# Title: C-Language Empowered GPU Kernel Optimization Towards Greener AI

## Abstract:

This paper examines how low-level programming, particularly in C/CUDA, can significantly impact kernel activity on GPU/TPU architectures, resulting in more efficient and environmentally conscious AI inference. With modern AI workloads consuming extensive computational and environmental resources, this investigation highlights how direct kernel control using C can improve memory access, computation throughput, and energy efficiency. We bridge the complex hardware-level theory and practical CUDA examples to present an accessible roadmap for optimizing AI workloads.

## 1. Introduction:

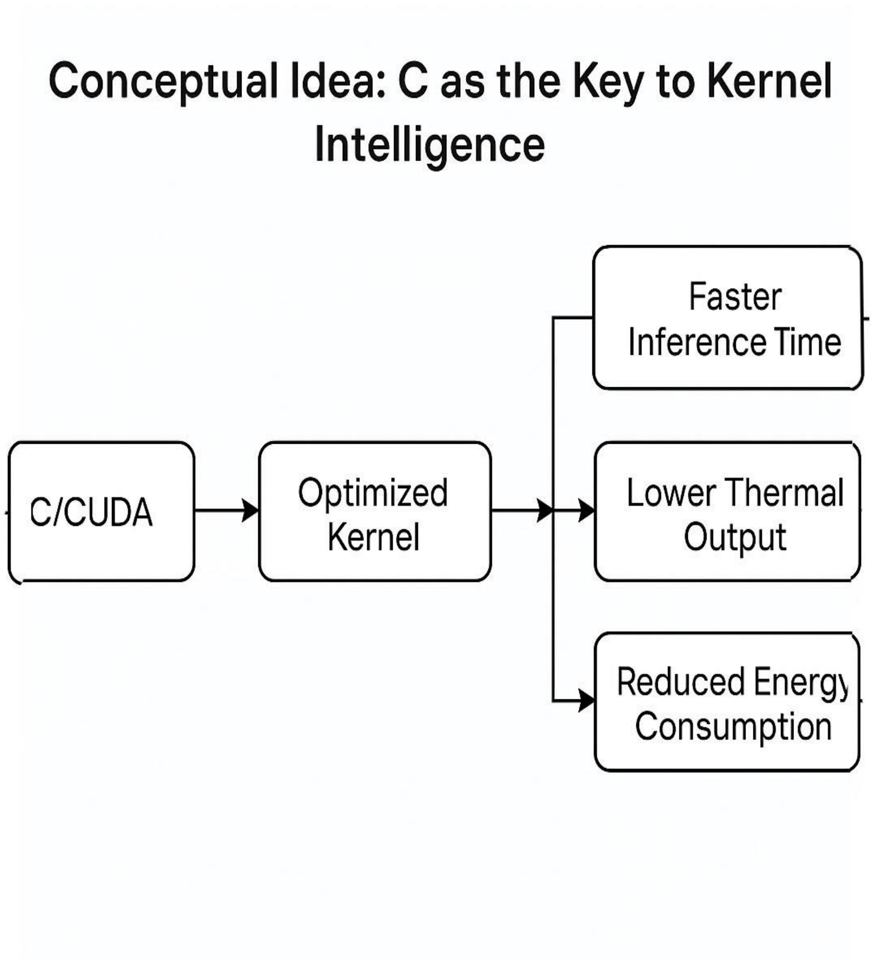
Artificial Intelligence (AI) models, profound deep learning architectures, are increasingly deployed for real-time inference. These workloads demand intensive computation and memory bandwidth, often resulting in significant energy consumption. While high-level libraries such as TensorFlow and PyTorch simplify AI deployment, they abstract away hardware control, resulting in less optimized kernel executions. This paper advocates the use of C and CUDA to enable direct control over GPU kernel behavior for performance gain.

## 2. Background:

### 2.1. GPU Kernel Fundamentals

A GPU Kernel is a function executed by multiple threads in parallel. Each thread performs a small portion of a larger task. Optimizing this function directly determines how efficiently AI tasks can be executed.

## Conceptual Idea Diagram:


Conceptual Idea: C as the Key to Kernel Intelligence
C/CUDA
Optimized Kernel
Faster Inference Time
Lower Thermal Output
Reduced Energy Consumption

## 2.2. Problem with High-Level Abstractions

High-Level APIs automatically generate kernels, often without tuning memory usage or thread patterns for specific hardware. This results in:

- Memory Latency
- Redundancy in Computation
- Poor Cache utilization

## 2.3. Role of C/CUDA in Low-Level Optimization

C, when extended with CUDA, for NVIDIA GPUs, enables developers to:

- Manually assign thread blocks and grid dimensions
- Used shared memory explicitly
- Minimize global memory access
- Perform instruction-level parallelism

## 3. Computational Idea: C as the Key to Kernel Intelligence

This idea stems from the notion that C-Level kernel writing allows fine-tuning of GPU/TPU behavior. Rather than relying on generic, opaque kernels, developers can hand-craft memory access, synchronization, and compute scheduling for:

- Faster inference time
- Lower thermal output
- Reduced energy consumption

## 4. Case Study: Shared Memory Matrix Multiplication in CUDA

```
#define TILE_SIZE 16
#define N 512


__global__ void MatMulShared(float *A, float *B, float *C, int N)
{
    __shared__ float Asub[TILE_SIZE][TILE_SIZE];  // tile from A
    __shared__ float Bsub[TILE_SIZE][TILE_SIZE];  // tile from B

    int row = blockIdx.y * TILE_SIZE + threadIdx.y;
    int col = blockIdx.x * TILE_SIZE + threadIdx.x;

    float sum = 0.0f;

    for (int tile = 0; tile < N / TILE_SIZE; ++tile) {
        Asub[threadIdx.y][threadIdx.x] = A[row * N + tile *
TILE_SIZE + threadIdx.x];

        Bsub[threadIdx.y][threadIdx.x] = B[(tile * TILE_SIZE +
threadIdx.y) * N + col];

        __syncthreads(); // wait for all threads

        for (int k = 0; k < TILE_SIZE; ++k) {
            sum += Asub[threadIdx.y][k] * Bsub[k][threadIdx.x];
        }

        __syncthreads(); // reuse tile
    }


    C[row * N + col] = sum;

}
```

## Explanation

- Shared Memory: Used to store titles from matrices A and B for faster access
- Thread Cooperation: Threads load data cooperatively and reuse it across computation
- Performance Benefit: Reduces global memory access, increasing throughput

## 5. Future Directions: TPU and MLIR/XLA

While TPUs don't currently allow direct C access, compilers like MLIR and XLA could benefit from C-style thinking:

- Considering memory locality in loop fusion
- Optimize tensor layout
- Prioritize low-level IR tuning

Future hardware-aware compilers may expose more granular kernel tuning hooks for TPU accelerators.

## 6. Conclusion

By stepping beyond high-level abstractions and embracing C/CUDA, we can unlock granular control over kernel behavior, enabling powerful performance gains and environmental efficiency. This approach bridges the gap between AI's increasing resource demands and the imperative for sustainable computing.

## Appendix: Installation of CUDA and Run *.cu* Files

- Install the CUDA Toolkit
- Visit: https://developer.nvidia.com/cuda-downloads
- Choose the operating system and follow the installation instructions

## Confirm installation using:

**Setup Environment**

- Ensure CUDA is added to the system PATH

  **Example path:**

  ```
  C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v11.X\bin
  ```

  **Run the Executable**

  ```
  ./matmul     # Linux/macOS

  matmul.exe   # Windows
  ```

  **Compile and Run**

  ```
  nvcc matmul.cu -o matmul
  ```

## Tips:

- N.B. Start with small matrix sizes for testing
- Match the CUDA version to your GPU driver
- Use GPU-Z or Nvidia-Smi to monitor usage

## Keywords

- C language, GPU kernel, CUDA, shared memory, AI inference, optimization, eco-friendly AI, TPU, MLIR, XLA