cpuBenchmark.cpp

```cpp
#include <iostream>
#include <intrin.h>
#include <windows.h>
#include <cmath>
#include <chrono>
void benchmarkCPU() {
    auto start = std::chrono::high_resolution_clock::now();

    volatile double sum = 0;
    for (int i = 0; i < 100000000; ++i) {
        sum += sin(i);
    }

    auto end = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> elapsed = end - start;

    std::cout << "CPU Benchmark (sin calculations): " << elapsed.count() << "
seconds" << std::endl;
}
```

cpuInfo.cpp

```cpp
#include <iostream>
#include <intrin.h>
#include <windows.h>

void printCPUInfo() {
    int cpuInfo[4] = {0};
    __cpuid(cpuInfo, 0);
    char vendor[13];
    memcpy(vendor, &cpuInfo[1], 4);
    memcpy(vendor + 4, &cpuInfo[3], 4);
    memcpy(vendor + 8, &cpuInfo[2], 4);
    vendor[12] = '\0';

    SYSTEM_INFO sysInfo;
    GetSystemInfo(&sysInfo);

    std::string architecture;
    switch (sysInfo.wProcessorArchitecture) {
        case PROCESSOR_ARCHITECTURE_AMD64:
            architecture = "x64 (AMD or Intel)";
            break;
```

```
        case PROCESSOR_ARCHITECTURE_INTEL:
            architecture = "x86 (Intel)";
            break;
        case PROCESSOR_ARCHITECTURE_ARM:
            architecture = "ARM";
            break;
        case PROCESSOR_ARCHITECTURE_ARM64:
            architecture = "ARM64";
            break;
        default:
            architecture = "Unknown Architecture";
            break;
    }

    std::cout << "CPU architecture: " << vendor << " - " << architecture <<
std::endl;
    std::cout << "CPU logical cores: " << sysInfo.dwNumberOfProcessors <<
std::endl;
    std::cout << "CPU physical cores: " << sysInfo.dwNumberOfProcessors / 2 <<
std::endl;
}
```

```
gpuBenchmark.cpp
#include <dxgi.h>
#include <d3d11.h>
#pragma comment(lib, "dxgi.lib")
#pragma comment(lib, "d3d11.lib")
#include <iostream>
#include <intrin.h>
#include <windows.h>
std::string getGPUInfo() {
    IDXGIFactory* pFactory = nullptr;
    HRESULT hr = CreateDXGIFactory(__uuidof(IDXGIFactory), (void**)&pFactory);

    if (FAILED(hr)) {
        return "Failed to create DXGIFactory.";
    }

    IDXGIAdapter* pAdapter = nullptr;
    pFactory->EnumAdapters(0, &pAdapter);

    DXGI_ADAPTER_DESC adapterDesc;
```

```
    pAdapter->GetDesc(&adapterDesc);

    std::wstring ws(adapterDesc.Description);
    std::string gpuName(ws.begin(), ws.end());

    pAdapter->Release();
    pFactory->Release();

    return gpuName;
}

double getGPUMemory() {
    IDXGIFactory* pFactory = nullptr;
    HRESULT hr = CreateDXGIFactory(__uuidof(IDXGIFactory), (void**)&pFactory);

    if (FAILED(hr)) {
        return 0;
    }

    IDXGIAdapter* pAdapter = nullptr;
    pFactory->EnumAdapters(0, &pAdapter);

    DXGI_ADAPTER_DESC adapterDesc;
    pAdapter->GetDesc(&adapterDesc);

    double gpuMemoryGB = adapterDesc.DedicatedVideoMemory / (1024.0 * 1024.0 *
1024.0);

    pAdapter->Release();
    pFactory->Release();

    return gpuMemoryGB;
}
```

```
gpuInfo.cpp
#include <iostream>
#include <intrin.h>
#include <windows.h>

void printGPUInfo() {
    DISPLAY_DEVICE displayDevice;
    displayDevice.cb = sizeof(displayDevice);
```

```
    if (EnumDisplayDevices(NULL, 0, &displayDevice, 0)) {
        std::wcout << "GPU Name: " << displayDevice.DeviceString << std::endl;
    }
}
```

memoryBenchmark.cpp

```
#include <iostream>
#include <intrin.h>
#include <windows.h>
#include <cmath>
#include <chrono>

void benchmarkMemory() {
    const int SIZE = 10000000;
    int* arr = new int[SIZE];

    auto start = std::chrono::high_resolution_clock::now();

    for (int i = 0; i < SIZE; ++i) {
        arr[i] = i;
    }

    auto end = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> elapsed = end - start;

    std::cout << "Memory Benchmark: " << elapsed.count() << " seconds" <<
std::endl;

    delete[] arr;
}
```

performanceTable.cpp

```
#include <iostream>
#include <iomanip>  // for std::setw
#include "performanceUtils.h"

PerformanceLevel classifyCPU(double clockSpeed, int cores) {
    if (clockSpeed >= 3.5 && cores >= 8) {
        return PerformanceLevel::HIGH;
    } else if (clockSpeed >= 2.5 && cores >= 4) {
        return PerformanceLevel::MEDIUM;
    } else {
```

```
        return PerformanceLevel::LOW;
    }
}

PerformanceLevel classifyMemory(double totalRAM) {
    if (totalRAM >= 16) {
        return PerformanceLevel::HIGH;
    } else if (totalRAM >= 8) {
        return PerformanceLevel::MEDIUM;
    } else {
        return PerformanceLevel::LOW;
    }
}

PerformanceLevel classifyGPU(double gpuMemory) {
    if (gpuMemory >= 8.0) {
        return PerformanceLevel::HIGH;
    } else if (gpuMemory >= 4.0) {
        return PerformanceLevel::MEDIUM;
    } else {
        return PerformanceLevel::LOW;
    }
}

PerformanceLevel classifyIO(double ioSpeedMBps) {
    if (ioSpeedMBps >= 500) {  // Example threshold for high-performance SSDs
        return PerformanceLevel::HIGH;
    } else if (ioSpeedMBps >= 100) {  // Example threshold for mid-range SSDs or
fast HDDs
        return PerformanceLevel::MEDIUM;
    } else {
        return PerformanceLevel::LOW;
    }
}



void printPerformanceTable(double clockSpeed, int cores, double totalRAM) {
    PerformanceLevel cpuPerformance = classifyCPU(clockSpeed, cores);
    PerformanceLevel memoryPerformance = classifyMemory(totalRAM);

    std::cout << std::setw(15) << "Component"
              << std::setw(20) << "Specification"
              << std::setw(20) << "Performance" << std::endl;
    std::cout << std::setw(15) << "CPU"
```

```
              << std::setw(11) << clockSpeed << " GHz, " << cores << " cores"
              << std::setw(10) << (cpuPerformance == PerformanceLevel::HIGH ?
"High" :
                                   cpuPerformance == PerformanceLevel::MEDIUM ?
"Medium" : "Low")
              << std::endl;

    std::cout << std::setw(15) << "Memory"
              << std::setw(15) << totalRAM << " GiB"
              << std::setw(15) << (memoryPerformance == PerformanceLevel::HIGH ?
"High" :
                                   memoryPerformance == PerformanceLevel::MEDIUM
? "Medium" : "Low")
              << std::endl;


}
```

performanceUtils.h

```
#ifndef PERFORMANCE_UTILS_H
#define PERFORMANCE_UTILS_H

#include <string>

enum class PerformanceLevel {
    HIGH,
    MEDIUM,
    LOW
};

std::string performanceLevelToString(PerformanceLevel level);

#endif
```

ramInfo.cpp

```
#include <iostream>
#include <intrin.h>
#include <windows.h>

void printMemoryInfo() {
    MEMORYSTATUSEX statex;
```

```
    statex.dwLength = sizeof(statex);
    GlobalMemoryStatusEx(&statex);

    std::cout << "Total System Memory (RAM): " << statex.ullTotalPhys / (1024 *
1024 * 1024) << " GB" << std::endl;
    std::cout << "Available Memory (RAM): " << statex.ullAvailPhys / (1024 * 1024
* 1024) << " GB" << std::endl;
}
```

```
Main.cpp
#include "cpuBenchmark.cpp"
#include "cpuinfo.cpp"
#include "gpuInfo.cpp"
#include "memoryBenchmark.cpp"
#include "ramInfo.cpp"
#include "performanceTable.cpp"
#include "gpuBenchmark.cpp"

int main() {
    SYSTEM_INFO sysInfo;
    GetSystemInfo(&sysInfo);

    printCPUInfo();

    MEMORYSTATUSEX statex;
    statex.dwLength = sizeof(statex);
    GlobalMemoryStatusEx(&statex);

    double clockSpeed = 3.7; // Example value; you'd normally get this from your
benchmarking
    int cores = sysInfo.dwNumberOfProcessors;
    double totalRAM = statex.ullTotalPhys / (1024.0 * 1024.0 * 1024.0);

    // Get GPU Information
    std::string gpuName = getGPUInfo();
    double gpuMemory = getGPUMemory();

    std::cout << "GPU: " << gpuName << std::endl;
    std::cout << "GPU Memory: " << gpuMemory << " GiB" << std::endl;

    // Classify Performance
    printPerformanceTable(clockSpeed, cores, totalRAM);
```

```
    PerformanceLevel gpuPerformance = classifyGPU(gpuMemory);
    std::cout << std::setw(15) << "GPU"
              << std::setw(15) << gpuMemory << " GiB"
              << std::setw(14) << (gpuPerformance == PerformanceLevel::HIGH ?
"High" :
                                   gpuPerformance == PerformanceLevel::MEDIUM ?
"Medium" : "Low")
              << std::endl;


    return 0;
}
```