Fintel

Generating GPU-Accelerated Quantitative Finance Signals

```
# Install some magic to make c++ programs look nice!
!wget -O cpp_plugin.py https://gist.github.com/akshaykhadse/7acc91dd41f52944c6150754e5530c4b/raw/cpp_plugin.py
%load_ext cpp_plugin
```

```
--2023-05-02 08:02:37--  https://gist.github.com/akshaykhadse/7acc91dd41f52944c6150754e5530c4b/raw/cpp_plugin.py
Resolving gist.github.com (gist.github.com)... 140.82.114.3
Connecting to gist.github.com (gist.github.com)|140.82.114.3|:443... connected.
HTTP request sent, awaiting response... 301 Moved Permanently
Location: https://gist.githubusercontent.com/akshaykhadse/7acc91dd41f52944c6150754e5530c4b/raw/cpp_plugin.py [following]
--2023-05-02 08:02:37--  https://gist.githubusercontent.com/akshaykhadse/7acc91dd41f52944c6150754e5530c4b/raw/cpp_plugin.py
Resolving gist.githubusercontent.com (gist.githubusercontent.com)... 185.199.108.133, 185.199.109.133, 185.199.110.133, ...
Connecting to gist.githubusercontent.com (gist.githubusercontent.com)|185.199.108.133|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 2730 (2.7K) [text/plain]
Saving to: ‘cpp_plugin.py’

cpp_plugin.py       100%[===================>]   2.67K  --.-KB/s    in 0s

2023-05-02 08:02:38 (39.6 MB/s) - ‘cpp_plugin.py’ saved [2730/2730]

The cpp_plugin extension is already loaded. To reload it, use:
  %reload_ext cpp_plugin
```

```
# make sure CUDA is installed
!nvcc --version
```

```
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2022 NVIDIA Corporation
Built on Wed_Sep_21_10:33:58_PDT_2022
Cuda compilation tools, release 11.8, V11.8.89
Build cuda_11.8.r11.8/compiler.31833905_0
```

```
# make sure you have a GPU runtime (if this fails go to runtime -> change runtime type)
!nvidia-smi
```

```
Tue May  2 08:02:38 2023
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 525.85.12    Driver Version: 525.85.12    CUDA Version: 12.0     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla T4            Off  | 00000000:00:04.0 Off |                    0 |
| N/A   46C    P8    11W /  70W |      0MiB / 15360MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

Download stock price data

```
!pip install yfinance > /dev/null 2>&1
```

```
import yfinance as yf

# Define the stock symbol and the date range for which you want to download the data
symbol = "MSFT"
start_date = "1986-03-12"
end_date = "2023-01-01"

# Download historical stock data using yfinance
```

```
stock_data = yf.download(symbol, start=start_date, end=end_date)

# Save the downloaded data as a CSV file
csv_filename = "stock_prices.csv"
stock_data.to_csv(csv_filename)

print(f"Historical stock data for {symbol} saved to {csv_filename}")
```

```
[*********************100%***********************]  1 of 1 completed
Historical stock data for MSFT saved to stock_prices.csv
```

The Interface

The output of the cell is formatted text double click the title to show or hide the raw code you can edit!

```
#@title The output of the cell is formatted text double click the title to show or hide the raw code you can edit!
%%cpp -n IFintel.h -s xcode

#ifndef IFintel_H
#define IFintel_H

#include <vector>
#include <string>

class IFintel
{
public:
    struct DataPoint
    {
        std::string date;
        float open;
        float high;
        float low;
        float close;
    };
    virtual ~IFintel(){};
    virtual std::vector<float> calculate_sma(const std::vector<DataPoint>& data, int window_size) = 0;
    virtual std::vector<float> calculate_ema(const std::vector<DataPoint>& data, int window_size) = 0;
    virtual std::vector<float> calculate_rsi(const std::vector<DataPoint>& data, int window_size) = 0;
};

#endif
```


```
#ifndef IFintel_H
#define IFintel_H

#include <vector>
#include <string>

class IFintel
{
public:
    struct DataPoint
    {
        std::string date;
        float open;
        float high;
        float low;
        float close;
    };
    virtual ~IFintel(){};
    virtual std::vector<float> calculate_sma(const std::vector<DataPoint>& data, int wi
    virtual std::vector<float> calculate_ema(const std::vector<DataPoint>& data, int wi
    virtual std::vector<float> calculate_rsi(const std::vector<DataPoint>& data, int wi
};
```


The CPU Implementation

Header File

```
#@title Header File
%%cpp -n CPUFintel.h -s xcode

#ifndef CPUFINTEL_H
#define CPUFINTEL_H

#include "IFintel.h"

class CPUFintel : public IFintel
{
public:
    CPUFintel(){};
    std::vector<float> calculate_sma(const std::vector<DataPoint>& data, int window_size);
    std::vector<float> calculate_ema(const std::vector<DataPoint>& data, int window_size);
    std::vector<float> calculate_rsi(const std::vector<DataPoint>& data, int window_size);
    ~CPUFintel(){};
};

#endif
```


```
#ifndef CPUFINTEL_H
#define CPUFINTEL_H

#include "IFintel.h"

class CPUFintel : public IFintel
{
public:
    CPUFintel(){};
    std::vector<float> calculate_sma(const std::vector<DataPoint>& data, int window_siz
    std::vector<float> calculate_ema(const std::vector<DataPoint>& data, int window_siz
    std::vector<float> calculate_rsi(const std::vector<DataPoint>& data, int window_siz
    ~CPUFintel(){};
};
```


Implementation

```
#@title Implementation
%%cpp -n CPUFintel.cpp -s xcode

#include "CPUFintel.h"

std::vector<float> CPUFintel::calculate_rsi(const std::vector<DataPoint>& data, int window_size) {
    std::vector<float> rsi(data.size());

    float up_sum = 0;
    float down_sum = 0;

    for (int i = 1; i < window_size; i++) {
        float delta = data[i].close - data[i-1].close;
        if (delta > 0) {
            up_sum += delta;
        } else {
            down_sum -= delta;
        }
    }

    float avg_up = up_sum / window_size;
    float avg_down = down_sum / window_size;
    float rs = avg_up / avg_down;
    rsi[window_size] = 100 - (100 / (1 + rs));

    for (unsigned int i = window_size + 1; i < data.size(); i++) {
        float delta = data[i].close - data[i-1].close;
        if (delta > 0) {
            up_sum += delta;
            down_sum -= 0;
        } else {
            down_sum -= delta;
            up_sum -= 0;
        }

        avg_up = up_sum / window_size;
        avg_down = down_sum / window_size;
```

```cpp
        rs = avg_up / avg_down;
        rsi[i] = 100 - (100 / (1 + rs));
    }

    return rsi;
}

std::vector<float> CPUFintel::calculate_ema(const std::vector<DataPoint>& data, int window_size)
{
    std::vector<float> ema(data.size());

    float multiplier = 2.0f / (window_size + 1);
    ema[0] = data[0].close;

    for (unsigned int i = 1; i < data.size(); i++) {
        ema[i] = (data[i].close - ema[i - 1]) * multiplier + ema[i - 1];
    }

    return ema;
}

std::vector<float> CPUFintel::calculate_sma(const std::vector<DataPoint>& data, int window_size)
{
    std::vector<float> sma(data.size());

    float sum = 0;
    for (int i = 0; i < window_size; i++) {
        sum += data[i].close;
        sma[i] = sum / (i + 1);
    }

    for (unsigned int i = window_size; i < data.size(); i++) {
        sum += data[i].close - data[i - window_size].close;
        sma[i] = sum / window_size;
    }

    return sma;
}
```

```
        }

        float avg_up = up_sum / window_size;
        float avg_down = down_sum / window_size;
        float rs = avg_up / avg_down;
        rsi[window_size] = 100 - (100 / (1 + rs));

        for (unsigned int i = window_size + 1; i < data.size(); i++) {
            float delta = data[i].close - data[i-1].close;
            if (delta > 0) {
                up_sum += delta;
                down_sum -= 0;
            } else {
                down_sum -= delta;
                up_sum -= 0;
            }

            avg_up = up_sum / window_size;
            avg_down = down_sum / window_size;
            rs = avg_up / avg_down;
            rsi[i] = 100 - (100 / (1 + rs));
        }

        return rsi;
    }

    std::vector<float> CPUFintel::calculate_ema(const std::vector<DataPoint>& data, int
    {
        std::vector<float> ema(data.size());

        float multiplier = 2.0f / (window_size + 1);
        ema[0] = data[0].close;
```

The Parallelized CPU Implementation

```
        return ema;
```

Header File

```
#@title Header File
%%cpp -n ParallelCPUFintel.h -s xcode

#ifndef PARALLELCPUFINTEL_H
#define PARALLELCPUFINTEL_H

#include "CPUFintel.h"
#include <thread>
#include <mutex>

class ParallelCPUFintel : public CPUFintel
{
public:
    ParallelCPUFintel(){};
    std::vector<float> calculate_sma(const std::vector<DataPoint>& data, int window_size) override;
    std::vector<float> calculate_ema(const std::vector<DataPoint>& data, int window_size) override;
    std::vector<float> calculate_rsi(const std::vector<DataPoint>& data, int window_size) override;
    ~ParallelCPUFintel(){};

private:
    void sma_worker(const std::vector<DataPoint>& data, int window_size, size_t start, size_t end, std::vector<float>& result, std::mutex& re
    void ema_worker(const std::vector<DataPoint>& data, int window_size, size_t start, size_t end, std::vector<float>& result, std::mutex& re
    void rsi_worker(const std::vector<DataPoint>& data, int window_size, size_t start, size_t end, std::vector<float>& result, std::mutex& re

};

#endif
```

```
#ifndef PARALLELCPUFINTEL_H
#define PARALLELCPUFINTEL_H

#include "CPUFintel.h"
#include <thread>
#include <mutex>

class ParallelCPUFintel : public CPUFintel
{
public:
    ParallelCPUFintel(){};
    std::vector<float> calculate_sma(const std::vector<DataPoint>& data, int window_size
    std::vector<float> calculate_ema(const std::vector<DataPoint>& data, int window_size
```

Implementation

```
#@title Implementation
%%cpp -n ParallelCPUFintel.cpp -s xcode

#include "ParallelCPUFintel.h"

std::vector<float> ParallelCPUFintel::calculate_sma(const std::vector<DataPoint>& data, int window_size) {
    std::vector<float> sma(data.size());
    std::mutex sma_mutex;

    size_t num_threads = std::thread::hardware_concurrency();
    size_t chunk_size = data.size() / num_threads;

    std::vector<std::thread> threads;

    for (size_t i = 0; i < num_threads; i++) {
        size_t start = i * chunk_size;
        size_t end = (i == num_threads - 1) ? data.size() : start + chunk_size;
        threads.emplace_back(&ParallelCPUFintel::sma_worker, this, std::ref(data), window_size, start, end, std::ref(sma), std::ref(sma_mute
    }

    for (std::thread& t : threads) {
        t.join();
    }

    return sma;
}

std::vector<float> ParallelCPUFintel::calculate_ema(const std::vector<DataPoint>& data, int window_size)
{
    std::vector<float> ema(data.size());
    std::mutex ema_mutex;

    size_t num_threads = std::thread::hardware_concurrency();
    size_t chunk_size = data.size() / num_threads;

    std::vector<std::thread> threads;

    for (size_t i = 0; i < num_threads; i++) {
        size_t start = i * chunk_size;
        size_t end = (i == num_threads - 1) ? data.size() : start + chunk_size;
        threads.emplace_back(&ParallelCPUFintel::ema_worker, this, std::ref(data), window_size, start, end, std::ref(ema), std::ref(ema_mute
    }

    for (std::thread& t : threads) {
        t.join();
    }

    return ema;
 }

std::vector<float> ParallelCPUFintel::calculate_rsi(const std::vector<DataPoint>& data, int window_size)
{
    // Not implemented because SMA and RSI were very slow anyways
    std::vector<float> rsi(data.size());
    return rsi;
}

void ParallelCPUFintel::sma_worker(const std::vector<DataPoint>& data, int window_size, size_t start, size_t end, std::vector<float>& result,
    float sum = 0;
```

```
    if (start == 0) {
        for (int i = 0; i < window_size; i++) {
            sum += data[i].close;
            result[i] = sum / (i + 1);
        }
        start = window_size;
    } else {
        for (size_t i = start - window_size; i < start; i++) {
            sum += data[i].close;
        }
    }

    for (size_t i = start; i < end; i++) {
        sum += data[i].close - data[i - window_size].close;
        float sma_value = sum / window_size;

        std::lock_guard<std::mutex> lock(result_mutex);
        result[i] = sma_value;
    }
}

void ParallelCPUFintel::ema_worker(const std::vector<DataPoint>& data, int window_size, size_t start, size_t end, std::vector<float>& result,
    float multiplier = 2.0f / (window_size + 1);
    float previous_ema;

    if (start == 0) {
        previous_ema = data[0].close;
        result[0] = previous_ema;
        start = 1;
    } else {
        std::lock_guard<std::mutex> lock(result_mutex);
        previous_ema = result[start - 1];
    }

    for (size_t i = start; i < end; i++) {
        float current_ema = (data[i].close - previous_ema) * multiplier + previous_ema;
        previous_ema = current_ema;

        std::lock_guard<std::mutex> lock(result_mutex);
        result[i] = current_ema;
    }
}

void ParallelCPUFintel::rsi_worker(const std::vector<DataPoint>& data, int window_size, size_t start, size_t end, std::vector<float>& result,
// Not implemented because SMA and RSI were very slow anyways
}
```

```cpp
std::vector<float> ParallelCPUFintel::calculate_rsi(const std::vector<DataPoint>& da
{
    // Not implemented because SMA and RSI were very slow anyways
    std::vector<float> rsi(data.size());
    return rsi;
}

void ParallelCPUFintel::sma_worker(const std::vector<DataPoint>& data, int window_si
    float sum = 0;

    if (start == 0) {
        for (int i = 0; i < window_size; i++) {
            sum += data[i].close;
            result[i] = sum / (i + 1);
        }
        start = window_size;
    } else {
        for (size_t i = start - window_size; i < start; i++) {
            sum += data[i].close;
        }
    }

    for (size_t i = start; i < end; i++) {
        sum += data[i].close - data[i - window_size].close;
        float sma_value = sum / window_size;

        std::lock_guard<std::mutex> lock(result_mutex);
        result[i] = sma_value;
    }
}

void ParallelCPUFintel::ema_worker(const std::vector<DataPoint>& data, int window_si
    float multiplier = 2.0f / (window_size + 1);
    float previous_ema;

    if (start == 0) {
        previous_ema = data[0].close;
        result[0] = previous_ema;
        start = 1;
    } else {
        std::lock_guard<std::mutex> lock(result_mutex);
        previous_ema = result[start - 1];
    }

    for (size_t i = start; i < end; i++) {
        float current_ema = (data[i].close - previous_ema) * multiplier + previous_e
        previous_ema = current_ema;

        std::lock_guard<std::mutex> lock(result_mutex);
        result[i] = current_ema;
    }
}

void ParallelCPUFintel::rsi_worker(const std::vector<DataPoint>& data, int window_si
// Not implemented because SMA and RSI were very slow anyways
```

The GPU Implementation

Header File

```
#@title Header File
%%cpp -n GPUFintel.h -s xcode

#ifndef GPUFINTEL_H
#define GPUFINTEL_H

#include "IFintel.h"

class GPUFintel : public IFintel
{
public:
    GPUFintel(){};
    std::vector<float> calculate_sma(const std::vector<DataPoint>& data, int window_size);
    std::vector<float> calculate_ema(const std::vector<DataPoint>& data, int window_size);
    std::vector<float> calculate_rsi(const std::vector<DataPoint>& data, int window_size);
    ~GPUFintel(){};

private:
    template <typename KernelFunc> std::vector<float> calculate_indicator(const std::vector<DataPoint>& data, int window_size, KernelFunc ker

};

#endif
```


```
#ifndef GPUFINTEL_H
#define GPUFINTEL_H

#include "IFintel.h"

class GPUFintel : public IFintel
{
public:
    GPUFintel(){};
    std::vector<float> calculate_sma(const std::vector<DataPoint>& data, int window_size
    std::vector<float> calculate_ema(const std::vector<DataPoint>& data, int window_size
    std::vector<float> calculate_rsi(const std::vector<DataPoint>& data, int window_size
    ~GPUFintel(){};

private:
    template <typename KernelFunc> std::vector<float> calculate_indicator(const std::ve

};

#endif
```


Header File

```
#@title Header File
%%cpp -n gpu_helpers.h -s xcode

#ifndef GPU_HELPERS_H
#define GPU_HELPERS_H

#include <cuda_runtime.h>

__device__ void sma_function(float *input, float *output, int window_size, int data_size, int idx);
__device__ void ema_function(float *input, float *output, int window_size, int data_size, int idx);
__device__ void rsi_function(float *input, float *output, int window_size, int data_size, int idx);

#endif
```


```
#ifndef GPU_HELPERS_H
#define GPU_HELPERS_H

#include <cuda_runtime.h>

__device__ void sma_function(float *input, float *output, int window_size, int data_size
__device__ void ema_function(float *input, float *output, int window_size, int data_size
__device__ void rsi_function(float *input, float *output, int window_size, int data_size
```


Implementation

```
#@title Implementation
%%writefile GPUFintel.cu

#include "GPUFintel.h"
#include "gpu_helpers.h"

#include <cuda_runtime.h>
#include <device_launch_parameters.h>

__global__ void sma_kernel(float *input, float *output, int window_size, int data_size) {
    sma_function(input, output, window_size, data_size, threadIdx.x + blockIdx.x * blockDim.x);
}

__global__ void ema_kernel(float *input, float *output, int window_size, int data_size) {
    ema_function(input, output, window_size, data_size, threadIdx.x + blockIdx.x * blockDim.x);
}

__global__ void rsi_kernel(float *input, float *output, int window_size, int data_size) {
    rsi_function(input, output, window_size, data_size, threadIdx.x + blockIdx.x * blockDim.x);
}

template <typename KernelFunc>
std::vector<float> GPUFintel::calculate_indicator(const std::vector<DataPoint>& data, int window_size, KernelFunc kernel)
{
    int data_size = data.size();
    float *h_input = new float[data_size];
    float *h_output = new float[data_size];

    for (int i = 0; i < data_size; i++) {
        h_input[i] = data[i].close;
    }

    float *d_input, *d_output;
    cudaMalloc((void **)&d_input, data_size * sizeof(float));
    cudaMalloc((void **)&d_output, data_size * sizeof(float));

    cudaMemcpy(d_input, h_input, data_size * sizeof(float), cudaMemcpyHostToDevice);

    int blockSize = 256;
    int gridSize = (data_size + blockSize - 1) / blockSize;

    kernel<<<gridSize, blockSize>>>(d_input, d_output, window_size, data_size);

    cudaMemcpy(h_output, d_output, data_size * sizeof(float), cudaMemcpyDeviceToHost);

    cudaFree(d_input);
    cudaFree(d_output);

    std::vector<float> result(h_output, h_output + data_size);

    delete[] h_input;
    delete[] h_output;

    return result;
}

std::vector<float> GPUFintel::calculate_sma(const std::vector<DataPoint>& data, int window_size)
{
    return calculate_indicator(data, window_size, sma_kernel);
}

std::vector<float> GPUFintel::calculate_ema(const std::vector<DataPoint>& data, int window_size)
{
    return calculate_indicator(data, window_size, ema_kernel);
}

std::vector<float> GPUFintel::calculate_rsi(const std::vector<DataPoint>& data, int window_size)
{
    return calculate_indicator(data, window_size, rsi_kernel);
}
```

```
Writing GPUFintel.cu
```

Implementation

```
#@title Implementation
%%writefile gpu_helpers.cu

#include "gpu_helpers.h"


__device__ void sma_function(float *input, float *output, int window_size, int data_size, int idx) {
    if (idx < data_size) {
        float sum = 0.0f;
        int count = 0;
        for (int i = idx - window_size + 1; i <= idx; i++) {
            if (i >= 0) {
                sum += input[i];
                count++;
            }
        }
        output[idx] = sum / count;
    }
}

__device__ void ema_function(float *input, float *output, int window_size, int data_size, int idx) {
    if (idx < data_size) {
        float alpha = 2.0f / (window_size + 1.0f);
        if (idx == 0) {
            output[idx] = input[idx];
        } else {
            output[idx] = alpha * input[idx] + (1.0f - alpha) * output[idx - 1];
        }
    }
}

__device__ void rsi_function(float *input, float *output, int window_size, int data_size, int idx) {
    if (idx >= window_size && idx < data_size) {
        float gain_sum = 0.0f;
        float loss_sum = 0.0f;
        for (int i = idx - window_size + 1; i <= idx; i++) {
            float diff = input[i] - input[i-1];
            if (diff > 0) {
                gain_sum += diff;
            } else if (diff < 0) {
                loss_sum += abs(diff);
            }
        }
        float rs = gain_sum / window_size / (loss_sum / window_size);
        output[idx] = 100.0f - 100.0f / (1.0f + rs);
    }
}
```

```
Writing gpu_helpers.cu
```

Header File

```
#@title Header File
%%cpp -n TimedGPUFintel.h -s xcode

#ifndef TIMEDGPUFINTEL_H
#define TIMEDGPUFINTEL_H

#include "IFintel.h"

class TimedGPUFintel : public IFintel {
public:
    TimedGPUFintel(){};
    std::vector<float> calculate_sma(const std::vector<DataPoint>& data, int window_size);
    std::vector<float> calculate_ema(const std::vector<DataPoint>& data, int window_size);
    std::vector<float> calculate_rsi(const std::vector<DataPoint>& data, int window_size);
    ~TimedGPUFintel(){};

private:
  template <typename KernelFunc, typename KernelFunc2> std::vector<float> calculate_indicator(const std::vector<DataPoint>& data, int window_
};

#endif
```

```
#ifndef TIMEDGPUFINTEL_H
#define TIMEDGPUFINTEL_H

#include "IFintel.h"

class TimedGPUFintel : public IFintel {
public:
    TimedGPUFintel(){};
    std::vector<float> calculate_sma(const std::vector<DataPoint>& data, int window_size
    std::vector<float> calculate_ema(const std::vector<DataPoint>& data, int window_size
    std::vector<float> calculate_rsi(const std::vector<DataPoint>& data, int window_size
    ~TimedGPUFintel(){};

private:
  template <typename KernelFunc, typename KernelFunc2> std::vector<float> calculate_ind
};

#endif
```

Implementation

```
#@title Implementation
%%writefile TimedGPUFintel.cu

#include "gpu_helpers.h"
#include "TimedGPUFintel.h"

#include <iostream>
#include <numeric>
#include <cuda_runtime.h>
#include <device_launch_parameters.h>

#define time_delta_us_timespec(start,end) (1e6*static_cast<double>(end.tv_sec - start.tv_sec)+1e-3*static_cast<double>(end.tv_nsec - start.tv
#define TEST_ITERS_GLOBAL 100

void printStats(std::vector<double> *times){
   double sum = std::accumulate(times->begin(), times->end(), 0.0);
   double mean = sum/static_cast<double>(times->size());
   std::cout << "Mean: " << mean << "\n";
}

__global__ void sma_kernel_notime(float *input, float *output, int window_size, int data_size) {
    sma_function(input, output, window_size, data_size, threadIdx.x + blockIdx.x * blockDim.x);
}

__global__ void ema_kernel_notime(float *input, float *output, int window_size, int data_size) {
    ema_function(input, output, window_size, data_size, threadIdx.x + blockIdx.x * blockDim.x);
}

__global__ void rsi_kernel_notime(float *input, float *output, int window_size, int data_size) {
    rsi_function(input, output, window_size, data_size, threadIdx.x + blockIdx.x * blockDim.x);
}


__global__
void sma_kernel_timing(float *input, float *output, int window_size, int data_size, int TEST_ITERS) {
    for(int iter = 0; iter < TEST_ITERS; iter++){
      sma_function(input, output, window_size, data_size, threadIdx.x + blockIdx.x * blockDim.x);
    }
}

__global__
void ema_kernel_timing(float *input, float *output, int window_size, int data_size, int TEST_ITERS) {
    for(int iter = 0; iter < TEST_ITERS; iter++){
      ema_function(input, output, window_size, data_size, threadIdx.x + blockIdx.x * blockDim.x);
    }
}

__global__
void rsi_kernel_timing(float *input, float *output, int window_size, int data_size, int TEST_ITERS) {
    for(int iter = 0; iter < TEST_ITERS; iter++){
      rsi_function(input, output, window_size, data_size, threadIdx.x + blockIdx.x * blockDim.x);
    }
}

template <typename KernelFunc>
```

```
__host__ void end_to_end_kernel_with_memory_test(float *h_input, float *h_output, float *d_input, float *d_output, int window_size, int data_
    cudaMemcpy(d_input, h_input, data_size * sizeof(float), cudaMemcpyHostToDevice);

    int blockSize = 256;
    int gridSize = (data_size + blockSize - 1) / blockSize;

    kernel<<<gridSize, blockSize>>>(d_input, d_output, window_size, data_size);

    cudaMemcpy(h_output, d_output, data_size * sizeof(float), cudaMemcpyDeviceToHost);
}

template <typename KernelFunc>
__host__ void end_to_end_kernel_no_memory_test(float *d_input, float *d_output, int window_size, int data_size, KernelFunc kernel){
  int blockSize = 256;
  int gridSize = (data_size + blockSize - 1) / blockSize;
    kernel<<<gridSize, blockSize>>>(d_input, d_output, window_size, data_size);
}

template <int TEST_ITERS, typename KernelFunc, typename KernelFunc2>
__host__
void test(int SWEEP_PARAMETER, float *h_input, float *h_output, float *d_input, float *d_output, int window_size, int data_size, KernelFunc k
      int blockSize = 256;
      int gridSize = (data_size + blockSize - 1) / blockSize;

      struct timespec start, end;
      std::vector<double> times = {};

      if (SWEEP_PARAMETER == 0) {
        clock_gettime(CLOCK_MONOTONIC,&start);
        kernel_timing<<<gridSize, blockSize>>>(d_input, d_output, window_size, data_size, TEST_ITERS);
        clock_gettime(CLOCK_MONOTONIC,&end);
        printf("[N:1]: Ignore Me -- Initial Slow One: %f\n",time_delta_us_timespec(start,end)/static_cast<double>(TEST_ITERS));

        clock_gettime(CLOCK_MONOTONIC,&start);
        kernel_timing<<<gridSize, blockSize>>>(d_input, d_output, window_size, data_size, TEST_ITERS);
        clock_gettime(CLOCK_MONOTONIC,&end);
        printf("[N:1]: Kernel Under Test: %f\n",time_delta_us_timespec(start,end)/static_cast<double>(TEST_ITERS));
      }
      else {
        for(int iter = 0; iter < TEST_ITERS; iter++){
          clock_gettime(CLOCK_MONOTONIC,&start);
          end_to_end_kernel_with_memory_test(h_input, h_output, d_input, d_output, window_size, data_size, kernel);
          clock_gettime(CLOCK_MONOTONIC,&end);
          times.push_back(time_delta_us_timespec(start,end));
        }
        printf("[N:%d]: Ignore Me -- Initial Slow One: ",SWEEP_PARAMETER); printStats(&times); times.clear();

        for(int iter = 0; iter < TEST_ITERS; iter++){
          clock_gettime(CLOCK_MONOTONIC,&start);
          end_to_end_kernel_with_memory_test(h_input, h_output, d_input, d_output, window_size, data_size, kernel);
          clock_gettime(CLOCK_MONOTONIC,&end);
          times.push_back(time_delta_us_timespec(start,end));
        }
        printf("[N:%d]: Kernel Under Test - With Memory: ",SWEEP_PARAMETER); printStats(&times); times.clear();

        for(int iter = 0; iter < TEST_ITERS; iter++){
          cudaMemcpy(d_input, h_input, data_size * sizeof(float), cudaMemcpyHostToDevice);
          clock_gettime(CLOCK_MONOTONIC,&start);
          end_to_end_kernel_no_memory_test(d_input, d_output, window_size, data_size, kernel);
          clock_gettime(CLOCK_MONOTONIC,&end);
          cudaMemcpy(h_output, d_output, data_size * sizeof(float), cudaMemcpyDeviceToHost);
          times.push_back(time_delta_us_timespec(start,end));
        }
        printf("[N:%d]: Kernel Under Test - Compute Only: ",SWEEP_PARAMETER); printStats(&times); times.clear();
      }
}

template <typename KernelFunc, typename KernelFunc2>
std::vector<float> TimedGPUFintel::calculate_indicator(const std::vector<DataPoint>& data, int window_size, KernelFunc kernel, KernelFunc2 ke
{
    int data_size = data.size();
    float *h_input = new float[data_size];
    float *h_output = new float[data_size];

    for (int i = 0; i < data_size; i++) {
        h_input[i] = data[i].close;
    }
```

```
    float *d_input, *d_output;
    cudaMalloc((void **)&d_input, data_size * sizeof(float));
    cudaMalloc((void **)&d_output, data_size * sizeof(float));

    test<TEST_ITERS_GLOBAL>(data_size, h_input, h_output, d_input, d_output, window_size, data_size, kernel, kernel_timing);

    cudaFree(d_input);
    cudaFree(d_output);

    std::vector<float> result(h_output, h_output + data_size);

    delete[] h_input;
    delete[] h_output;

    return result;
}

std::vector<float> TimedGPUFintel::calculate_sma(const std::vector<DataPoint>& data, int window_size)
{
    return calculate_indicator(data, window_size, sma_kernel_notime, sma_kernel_timing);
}

std::vector<float> TimedGPUFintel::calculate_ema(const std::vector<DataPoint>& data, int window_size)
{
    return calculate_indicator(data, window_size, ema_kernel_notime, ema_kernel_timing);
}

std::vector<float> TimedGPUFintel::calculate_rsi(const std::vector<DataPoint>& data, int window_size)
{
    return calculate_indicator(data, window_size, rsi_kernel_notime, rsi_kernel_timing);
}
```

```
Writing TimedGPUFintel.cu
```

Controller

Header

```
#@title Header
%%cpp -n Controller.h -s xcode

#ifndef CONTROLLER_H
#define CONTROLLER_H

class Controller
{
private:
    std::shared_ptr<IFintel> fintel;

public:
    Controller(std::shared_ptr<IFintel> fintel_) : fintel(fintel_){};
    void benchmark(std::string indicator);
    std::vector<IFintel::DataPoint> getData(const std::string& filename);
    std::vector<IFintel::DataPoint> getFakeData(int data_size);

};

#endif
```

```
#ifndef CONTROLLER_H
#define CONTROLLER_H
```

Implementation

```
#@title Implementation
%%cpp -n controller.cpp -s xcode

#include <iostream>
#include <memory>
#include <vector>
#include <fstream>
#include <sstream>
#include <random>

#include "IFintel.h"
#include "CPUFintel.h"
#include "ParallelCPUFintel.h"
#include "GPUFintel.h"
#include "TimedGPUFintel.h"
#include "Controller.h"

#define time_delta_us_timespec(start,end) (1e6*static_cast<double>(end.tv_sec - start.tv_sec)+1e-3*static_cast<double>(end.tv_nsec - start.tv


void Controller::benchmark(std::string indicator) {
    int window_size = 10;
    struct timespec start, end;
    int num = 100;
    {
      std::vector<IFintel::DataPoint> data = getFakeData(num);
      if (indicator == "sma")
        std::vector<float> sma = fintel->calculate_sma(data, window_size);
      else if (indicator == "ema")
        std::vector<float> ema = fintel->calculate_ema(data, window_size);
      else if (indicator == "rsi")
        std::vector<float> rsi = fintel->calculate_rsi(data, window_size);
    }
    while (num <= 10000000) {
      std::vector<IFintel::DataPoint> data = getFakeData(num);
      clock_gettime(CLOCK_MONOTONIC,&start);
      if (indicator == "sma")
        std::vector<float> sma = fintel->calculate_sma(data, window_size);
      else if (indicator == "ema")
        std::vector<float> ema = fintel->calculate_ema(data, window_size);
      else if (indicator == "rsi")
        std::vector<float> rsi = fintel->calculate_rsi(data, window_size);
      clock_gettime(CLOCK_MONOTONIC,&end);
      printf("[N:%d] E2E Timing: %f\n", num, time_delta_us_timespec(start,end)/static_cast<double>(1));
      num *= 2;
    }
    return;
}

std::vector<IFintel::DataPoint> Controller::getData(const std::string& filename) {
    std::vector<IFintel::DataPoint> data;
    std::ifstream file(filename);
    std::string line;

    if (file.is_open()) {
        std::getline(file, line);
        while (getline(file, line)) {
            std::stringstream ss(line);

            std::string date, open, high, low, close, adj_close, volume;
            getline(ss, date, ',');
            getline(ss, open, ',');
            getline(ss, high, ',');
            getline(ss, low, ',');
            getline(ss, close, ',');
            getline(ss, adj_close, ',');
            getline(ss, volume, ',');

            IFintel::DataPoint dp;
            dp.date = date;
```

```
            dp.open = std::stof(open);
            dp.high = std::stof(high);
            dp.low = std::stof(low);
            dp.close = std::stof(close);

            data.push_back(dp);
        }
        file.close();
    } else {
        std::cout << "Unable to open file: " << filename << std::endl;
    }

    return data;
}

std::vector<IFintel::DataPoint> Controller::getFakeData(int data_size) {
    std::vector<IFintel::DataPoint> data;
    std::random_device rd;
    std::default_random_engine generator(rd());
    std::uniform_real_distribution<double> distribution(0.1,10);

    for (int i = 0; i < data_size; i++) {
        IFintel::DataPoint dp;
        dp.date = "";
        dp.open = distribution(generator);
        dp.high = distribution(generator);
        dp.low = distribution(generator);
        dp.close = distribution(generator);
        data.push_back(dp);
    }
    return data;
}


int main() {
    std::cout << "Benchmarking CPU Implementation E2E\n";
    std::shared_ptr<IFintel> cpuFintel = std::make_shared<CPUFintel>();
    Controller c1(cpuFintel);
    std::cout << "SMA:\n";
    c1.benchmark("sma");
    std::cout << "EMA:\n";
    c1.benchmark("ema");
    std::cout << "RSI:\n";
    c1.benchmark("rsi");

    std::cout << "------------------------------------------------------------\n";

    std::cout << "Benchmarking Parallel CPU Implementation E2E\n";
    std::shared_ptr<IFintel> parallelCPUFintel = std::make_shared<ParallelCPUFintel>();
    Controller c5(parallelCPUFintel);
    std::cout << "SMA:\n";
    c5.benchmark("sma");
    std::cout << "EMA:\n";
    c5.benchmark("ema");
    std::cout << "RSI:\n";
    c5.benchmark("rsi");

    std::cout << "------------------------------------------------------------\n";


    std::cout << "Benchmarking GPU Implementation E2E (Slow one -- ignore)\n";
    std::shared_ptr<IFintel> gpuFintel = std::make_shared<GPUFintel>();
    Controller c2(gpuFintel);
    std::cout << "SMA:\n";
    c2.benchmark("sma");
    std::cout << "EMA:\n";
    c2.benchmark("ema");
    std::cout << "RSI:\n";
    c2.benchmark("rsi");

    std::cout << "------------------------------------------------------------\n";

    std::cout << "Benchmarking GPU Implementation E2E\n";
    std::shared_ptr<IFintel> gpuFintel3 = std::make_shared<GPUFintel>();
    Controller c4(gpuFintel3);
    std::cout << "SMA:\n";
    c4.benchmark("sma");
```

```
    std::cout << "EMA:\n";
    c4.benchmark("ema");
    std::cout << "RSI:\n";
    c4.benchmark("rsi");

    std::cout << "------------------------------------------------------------\n";

    std::cout << "Benchmarking GPU Implementation Deep\n";
    std::shared_ptr<IFintel> gpuFintel2 = std::make_shared<TimedGPUFintel>();
    Controller c3(gpuFintel2);
    std::cout << "SMA:\n";
    c3.benchmark("sma");
    std::cout << "EMA:\n";
    c3.benchmark("ema");
    std::cout << "RSI:\n";
    c3.benchmark("rsi");
    return 0;
}
```

```
std::cout << "SMA:\n";
c1.benchmark("sma");
std::cout << "EMA:\n";
c1.benchmark("ema");
std::cout << "RSI:\n";
c1.benchmark("rsi");

std::cout << "------------------------------------------------------------\n";

std::cout << "Benchmarking Parallel CPU Implementation E2E\n";
std::shared_ptr<IFintel> parallelCPUFintel = std::make_shared<ParallelCPUFintel>
Controller c5(parallelCPUFintel);
std::cout << "SMA:\n";
c5.benchmark("sma");
std::cout << "EMA:\n";
c5.benchmark("ema");
std::cout << "RSI:\n";
c5.benchmark("rsi");

std::cout << "------------------------------------------------------------\n";


std::cout << "Benchmarking GPU Implementation E2E (Slow one -- ignore)\n";
std::shared_ptr<IFintel> gpuFintel = std::make_shared<GPUFintel>();
Controller c2(gpuFintel);
std::cout << "SMA:\n";
c2.benchmark("sma");
std::cout << "EMA:\n";
c2.benchmark("ema");
std::cout << "RSI:\n";
c2.benchmark("rsi");

std::cout << "------------------------------------------------------------\n";

std::cout << "Benchmarking GPU Implementation E2E\n";
std::shared_ptr<IFintel> gpuFintel3 = std::make_shared<GPUFintel>();
Controller c4(gpuFintel3);
std::cout << "SMA:\n";
c4.benchmark("sma");
std::cout << "EMA:\n";
c4.benchmark("ema");
std::cout << "RSI:\n";
c4.benchmark("rsi");

std::cout << "------------------------------------------------------------\n";

std::cout << "Benchmarking GPU Implementation Deep\n";
std::shared_ptr<IFintel> gpuFintel2 = std::make_shared<TimedGPUFintel>();
Controller c3(gpuFintel2);
std::cout << "SMA:\n";
c3.benchmark("sma");
std::cout << "EMA:\n";
c3.benchmark("ema");
std::cout << "RSI:\n";
c3.benchmark("rsi");
return 0;
```