

@ -1,22 +1,57 @@

Project 3: Improved Matrix Multiplication in C

Project 5: GPU Acceleration with CUDA

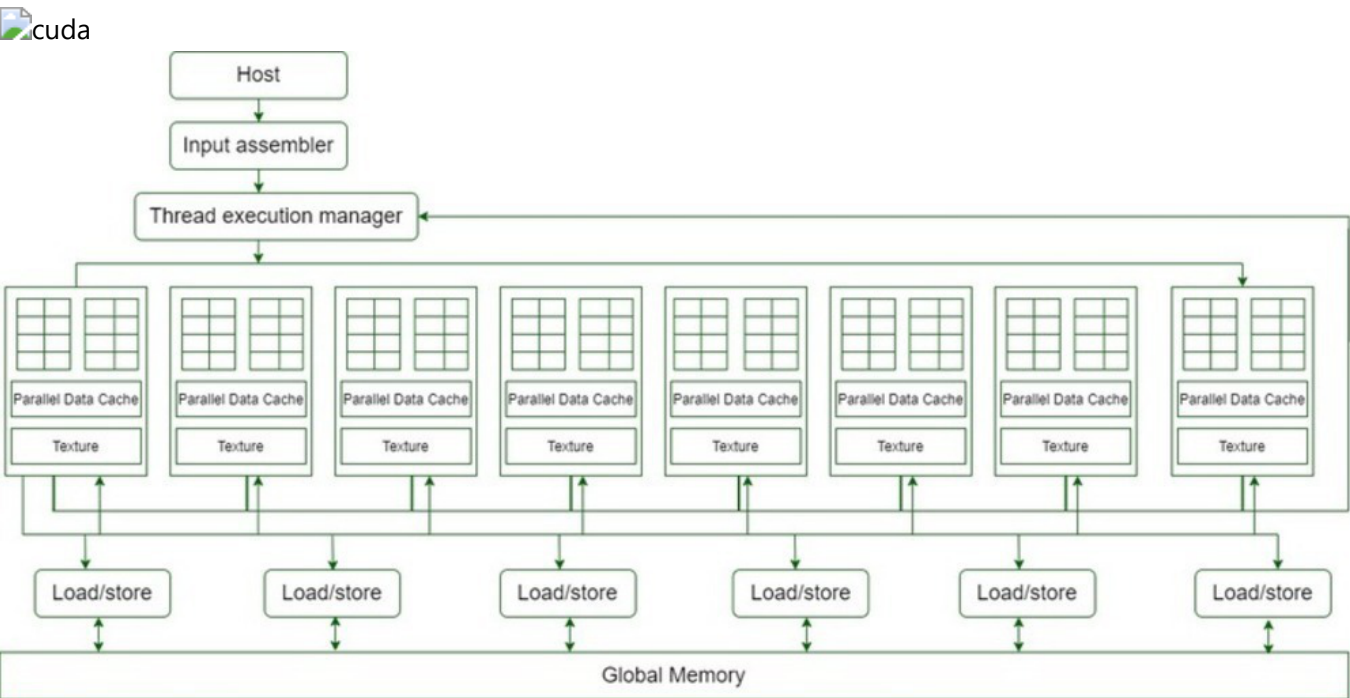
12113053 THA Sreyny

Introduction

I. Introduction

Why do we need CUDA? GPUs are designed to perform high-speed parallel computations to display graphics such as games. Use available CUDA resources. It provides 30-100x speed-up over other microprocessors for some applications. GPUs have very small Arithmetic Logic Units (ALUs) compared to the somewhat larger CPUs. This allows for many parallel calculations, such as calculating the color for each pixel on the screen, etc.

Architecture of CUDA Architecture of CUDA Design



credit: geeksforgeeks.org

Task1

The diagram shows 16 Streaming Multiprocessors (SMs), each with 8 Streaming Processors (SPs), totaling 128 SPs. Each SP includes a MAD (Multiplication and Addition) unit and a multiplication unit.

Key points:

- GT200: 30 SMs, each with 8 SPs, totaling 240 SPs, and over 1 TFLOP processing power.

- G80 card: 16 SMs, each with 8 SPs, totaling 128 SPs. Supports 768 threads per SM (96 threads per SP), allowing up to 12,288 concurrent threads across 128 SPs.
- Memory Bandwidth: 86.4GB/s for G80 chips.
- CPU Communication: 8GB/s (4GB/s each for uploading to and downloading from CPU RAM).

These features contribute to the massively parallel nature of these processors.

Typical CUDA program flow:

- Load data into CPU memory.
- Copy data from CPU to GPU memory using `cudaMemcpy(..., cudaMemcpyHostToDevice)`.
- Call the GPU kernel with device variables using `kernel<<<>>> (gpuVar)`.
- Copy results from GPU to CPU memory using `cudaMemcpy(..., cudaMemcpyDeviceToHost)`.
- Use the results on the CPU.

How CUDA works?

- GPUs run one kernel (a group of tasks) at a time.
- Each kernel consists of blocks, which are independent groups of ALUs.
- Each block contains threads, which are levels of computation.
- The threads in each block typically work together to calculate a value.
- Threads in the same block can share memory.
- In CUDA, sending information from the CPU to the GPU is often the most typical part of the computation.
- For each thread, local memory is the fastest, followed by shared memory, global, static, and texture memory the slowest.

Work distribution in CUDA:

- Each thread knows its block's x and y coordinates and its position within the block.
- These positions are used to calculate a unique thread ID for each thread.
- The computational work assigned to a thread depends on its thread ID.

II. Task 1

There two methods to calculate $A=aB+b$. One uses CPU and another one uses GPU.

```
```cpp
bool addCPU(const Matrix * pMat1, Matrix * pMat2, float a, float b)
{
 if(pMat1 == NULL
@ -38,7 +73,7 @@ bool addCPU(const Matrix * pMat1, Matrix * pMat2, float a, float
b)
}
```

The CUDA kernel for performing the same operation on the GPU. Each thread processes one element of the array.

```

```cpp
__global__ void addKernel(const float * input1, const float * input2, float *
output, size_t len, float a, float b)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
@ -46,7 +81,7 @@ __global__ void addKernel(const float * input1, const float *
input2, float * ou
        output[i] = input1[i] * a + b;
}

```

```

```cpp
bool addGPU(const Matrix * pMat1, Matrix * pMat2, float a, float b)
{
 if(pMat1 == NULL
@ -76,10 +111,13 @@ bool addGPU(const Matrix * pMat1, Matrix * pMat2, float a,
float b)
}

```

## Result



## Task 2

```

sreyeny@SREYNY-ROGSTRIX:/mnt/d/SUSTech-Courses/CS205-C++/project5$ nvcc matrix.cu
sreyeny@SREYNY-ROGSTRIX:/mnt/d/SUSTech-Courses/CS205-C++/project5$./a.out
You have 1 CUDA devices.
You are using device 0.
addCPU Time = 74.432000 ms.
 Result = [5.0, ..., 5.0]
addGPU Time = 24.414000 ms.
 Result = [5.0, ..., 5.0]
sreyeny@SREYNY-ROGSTRIX:/mnt/d/SUSTech-Courses/CS205-C++/project5$

```

Based on the result, when using CUDA to calculate  $B = aA + b$ , it can execute more than 2 times faster than normal CPU calculation for matrix size of 4096.

## III. Task 2

Compare the matrix multiplication using openBlas and cuBlas.

```

```cpp
void matrixMultiplyOpenBLAS(int N) {
    float *A = new float[N * N];
    float *B = new float[N * N];
@ -104,7 +142,7 @@ void matrixMultiplyOpenBLAS(int N) {

```

```
delete[] C;
}
```

```
```cpp
void matrixMultiplyCuBLAS(int N) {
 float *h_A = new float[N * N];
 float *h_B = new float[N * N];
@ -147,11 +185,13 @@ void matrixMultiplyCuBLAS(int N) {
}
```

## Result

```
-O3 -lcublas -lopenblas -xcompiler "-Wall" && ./a.out
Matrix multiplication with OpenBLAS and cuBLAS for 4096x4096 matrices
OpenBLAS: 0.481412 seconds
cuBLAS: 0.00743191 seconds
```

The result shows that cuBlas is about 60 times faster than openBlas for matrix size of 4096 with -O3 flag.

## Advantages and disadvantages of openBlas

### Advantages:

- The code is simple, without the need for device memory management.
- Runs on any system without the need for a compatible GPU.
- The code is simple, without the need for device memory management like GPU.
- Runs on any operating system without the need for a compatible GPU.

### Disadvantages:

- Constrained by the computational power and memory bandwidth of the CPU.
- Not suitable for very large matrices or high-performance computing tasks requiring the power of a GPU. @ -159,12 +199,176 @@ void matrixMultiplyCuBLAS(int N) { **Advantages:**
- Utilizes the parallel processing power of the GPU, significantly speeding up matrix multiplication for large matrices.
- Suitable for large-scale computations and high-performance computing tasks.

### Disadvantages:

- Requires careful management of device memory, including allocation, data transfer, and deallocation.
- Requires a compatible GPU and appropriate drivers, limiting portability to systems with the necessary hardware.

## Performance Comparison

- OpenBLAS will generally be slower, especially for large matrices, due to the limited computational power and memory bandwidth of the CPU.
- cuBLAS leverages the massive parallelism and high memory bandwidth of the GPU, offering much faster execution for large matrices. **Scalability:**

- OpenBLAS performance scales poorly with matrix size compared to cuBLAS.
- cuBLAS scales much better, efficiently handling larger matrices due to the GPU's parallel architecture.


## IV. How to Optimize a CUDA Matmul Kernel for cuBLAS-like Performance: a Worklog

### 1. Kernel 1: Naive Implementation

In the CUDA programming model, computation is structured in a three-level hierarchy:

- Grid: Each kernel invocation creates a grid.
- Blocks: The grid is composed of multiple blocks.
- Threads: Each block can contain up to 1024 threads.

Key details:

- Threads within the same block can access a shared memory region (SMEM).
- The number of threads in a block is specified by `blockDim`, a vector with three integers: `blockDim.x`, `blockDim.y`, and `blockDim.z`, representing the dimensions of the block. 

credit: siboehm.com

In the first kernel, each thread is assigned a unique entry in the result matrix  $C$  using the grid, block, and thread hierarchy. Each thread computes the dot product of the corresponding row of  $A$  and column of  $B$ , writing the result to  $C$ . Since each thread writes to a unique location in  $C$ , no synchronization is needed. The kernel is launched accordingly.

```
// create as many blocks as necessary to map all of C
dim3 blockDim(CEIL_DIV(M, 32), CEIL_DIV(N, 32), 1);
// 32 * 32 = 1024 thread per block
dim3 blockDim(32, 32, 1);
// launch the asynchronous execution of the kernel on the device
// The function call returns immediately on the host
sgemm_naive<<<gridDim, blockDim>>>(M, N, K, alpha, A, B, beta, C);
```

In CUDA, kernel code is written from a single-thread perspective, using `blockIdx` and `threadIdx` to identify the specific block and thread executing the code. This allows each thread to process different data for parallel computation.

```
__global__ void sgemm_naive(int M, int N, int K, float alpha, const float *A,
 const float *B, float beta, float *C) {
 // compute position in C that this thread is responsible for
 const uint x = blockIdx.x * blockDim.x + threadIdx.x;
 const uint y = blockIdx.y * blockDim.y + threadIdx.y;

 // `if` condition is necessary for when M or N aren't multiples of 32.
 if (x < M && y < N) {
 float tmp = 0.0;
 for (int i = 0; i < K; ++i) {
 tmp += A[x * K + i] * B[i * N + y];
 }
 }
}
```


```

 }
 // C = $\alpha \cdot (A @ B) + \beta \cdot C$
 C[x * N + y] = alpha * tmp + beta * C[x * N + y];
 }
}

```

### Visualization of Kernel

credit: siboehm.com

**Memory Access Pattern of the Naive Kernel** In the kernel, two threads in the same block with ThreadIds (0,0) and (0,1) load the same column of  $B$  but different rows of  $A$ . Assuming no caching, each thread must load  $2 \cdot 4092 + 1$  floats from global memory. With  $4092^2$  threads in total, this results in 548GB of memory traffic. The visualization below illustrates the memory access pattern for two example threads,  $A$  (red) and  $B$  (green). 

credit: siboehm.com

## 2. Kernel 5: Increasing Arithmetic Intensity via 2D Blocktiling

loop structure looks like this: 

credit: siboehm.com

In kernel 5, each thread computes an  $8 \times 8$  grid of elements in matrix  $C$ . Initially, all threads cooperate to populate the shared memory (SMEM) cache by loading multiple elements. This shared memory loading stage is crucial for efficient computation.

```

for (uint loadOffset = 0; loadOffset < BM; loadOffset += strideA) {
 As[(innerRowA + loadOffset) * BK + innerColA] =
 A[(innerRowA + loadOffset) * K + innerColA];
}
for (uint loadOffset = 0; loadOffset < BK; loadOffset += strideB) {
 Bs[(innerRowB + loadOffset) * BN + innerColB] =
 B[(innerRowB + loadOffset) * N + innerColB];
}
__syncthreads();

```

Once the SMEM cache is populated, each thread multiplies its relevant SMEM entries and accumulates the result into local registers. The outer loop over the input matrices remains unchanged, while the three inner loops handle the dot product and traverse the TN and TM dimensions.



The interesting parts of the code look like this:



The resulting performance is 16 TFLOPs, a 2x improvement. Memory access calculations show each thread now calculates 64 results ( $8 \times 8$ ) per thread. Memory access per result is:

- Global Memory (GMEM): K/64
- Shared Memory (SMEM): K/4 Performance is improving, but warp stalls due to memory pipeline congestion are still frequent. In kernel 6, two measures will be taken to address this: transposing matrices A to enable auto-vectorization of SMEM loads, and ensuring alignment of GMEM accesses for the compiler.

## Build OpenCV with DNN and CUDA for GPU-Accelerated Face Detection

GPU can be used in OpenCV to accelerate the speed of reading and processing images. The user created a script to verify that OpenCV can utilize a GPU-accelerated Caffe model for face detection. The script:

- Loads a pre-trained Caffe model for face detection.
- Sets OpenCV to use CUDA for GPU acceleration.
- Loads an image and prepares it for the neural network.
- Performs face detection using the GPU.
- Draws bounding boxes around detected faces.
- Displays the output image with the detected faces.

```
#include <opencv2/opencv.hpp>
#include <opencv2/dnn.hpp>
#include <opencv2/core/cuda.hpp>

int main() {
 // Load the pre-trained Caffe model for face detection
 std::string model = "deploy.prototxt";
 std::string weights = "res10_300x300_ssd_iter_140000.caffemodel";

 // Load the model using OpenCV's DNN module
 cv::dnn::Net net = cv::dnn::readNetFromCaffe(model, weights);
 net.setPreferableBackend(cv::dnn::DNN_BACKEND_CUDA);
 net.setPreferableTarget(cv::dnn::DNN_TARGET_CUDA);

 // Load an image to test face detection
 cv::Mat image = cv::imread("test.jpg");
 if (image.empty()) {
 std::cerr << "Error: Could not load image." << std::endl;
 return -1;
 }

 // Prepare the image for the network
 cv::Mat blob = cv::dnn::blobFromImage(cv::resize(image, cv::Size(300, 300)),
 1.0, cv::Size(300, 300), cv::Scalar(104.0, 177.0, 123.0));

 // Perform face detection
 net.setInput(blob);
 cv::Mat detections = net.forward();

 // Loop over the detections and draw bounding boxes
 for (int i = 0; i < detections.size[2]; ++i) {
 float confidence = detections.at<float>(0, 0, i, 2);
 if (confidence > 0.5) {
```

```
 int startX = static_cast<int>(detections.at<float>(0, 0, i, 3) *
image.cols);
 int startY = static_cast<int>(detections.at<float>(0, 0, i, 4) *
image.rows);
 int endX = static_cast<int>(detections.at<float>(0, 0, i, 5) *
image.cols);
 int endY = static_cast<int>(detections.at<float>(0, 0, i, 6) *
image.rows);
 cv::rectangle(image, cv::Point(startX, startY), cv::Point(endX, endY),
cv::Scalar(0, 255, 0), 2);
 }
}

// Display the output image
cv::imshow("Output", image);
cv::waitKey(0);

return 0;
}
```

Test Image:



## Reference

1. siboehm, December 2022, How to Optimize a CUDA Matmul Kernel for cuBLAS-like Performance: a Worklog [siboehm](#).
2. geekforgeek, Introduction to CUDA Programming, 14 Mar, 2023 [geekforgeek](#).
3. Amos Stailey-Young, Mar 1, 2024, Build OpenCV with DNN and CUDA for GPU-Accelerated Face Detection, [geekforgeek](#).