Architektura i programowanie GPU

Zrównoleglanie wybranych algorytmów numerycznych

Autorzy: Krystian Sitarz, Aleksandra Makara

Spis treści

1. Cel projektu 3
2. Analiza problemu 3
   Metoda prostokątów 3
   Metoda trapezów 3
   Metoda Simpsona 4
   Metoda Monte Carlo 4
   Kwadratura Gaussa 5
3. Implementacja 6
   Katalog src 7
   Kernele 8
   Struktura katalogu tests 10
   Struktura katalogu common 11
   Środowisko 11
4. Testowanie 12
   Testy jednostkowe 12
   Testy wydajnościowe 13
5. Możliwości rozwoju 14
6. Instrukcja kompilacji i uruchomienia 15
   Wymagania 15
   Kompilacja 15
   Uruchomienie 15
7. Instrukcja obsługi 16

1. Cel projektu

Celem projektu jest implementacja wybranych algorytmów numerycznych wykorzystując technologię CUDA.

Algorytmy do implementacji:

- Całkowanie:
  - Metoda prostokątów
  - Metoda trapezów
  - Metoda Simpsona
  - Metoda Monte Carlo
  - Kwadratura Gaussa

2. Analiza problemu

Metoda prostokątów

Pierwszym krokiem jest podzielenie zakresu całkowania na “n” przedziałów.

$$\Delta x = \frac{n}{b-a}$$

gdzie “a” jest początkiem zakresu, a “b” końcem.

Następnie oblicza się wartości funkcji w punktach $x_i = a + i * \Delta x$.

Ostatecznie sumuje się wartości obliczonych funkcji: $I \approx \sum_{i=1}^{n} f(x_i) \cdot \Delta x$.

Metoda trapezów

Kroki w metodzie trapezów:

- przedział na przedziały od a do b na równe części.
- Dla każdej pary punktów obliczenie pól trapezów pomiędzy nimi
- Dodanie wszystkie pól trapezów.

Metoda Simpsona

- Dzielimy cały przedział na parzystą liczbę równych części (np. 2, 4, 6...).
- Obliczamy wartości funkcji w kolejnych punktach podziału.
- Pierwszą i ostatnią wartość dodajemy raz, środkowe wartości dodajemy na przemian: raz mnożymy przez 4, raz przez 2.
- Sumujemy wszystkie wyniki i mnożymy przez odpowiedni ułamek (na podstawie szerokości przedziałów), aby uzyskać przybliżoną wartość całki.

Metoda Monte Carlo

Metoda Monte Carlo przybliża wartość całki jako wartość średnią funkcji na przedziale (lub obszarze) na podstawie losowo dobranych punktów:

$$\int_a^b f(x)dx \approx (b-a) \cdot \frac{1}{N}\sum_{i=1}^{N} f(x_i),$$

gdzie x_i to punkty losowe z rozkładu jednostajnego na $[a, b]$

Wykorzystywana jest przede wszystkim, kiedy szybkość otrzymania wyniku jest ważniejsza od jego dokładności oraz gdy funkcje są trudne do opisania analitycznie. Działa dobrze w wysokich wymiarach.

Kwadratura Gaussa

Kwadratura Gaussa (metoda Gaussa-Legendre'a) to metoda wagowa – wybieramy optymalne punkty x_i i odpowiadające im wagi w_i tak, aby:

$$\int_a^b f(x)\ dx \approx h \cdot \sum_{i=1}^{n} w_i \cdot f(h \cdot x_i + c),$$

gdzie h i c są wyznaczane w celu zmapowania standardowego przedziału kwadratury $[-1,1]$ na $[a,b]$. Wynoszą:

$$h = \frac{b-a}{2}, \quad c = \frac{a+b}{2}$$

Punkty x_i to miejsca zerowe wielomianu ortogonalnego (np. Legendre'a), a wagi są tak dobrane, że dla n punktów całkowanie jest dokładne dla wielomianów stopnia

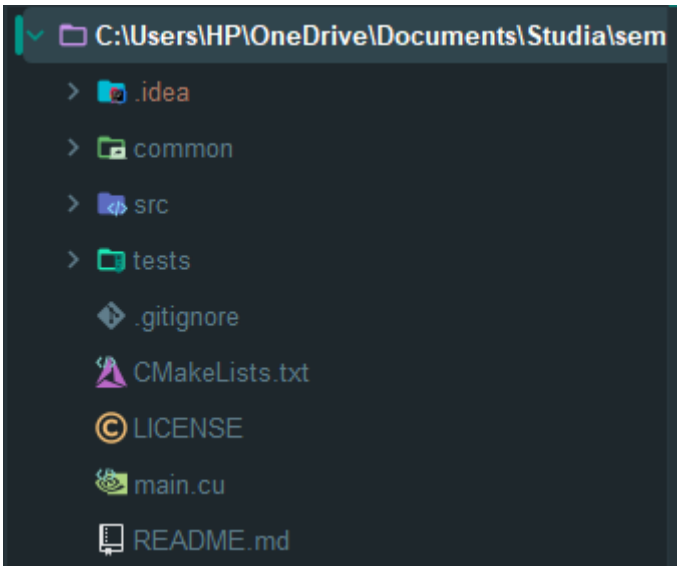
$$2n-1\,.$$

Wykorzystywana dla funkcji gładkich. Dla małej liczby punktów daje wysoką wydajność.

3. Implementacja

Struktura programu przedstawiona została na rysunku 1. Wyszczególnione katalogi zawierają:

- **.idea -** pliki konfiguracyjne środowiska JetBrains
- **common -** części programu wykorzystywane w wielu plikach
- **src -** główna logika biznesowa programu
- **tests –** testy jednostkowe i wydajnościowe



[1]Struktura katalogów programu

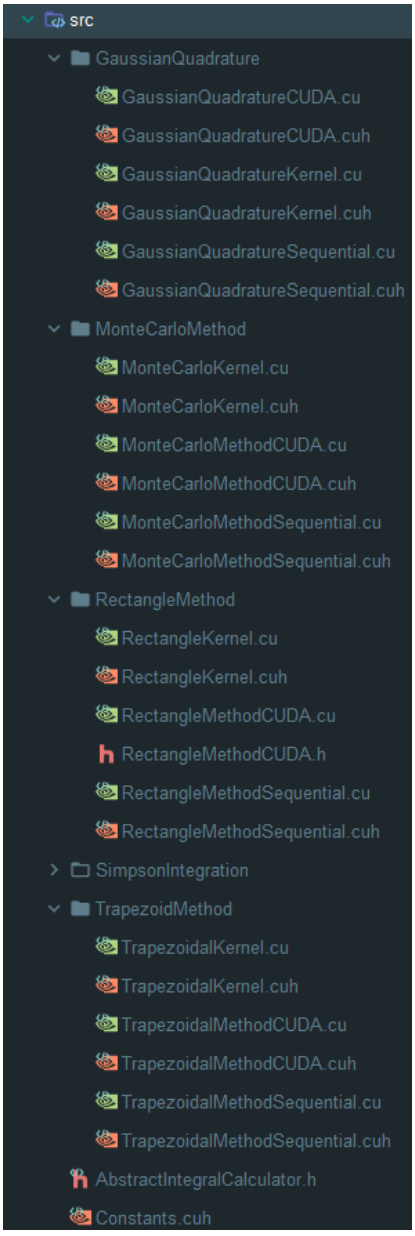
Pozostałe pliki to:

- **.gitignore -** plik konfiguracyjny narzędzia do zarządzania wersją git
- **CMakeLists.txt -** plik konfiguracyjny środowiska ułatwiającego kompilację cmake
- **LICENSE –** licencja produktu
- **main -** plik rozruchowy
- **README.md -** opis oprogramowania

[1] Struktura katalogów programu

Katalog src

Struktura katalogu została zamieszczona na rysunku 2. Zawiera główną logikę biznesową programu. Każda z metod całkowania zawiera się w parze plików z rozszerzeniami “cu’ oraz “cuh”. Dodatkowo w każdym katalogu związanym z metodą całkowania znajduje się dodatkowo kernel działający na karcie graficznej. Katalog dostał wzbogacony z plik *constants.cuh zawierający stałe*.



[2] Struktura katalogu src

[2] Struktura katalogu src

Kernele

GaussianQuadratureKernel

```
__global__ void gaussianQuadratureKernel(FunctionType functionType,
double a, double b, int n, double* results, double* nodes, double*
weights) {
    const unsigned int idx = threadIdx.x + blockIdx.x * blockDim.x;
    DoubleFunctionPtr function =
FunctionStrategy::getFunctionReference(functionType);
    if (idx < n) {
        double x_i = nodes[idx];
        double w_i = weights[idx];
        results[idx] = w_i * (*function)(0.5 * ((b - a) * x_i + (b +
a)));
    }
}
```

MonteCarloKernel

```
__global__ void monteCarloKernel(FunctionType functionType, double
a, double b, int n, double* results) {
    const unsigned int idx = threadIdx.x + blockIdx.x * blockDim.x;
    DoubleFunctionPtr function =
FunctionStrategy::getFunctionReference(functionType);
    if (idx < n) {
        curandState state;
        unsigned long seed = 1234ULL + idx;
        // unsigned long seed = clock64();
        curand_init(seed, idx, 0, &state);
        results[idx] = (*function)(a + (b - a) *
curand_uniform(&state));
    }
}
```

RectangleIntegrationKernel

```
__global__ void rectangleKernel(FunctionType functionType, double
delta, double a, int n, double* results) {
    const unsigned int idx = threadIdx.x + blockIdx.x * blockDim.x;
    DoubleFunctionPtr function =
FunctionStrategy::getFunctionReference(functionType);
    if (idx < n) {
        results[idx] = (*function)(a + static_cast<double>(idx) *
delta) * delta;
    }
}
```

SimpsonKernel

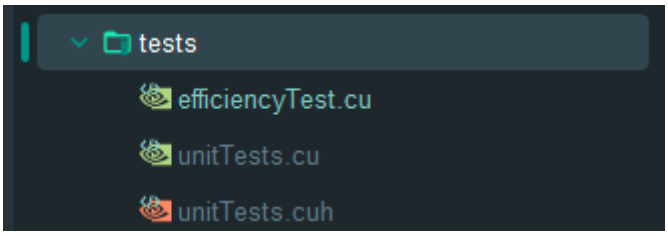
```
__global__ void simpsonKernel(FunctionType functionType, double a,
double delta, int n, double* values, double* oddBlockSums, double*
evenBlockSums) {
    __shared__ double odd_sum[BLOCK_SIZE];
    __shared__ double even_sum[BLOCK_SIZE];

    unsigned int idx = threadIdx.x + blockIdx.x * blockDim.x;
    unsigned int tid = threadIdx.x;

    double val = 0.0;
    if (idx < n + 1) {
        DoubleFunctionPtr f =
FunctionStrategy::getFunctionReference(functionType);
        double x = a + idx * delta;
        val = f(x);
        values[idx] = val;
    }
```

TrapezoidalKernel

```
__global__ void trapezoidKernel(FunctionType functionType, double
delta, double a, int n, double *results) {
    __shared__  double sharedData[BLOCK_SIZE];

    unsigned int idx = threadIdx.x + blockIdx.x * blockDim.x;
    DoubleFunctionPtr functionToCalculate =
FunctionStrategy::getFunctionReference(functionType);

    sharedData[threadIdx.x] = functionToCalculate(a +
static_cast<double>(idx) * delta);

    __syncthreads();

    if (idx < n && threadIdx.x < BLOCK_SIZE - 1) {
        results[idx] = (sharedData[threadIdx.x] +
sharedData[threadIdx.x + 1]) / 2.0 * delta;
    }
}
```

Struktura katalogu tests

Struktura katalogu została zamieszczona na rysunku 3. Katalog zawiera testy jednostkowe oraz wydajnościowe dla wszystkich zaimplementowanych metod całkowania.

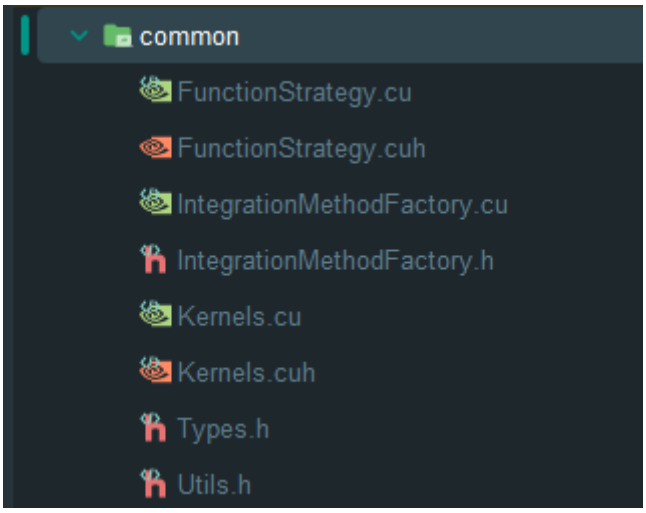


[3]Struktura katalogu tests

[3] Struktura katalogu tests

Struktura katalogu common

Struktura katalogu common znajduje się na rysunku 4. Katalog zawiera elementy projektu używane w całym projekcie.



[4]Struktura katalogu common

Środowisko

Jako system budowania zdecydowano się na Cmake. CMake to system budowania (build system generator) — narzędzie, które:

- generuje pliki projektów dla kompilatorów (np. `Makefile`, `Visual Studio`, `Ninja`, itp.),
- obsługuje wieloplatformowość (Linux, Windows, macOS),
- pozwala konfigurować kompilację dużych projektów w sposób elastyczny i modułowy.

Co robi Cmake ze środowiskiem wykonawczym CUDA:

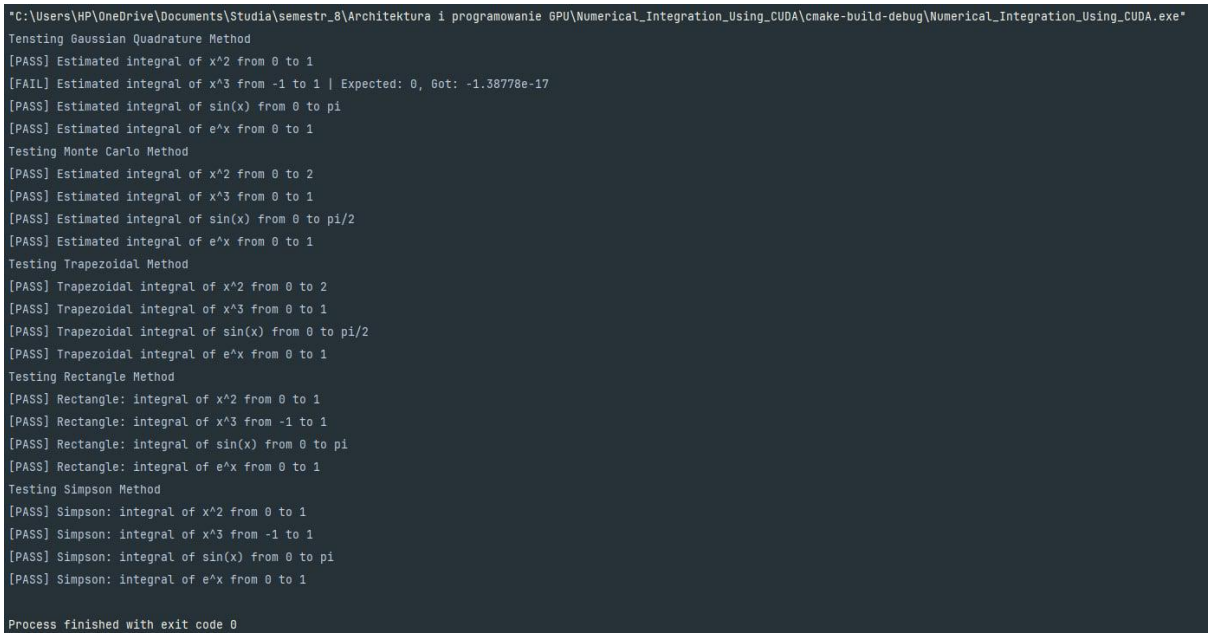
- CMake wykrywa CUDA i NVCC.
- Tworzy build:
  - `nvcc -c main.cu -o main.o`
  - `g++ main.o -lcudart -o program`
- Finalnie otrzymuje się działający program, który można uruchomić na GPU.

[4] Struktura katalogu common

4. Testowanie

Testy jednostkowe

Stworzone zostały testy jednostkowe sprawdzające poprawność wykonywanych metod z przybliżeniem do wybranego zaokrąglenia. Jak można zauważyć na załączonym obrazku, dla wybranych w teście parametrów, kwadratura Gaussa przybliża niewystarczająco.



[5]Wyniki testów jednostkowych

Metoda wykorzystywana do testów jednostkowych:

static void assertEqual(double actual, double expected, double delta, const std::string& testName) {...}

double actual – wyznaczona przez metodę wartość całki

double expected – faktyczna wartość całki

double delta – dopuszczany błąd dla metody

const std::string& testName – nazwa testu

[5]Wyniki testów jednostkowych

Testy wydajnościowe

W celu porównania wydajności algorytmów sekwencyjnych i równoległych (wykorzystujących architekturę CUDA) przeprowadzono testy wydajnościowe. Uzyskane wyniki wskazują, że w przeprowadzonych przez nas eksperymentach podejście sekwencyjne okazało się bardziej efektywne. Przyczyną takiego wyniku jest wartość parametru n, który – zależnie od zastosowanej metody całkowania – może przyjmować różne rozmiary. W sytuacji, gdy n nie jest wystarczająco duże, narzut związany z uruchomieniem i obsługą obliczeń na GPU przewyższa potencjalne zyski wydajnościowe, przez co użycie architektury CUDA staje się nieopłacalne.

Warto jednak zaznaczyć, że w przypadku niektórych metod numerycznych – takich jak metoda trapezów lub prostokątów czy metody wymagające bardzo dużej liczby podziałów przedziału – wartość n może być na tyle duża, że obliczenia wykonywane z użyciem CUDA są zauważalnie szybsze niż ich odpowiedniki sekwencyjne. W takich przypadkach architektura równoległa pozwala w pełni wykorzystać potencjał obliczeniowy GPU.

```
Gaussian quadrature
[PARALLEL] Time: 14.0524 ms
[SEQUENTIAL] Time: 9 microSeconds
Monte Carlo method
[PARALLEL] Time: 6234.07 ms
[SEQUENTIAL] Time: 3914 ms
Trapezoidal method
[PARALLEL] Time 17664.2 ms
[SEQUENTIAL] Time: 30107.1 ms
Rectangle method
[PARALLEL] Time 4507.3 ms
[SEQUENTIAL] Time: 14680.7 ms
Simpson method
[PARALLEL] Time 11726.4 ms
[SEQUENTIAL] Time: 12096 ms
```

[6] Wyniki testów wydajnościowych

Testy przeprowadzane są poprzez wykonanie tych samych obliczeń dla tej samej metody przy użyciu różnych algorytmów (sekwencyjnego i równoległego). Każda z metod przyjmuje jako parametr dodatkową flagę, która umożliwia pomiar czasu wykonywanych algorytmów.

[6] Wyniki testów wydajnościowych

5. Możliwości rozwoju

Aplikacja została zaprojektowana z myślą o jej przyszłym rozwoju. W kolejnych etapach możliwa jest realizacja następujących elementów:

- Implementacja nowych metod całkowania.
- Dodanie kolejnych, dostępnych do całkowania funkcji.
- Dodanie obsługi równań wejściowych w formie tekstowej dla bardziej złożonych funkcji, np. $\sin(x) \cdot x^2$
- Wizualizacja wyników i ich zapis do pliku, np.:
  - wykres funkcji w zadanym przedziale,
  - obszar pod krzywą (czyli całkowany obszar),
  - porównanie działania różnych metod (np. Monte Carlo vs trapezy).
- Dodanie opcji analitycznego porównania, np.:
  - porównywanie wyniku numerycznego z dokładnym,
  - wyznaczanie błędów.
- Interaktywne GUI.
- Przekształcenie aplikacji terminalowej w samodzielną aplikację desktopową.

6. Instrukcja kompilacji i uruchomienia

Wymagania

Aby skompilować ten projekt CUDA, potrzeba:

1. **CUDA Toolkit** w wersji odpowiedniej do sterownika karty graficznej,
2. **CMake** w wersji >= 3.18

Kompilacja

Aby skompilować projekt należy wykonać następujące kroki:

- Przejść w terminalu do katalogu projektu
- Wykonać polecenia:
  - mkdir build
  - cd build
  - cmake ..
  - cmake –build .

Upewnij się, że:

- Masz ustawione w systemie PATH do `nvcc` (np. `/usr/local/cuda/bin`).
- Masz sterowniki NVIDIA i kompatybilną wersję CUDA Toolkit.
- Używasz CMake **≥ 3.18** (wtedy obsługuje CUDA jako język).

Uruchomienie

Aby uruchomić program:

- Przejdź do folderu build
- Uruchom: Numerical_Integration_Using_CUDA

7. Instrukcja obsługi

Po uruchomieniu aplikacji w terminalu wypisują się po kolei polecenia dla użytkownika w celu wprowadzenia parametrów, dla których ma zostać wykonany program. Program przyjmuje po kolei:

- Nazwę metody, która zostanie wykorzystana do całkowania

  Po wyświetleniu się polecenia: “Set integration method (Rectangle, Trapezoidal, MonteCarlo, GaussianQuadrature, Simpson):”, użytkownik musi wpisać jedną z nazw metod wymienionych w nawiasie.

- Nazwę funkcji, która ma zostać poddana całkowaniu

  Po wyświetleniu się polecenia: “Set function (square, cubic, sinus, cosinus, exponential, hyperbolic, logarithm, squareRoot):”, użytkownik musi wpisać jedną z nazw funkcji wymienionych w nawiasie.

- Przedział, na którym funkcja będzie całkowana

  Polecenie jest podzielone na dwa osobne, które przyjmują odpowiednio początek i koniec przedziału.
  Po wyświetleniu się polecenia: “Set lower bound (a):”, użytkownik musi wpisać początek przedziału.
  Po wyświetleniu się polecenia: “Set upper bound (b):”, użytkownik musi wpisać koniec przedziału.

- Liczbę n, która w zależności od metody może oznaczać:

  - Liczbę przedziałów (metoda prostokątów i metoda trapezów)
  - Liczbę losowanych punktów (metoda Monte Carlo)
  - Liczbę węzłów (kwadratura Gaussa)
  - Liczbę danych wejściowych (metoda Simpsona)

  Po wyświetleniu się polecenia: “Set number of intervals/random points/nodes/inputs (n):”, użytkownik musi wpisać wartość liczby n.

Gdy użytkownik wprowadzi wszystkie parametry poprawnie, program wyznaczy całkę dla podanych argumentów i zwróci wynik w postaci “Result: <integration result>”. W przeciwnym wypadku wypisany zostanie błąd jaki popełnił użytkownik lub program podczas działania aplikacji dla wybranych parametrów.