# GPU Programming Code Examples

This document provides practical code examples of GPU programming using different frameworks and for various applications.

## Table of Contents

1. [CUDA Examples](#cuda-examples)

2. [OpenCL Examples](#opencl-examples)

3. [Python with GPU Acceleration](#python-gpu)

4. [Deep Learning Examples](#deep-learning)

5. [Performance Comparison](#performance)

<a name="cuda-examples"></a>

## 1. CUDA Examples

### 1.1 Vector Addition

```cuda

#include <stdio.h>

#include <cuda\_runtime.h>

// CUDA Kernel for vector addition

\_\_global\_\_ void vectorAdd(float \*a, float \*b, float \*c, int n) {

// Calculate global thread ID

int id = blockIdx.x \* blockDim.x + threadIdx.x;

// Boundary check

if (id < n) {

c[id] = a[id] + b[id];

}

}

int main() {

// Vector size

int n = 1000000;

size\_t bytes = n \* sizeof(float);

// Host vectors

float \*h\_a, \*h\_b, \*h\_c;

// Device vectors

float \*d\_a, \*d\_b, \*d\_c;

// Allocate host memory

h\_a = (float\*)malloc(bytes);

h\_b = (float\*)malloc(bytes);

h\_c = (float\*)malloc(bytes);

// Initialize host vectors

for (int i = 0; i < n; i++) {

h\_a[i] = rand() / (float)RAND\_MAX;

h\_b[i] = rand() / (float)RAND\_MAX;

}

// Allocate device memory

cudaMalloc(&d\_a, bytes);

cudaMalloc(&d\_b, bytes);

cudaMalloc(&d\_c, bytes);

// Copy data from host to device

cudaMemcpy(d\_a, h\_a, bytes, cudaMemcpyHostToDevice);

cudaMemcpy(d\_b, h\_b, bytes, cudaMemcpyHostToDevice);

// Define grid and block dimensions

int blockSize = 256;

int gridSize = (n + blockSize - 1) / blockSize;

// Launch kernel

vectorAdd<<<gridSize, blockSize>>>(d\_a, d\_b, d\_c, n);

// Copy result back to host

cudaMemcpy(h\_c, d\_c, bytes, cudaMemcpyDeviceToHost);

// Verify result

for (int i = 0; i < n; i++) {

if (fabs(h\_a[i] + h\_b[i] - h\_c[i]) > 1e-5) {

fprintf(stderr, "Verification failed at element %d!\n", i);

break;

}

}

printf("Vector addition completed successfully\n");

// Free memory

cudaFree(d\_a);

cudaFree(d\_b);

cudaFree(d\_c);

free(h\_a);

free(h\_b);

free(h\_c);

return 0;

}

```

### 1.2 Matrix Multiplication

```cuda

#include <stdio.h>

#include <cuda\_runtime.h>

// CUDA Kernel for matrix multiplication

\_\_global\_\_ void matrixMul(float \*a, float \*b, float \*c, int width) {

// Calculate row and column indices

int row = blockIdx.y \* blockDim.y + threadIdx.y;

int col = blockIdx.x \* blockDim.x + threadIdx.x;

// Compute if within matrix bounds

if (row < width && col < width) {

float sum = 0.0f;

for (int i = 0; i < width; i++) {

sum += a[row \* width + i] \* b[i \* width + col];

}

c[row \* width + col] = sum;

}

}

int main() {

// Matrix dimensions

int width = 1024;

size\_t bytes = width \* width \* sizeof(float);

// Host matrices

float \*h\_a, \*h\_b, \*h\_c;

// Device matrices

float \*d\_a, \*d\_b, \*d\_c;

// Allocate host memory

h\_a = (float\*)malloc(bytes);

h\_b = (float\*)malloc(bytes);

h\_c = (float\*)malloc(bytes);

// Initialize host matrices

for (int i = 0; i < width \* width; i++) {

h\_a[i] = rand() / (float)RAND\_MAX;

h\_b[i] = rand() / (float)RAND\_MAX;

}

// Allocate device memory

cudaMalloc(&d\_a, bytes);

cudaMalloc(&d\_b, bytes);

cudaMalloc(&d\_c, bytes);

// Copy data from host to device

cudaMemcpy(d\_a, h\_a, bytes, cudaMemcpyHostToDevice);

cudaMemcpy(d\_b, h\_b, bytes, cudaMemcpyHostToDevice);

// Define grid and block dimensions

dim3 blockDim(16, 16);

dim3 gridDim((width + blockDim.x - 1) / blockDim.x,

(width + blockDim.y - 1) / blockDim.y);

// Launch kernel

matrixMul<<<gridDim, blockDim>>>(d\_a, d\_b, d\_c, width);

// Copy result back to host

cudaMemcpy(h\_c, d\_c, bytes, cudaMemcpyDeviceToHost);

printf("Matrix multiplication completed successfully\n");

// Free memory

cudaFree(d\_a);

cudaFree(d\_b);

cudaFree(d\_c);

free(h\_a);

free(h\_b);

free(h\_c);

return 0;

}

```

<a name="opencl-examples"></a>

## 2. OpenCL Examples

### 2.1 Vector Addition in OpenCL

```c

#include <stdio.h>

#include <stdlib.h>

#include <CL/cl.h>

// OpenCL kernel source

const char \*kernelSource =

"\_\_kernel void vectorAdd(\_\_global const float \*a, \n" \

" \_\_global const float \*b, \n" \

" \_\_global float \*c, \n" \

" const int n) \n" \

"{ \n" \

" // Get global thread ID \n" \

" int id = get\_global\_id(0); \n" \

" \n" \

" // Boundary check \n" \

" if (id < n) { \n" \

" c[id] = a[id] + b[id]; \n" \

" } \n" \

"} \n";

int main() {

// Vector size

int n = 1000000;

size\_t bytes = n \* sizeof(float);

// Host vectors

float \*h\_a = (float\*)malloc(bytes);

float \*h\_b = (float\*)malloc(bytes);

float \*h\_c = (float\*)malloc(bytes);

// Initialize host vectors

for (int i = 0; i < n; i++) {

h\_a[i] = rand() / (float)RAND\_MAX;

h\_b[i] = rand() / (float)RAND\_MAX;

}

// OpenCL variables

cl\_platform\_id platform;

cl\_device\_id device;

cl\_context context;

cl\_command\_queue queue;

cl\_program program;

cl\_kernel kernel;

cl\_mem d\_a, d\_b, d\_c;

// Get platform and device

clGetPlatformIDs(1, &platform, NULL);

clGetDeviceIDs(platform, CL\_DEVICE\_TYPE\_GPU, 1, &device, NULL);

// Create context and command queue

context = clCreateContext(NULL, 1, &device, NULL, NULL, NULL);

queue = clCreateCommandQueue(context, device, 0, NULL);

// Create and build program

program = clCreateProgramWithSource(context, 1, &kernelSource, NULL, NULL);

clBuildProgram(program, 1, &device, NULL, NULL, NULL);

// Create kernel

kernel = clCreateKernel(program, "vectorAdd", NULL);

// Create device buffers

d\_a = clCreateBuffer(context, CL\_MEM\_READ\_ONLY, bytes, NULL, NULL);

d\_b = clCreateBuffer(context, CL\_MEM\_READ\_ONLY, bytes, NULL, NULL);

d\_c = clCreateBuffer(context, CL\_MEM\_WRITE\_ONLY, bytes, NULL, NULL);

// Copy data to device

clEnqueueWriteBuffer(queue, d\_a, CL\_TRUE, 0, bytes, h\_a, 0, NULL, NULL);

clEnqueueWriteBuffer(queue, d\_b, CL\_TRUE, 0, bytes, h\_b, 0, NULL, NULL);

// Set kernel arguments

clSetKernelArg(kernel, 0, sizeof(cl\_mem), &d\_a);

clSetKernelArg(kernel, 1, sizeof(cl\_mem), &d\_b);

clSetKernelArg(kernel, 2, sizeof(cl\_mem), &d\_c);

clSetKernelArg(kernel, 3, sizeof(int), &n);

// Execute kernel

size\_t globalSize = n;

size\_t localSize = 256;

clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &globalSize, &localSize, 0, NULL, NULL);

// Copy result back to host

clEnqueueReadBuffer(queue, d\_c, CL\_TRUE, 0, bytes, h\_c, 0, NULL, NULL);

// Verify result

for (int i = 0; i < n; i++) {

if (fabs(h\_a[i] + h\_b[i] - h\_c[i]) > 1e-5) {

fprintf(stderr, "Verification failed at element %d!\n", i);

break;

}

}

printf("Vector addition completed successfully\n");

// Clean up

clReleaseMemObject(d\_a);

clReleaseMemObject(d\_b);

clReleaseMemObject(d\_c);

clReleaseKernel(kernel);

clReleaseProgram(program);

clReleaseCommandQueue(queue);

clReleaseContext(context);

free(h\_a);

free(h\_b);

free(h\_c);

return 0;

}

```

<a name="python-gpu"></a>

## 3. Python with GPU Acceleration

### 3.1 NumPy with CuPy

```python

import numpy as np

import cupy as cp

import time

# Vector size

n = 10000000

# CPU implementation

def vector\_add\_cpu(a, b):

return a + b

# Create random vectors

a\_cpu = np.random.random(n).astype(np.float32)

b\_cpu = np.random.random(n).astype(np.float32)

# CPU timing

start\_time = time.time()

c\_cpu = vector\_add\_cpu(a\_cpu, b\_cpu)

cpu\_time = time.time() - start\_time

print(f"CPU time: {cpu\_time:.6f} seconds")

# GPU implementation

# Transfer data to GPU

a\_gpu = cp.asarray(a\_cpu)

b\_gpu = cp.asarray(b\_cpu)

# GPU timing

start\_time = time.time()

c\_gpu = a\_gpu + b\_gpu

cp.cuda.Stream.null.synchronize() # Ensure GPU operations complete

gpu\_time = time.time() - start\_time

print(f"GPU time: {gpu\_time:.6f} seconds")

# Verify results

c\_from\_gpu = cp.asnumpy(c\_gpu)

np.testing.assert\_allclose(c\_cpu, c\_from\_gpu, rtol=1e-5)

print(f"Results match! Speedup: {cpu\_time/gpu\_time:.2f}x")

```

<a name="deep-learning"></a>

## 4. Deep Learning Examples

### 4.1 PyTorch GPU Acceleration

```python

import torch

import time

import torchvision

import torchvision.transforms as transforms

from torch import nn, optim

from torch.utils.data import DataLoader

# Check if GPU is available

device = torch.device("cuda:0" if torch.cuda.is\_available() else "cpu")

print(f"Using device: {device}")

# Define a simple CNN

class SimpleCNN(nn.Module):

def \_\_init\_\_(self):

super(SimpleCNN, self).\_\_init\_\_()

self.conv1 = nn.Conv2d(1, 32, 3, 1)

self.conv2 = nn.Conv2d(32, 64, 3, 1)

self.dropout1 = nn.Dropout2d(0.25)

self.dropout2 = nn.Dropout2d(0.5)

self.fc1 = nn.Linear(9216, 128)

self.fc2 = nn.Linear(128, 10)

self.relu = nn.ReLU()

self.max\_pool = nn.MaxPool2d(2)

self.log\_softmax = nn.LogSoftmax(dim=1)

def forward(self, x):

x = self.relu(self.conv1(x))

x = self.relu(self.conv2(x))

x = self.max\_pool(x)

x = self.dropout1(x)

x = torch.flatten(x, 1)

x = self.relu(self.fc1(x))

x = self.dropout2(x)

x = self.fc2(x)

return self.log\_softmax(x)

# Load MNIST dataset

transform = transforms.Compose([

transforms.ToTensor(),

transforms.Normalize((0.1307,), (0.3081,))

])

train\_dataset = torchvision.datasets.MNIST('./data', train=True, download=True, transform=transform)

test\_dataset = torchvision.datasets.MNIST('./data', train=False, transform=transform)

train\_loader = DataLoader(train\_dataset, batch\_size=64, shuffle=True)

test\_loader = DataLoader(test\_dataset, batch\_size=1000)

# Initialize model, loss function, and optimizer

model = SimpleCNN().to(device)

criterion = nn.CrossEntropyLoss()

optimizer = optim.Adam(model.parameters(), lr=0.001)

# Training function

def train(model, device, train\_loader, optimizer, epoch):

model.train()

for batch\_idx, (data, target) in enumerate(train\_loader):

data, target = data.to(device), target.to(device)

optimizer.zero\_grad()

output = model(data)

loss = criterion(output, target)

loss.backward()

optimizer.step()

if batch\_idx % 100 == 0:

print(f'Train Epoch: {epoch} [{batch\_idx \* len(data)}/{len(train\_loader.dataset)} '

f'({100. \* batch\_idx / len(train\_loader):.0f}%)]\tLoss: {loss.item():.6f}')

# Test function

def test(model, device, test\_loader):

model.eval()

test\_loss = 0

correct = 0

with torch.no\_grad():

for data, target in test\_loader:

data, target = data.to(device), target.to(device)

output = model(data)

test\_loss += criterion(output, target).item()

pred = output.argmax(dim=1, keepdim=True)

correct += pred.eq(target.view\_as(pred)).sum().item()

test\_loss /= len(test\_loader.dataset)

print(f'\nTest set: Average loss: {test\_loss:.4f}, '

f'Accuracy: {correct}/{len(test\_loader.dataset)} '

f'({100. \* correct / len(test\_loader.dataset):.0f}%)\n')

# Train and test the model

start\_time = time.time()

for epoch in range(1, 3): # Just 2 epochs for demonstration

train(model, device, train\_loader, optimizer, epoch)

test(model, device, test\_loader)

total\_time = time.time() - start\_time

print(f"Total training time: {total\_time:.2f} seconds")

```

<a name="performance"></a>

## 5. Performance Comparison

The following table shows typical performance improvements when moving from CPU to GPU implementation:

| Application | CPU Performance | GPU Performance | Speedup Factor |

|-------------|----------------|-----------------|----------------|

| Vector Addition (1M elements) | 5.2 ms | 0.3 ms | 17x |

| Matrix Multiplication (1024x1024) | 2.3 s | 45 ms | 51x |

| Neural Network Training (MNIST) | 45 min | 2 min | 22.5x |

| Image Processing (4K image) | 320 ms | 12 ms | 26.7x |

| Monte Carlo Simulation (1M paths) | 8.5 s | 180 ms | 47x |

Note: Performance numbers are representative and will vary based on specific hardware configurations.