efficiency**
3. **Latency comparison**

This multi-view figure captures the essence of our optimization study:

- How fast each kernel is
- How much power it uses
- How much energy it consumes
- How efficiently it converts energy into useful computation

Why this is valuable

A single combined PDF is ideal for:

- Reports
- Presentations
- Posters
- GitHub documentation

Phase 11: What This Project Taught Us (Final Reflection)

1.1 End-to-End GPU Optimization Is a Multi-Layer Stack

We had to understand and debug interactions between:

- Python drivers
- PyTorch C++ frontend
- CUDA kernels
- nvcc compiler
- H100 hardware architecture
- NVML power instrumentation

This forced us to think like systems engineers, not just programmers.

11.2 GPU Performance Is Memory-Bound More Often Than Compute-Bound :

Memory hierarchy design dominates GPU algorithm performance.FlashAttention's breakthrough is not "**faster math**" but "**better memory movement**"

11.3 Block/Tiling Configuration Has Real Energy Implications

We discovered that:

- **Oversaturating SMs increases instantaneous power.**
- **Moderate occupancy often yields superior energy-per-workload.**
- **Block sizes act as a *tuning knob* between performance and efficiency.**

- This insight is rarely obvious until you run real power measurements.

11.4 H100 MIG behaves differently from consumer GPUs

We originally developed some parts on an RTX 2080 Ti (sm_75) and later moved to H100 MIG (sm_90):

- H100 has more stable power curves.
- Its SM scheduling is more aggressive.
- It throttles differently under sustained load.

These differences taught us about hardware portability and the importance of testing across architectures.

11.5 Energy Matters More Than Speed Alone :

A kernel that is “**fast**” but “**power hungry**” may be worse overall than a moderately slower one that uses half the power.

Future transformer kernels must balance:

- computation
- memory traffic
- power draw
- temperature
- cost per token

Dynamic FlashAttention++ – Power-Aware GPU Scheduling for Transformer Attention

Dynamic FlashAttention++ – Power-Aware GPU Scheduling for Transformer Attention