Міністерство освіти і науки України

Харківський національний університет радіоелектроніки

МЕТОДИЧНІ ВКАЗІВКИ

до лабораторних робіт з дисципліни

«РОЗПІЗНАВАННЯ ОБРАЗІВ НА ОСНОВІ ТЕХНОЛОГІЙ ПРОГРАМУВАННЯ ГРАФІЧНИХ ПРОЦЕСОРІВ»

для студентів денної форми навчання
спеціальності 123 Комп’ютерна інженерія
за освітньо-професійною програмою
«Комп’ютерні інтелектуальні технології»

Електронне видання

ЗАТВЕРДЖЕНО
кафедрою КІТС
Протокол №1 від30.08.2021

Харків 2021

Методичні вказівки до лабораторних робіт з дисципліни «РОЗПІЗНАВАННЯ ОБРАЗІВ НА ОСНОВІ ТЕХНОЛОГІЙ ПРОГРАМУВАННЯ ГРАФІЧНИХ ПРОЦЕСОРІВ» для студентів денної форми навчання спеціальності 123 Комп’ютерна інженерія за освітньо-професійною програмою «Комп’ютерні інтелектуальні технології» [Електронне видання] / Упорядн . Н.М.Сердюк – Харків: ХНУРЕ, 2021. – 26с.

Упорядник Н.М.Сердюк

ЗМІСТ

ВСТУП

Графічний процесор відеокарти (Graphics processing unit, GPU) дуже ефективно обробляє і відображає комп'ютерну графіку завдяки спеціалізованої конвеєрної архітектурі, тим самим набагато ефективніше в обробці графічної інформації, ніж типовий центральний процесор (central processing unit, CPU).

Графічні процесори використовувалися для не графічних обчислень протягом декількох років в таких додатках: симуляція фізики, обробка сигналів, обчислювальна математика і геометрія, операції з базами даних, обчислювальна біологія та економіка, комп'ютерний зір.

Протягом декількох років, GPU розвинувся в абсолютну обчислювальну машину, сьогоднішні GPU пропонують неймовірні ресурси для графічної і не графічної обробки.

1. Програмування графічних процесорів за допомогою технології NVIDIA CUDA

1.1 Мета роботи: Ознайомитися з технологією NVIDIA CUDA. Навчитися компілювати і запускати програми, що містять CUDA- код.

1.2 Теоретична частина

CUDA (англ. Compute Unified Device Architecture) - технологія GPGPU (англ. General-Purpose computing on Graphics Processing Units - обчислення загального призначення на графічних процесорах), що дозволяє програмістам реалізовувати на спрощеному мові програмування Сі алгоритми, здійснимі на графічних процесорах відеокарт GeForce восьмого покоління і старше. Технологія CUDA розроблена компанією Nvidia. Фактично CUDA дозволяє включати в текст Сі програми спеціальні функції. Ці функції пишуться на спрощеному мові програмування Сі та виконуються на графічних процесорах Nvidia.

У середньому, при перенесенні обчислень на GPU, у багатьох задачах досягається прискорення в 5-30 разів, у порівнянні з швидкими універсальними процесорами.

CUDA включає два API: високого рівня (CUDA Runtime API) і низького (CUDA Driver API), хоча в одній програмі одночасне використання обох неможливо, потрібно використовувати або один або інший. Високорівнева працює «зверху» низкоуровневого, всі виклики runtime транслюються в прості інструкції, оброблювані низькорівневим Driver API. «Високорівнева» API передбачає знання основ про пристрій і роботу відеокарт NVIDIA.

Є і ще один рівень, вищий - дві бібліотеки:

CUBLAS - CUDA варіант BLAS (Basic Linear Algebra Subprograms), призначений для обчислень задач лінійної алгебри і використовує прямий доступ до ресурсів GPU;

CUFFT - CUDA варіант бібліотеки Fast Fourier Transform для розрахунку швидкого перетворення Фур'є, широко використовуваного при обробці

сигналів. Підтримуються наступні типи перетворень: complex-complex (C2C), real-complex (R2C) і complex-real (C2R).

1.3 Порядок виконання роботи

1.3.1 Написання програм на CUDA

Для розробки власних програм слід розбиратися в базових архітектурні особливості відеочіпів NVIDIA. Всі інструкції в відеокарті виконуються за принципом SIMD, коли одна інструкція застосовується до всіх потоків в warp (в CUDA це група з 32 потоків - мінімальний обсяг даних, які обробляються Мультіпроцесори). Цей спосіб виконання назвали SIMT (single instruction multiple threads - одна інструкція і багато потоків).

При виконанні програми, центральний процесор виконує свої порції коду, а GPU виконує CUDA код з найбільш важкими паралельними обчисленнями. Ця частина, призначена для GPU, називається ядром (kernel). В ядрі визначаються операції, які будуть виконані над даними.

Таким чином, CUDA використовує паралельну модель обчислень, коли кожен з SIMD процесорів виконує ту ж інструкцію над різними елементами даних паралельно. GPU є обчислювальним пристроєм (device) для центрального процесора (host), що володіє власною пам'яттю і обробляють паралельно велику кількість потоків. Ядром (kernel) називається функція для GPU, що виконується потоками. Відеокарта відрізняється від CPU тим, що може обробляти одночасно сотні тисяч потоків, що зазвичай для графіки, яка добре розпаралелюється.

Модель програмування в CUDA передбачає групування потоків. Потоки об'єднуються в блоки потоків (thread block) - одномірні або двовимірні сітки потоків, що взаємодіють між собою за допомогою розділяється пам'яті і точок синхронізації. Програма (ядро, kernel) виповнюється над сіткою (grid) блоків потоків (thread blocks), див. рис.3.1. Одночасно виконується одна сітка. Кожен блок може бути одно-, дво- або тривимірним за формою, і може складатися з 512 потоків на поточному апаратному забезпеченні.

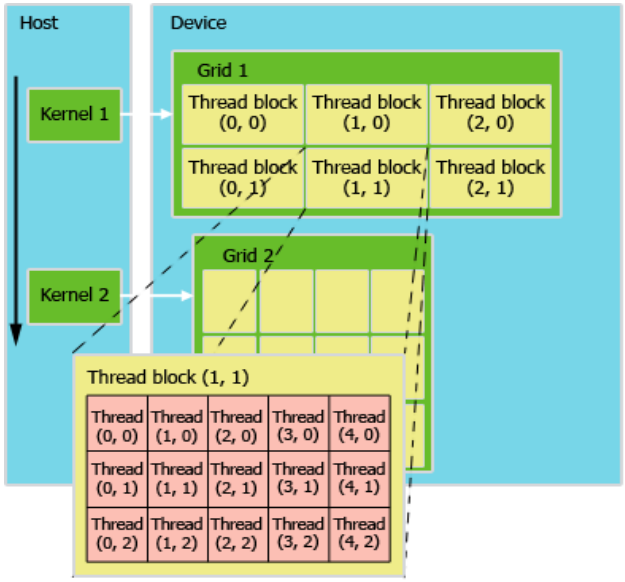

Рисунок 1.1 - Модель програмування в CUDA

1.3.2 Приклад програми на CUDA

Як приклад розглянемо задачу множення квадратної матриці на вектор.
Початкові дані:
A [n] [n] - матриця розмірності n x n;
b [n] - вектор, що складається з n елементів.
результат:
c [n] - вектор з n елементів.

```
//Послідовний алгоритм множення матриці на вектор
for (i=0; i<n; i++)
{
          c[i]=0;
          for (j=0; j<m; j++)
          {
               c[i]=A[i][j]*b[j];
          }
}
```

Тепер розглянемо рішення цього завдання на відеокарті. Наступний код ілюструє приклад виклику функції CUDA:

```
// ініціалізація CUDA
if(!InitCUDA())   { return 0; }

int Size = 1000;
// звичайні масиви в оперативній пам'яті
```

```
float *h_a,*h_b,*h_c;
h_a = new float[Size*Size];
h_b = new float[Size];
h_c = new float[Size];

for (int i=0;i<Size;i++)  // ініціалізація масивів a и b
{
  for (int k=0;k<Size;k++)
     {
       h_a[i*Size+k]=1;
     }
  h_b[i]=2;
}

// покажчики на масиви в відеопам'яті
float *d_a,*d_b,*d_c;

// виділення відеопам'яті
cudaMalloc((void **)&d_a, sizeof(float)*Size*Size);
cudaMalloc((void **)&d_b, sizeof(float)*Size);
cudaMalloc((void **)&d_c, sizeof(float)*Size);

// копіювання з оперативної пам'яті у відеопам'ять
CUDA_SAFE_CALL(cudaMemcpy(d_a, h_a, sizeof(float)*Size*Size,
                                cudaMemcpyHostToDevice) );
CUDA_SAFE_CALL(cudaMemcpy(d_b, h_b, sizeof(float)*Size,
                                cudaMemcpyHostToDevice) );

// установка кількості блоків
dim3 grid((Size+255)/256, 1, 1);
// установка кількості потоків у блокі
dim3 threads(256, 1, 1);

// визов функції
MatrVectMul<<< grid, threads >>> (d_c, d_a, d_b,Size);

// копіювання з відеопам'яті в оперативну пам'ять
CUDA_SAFE_CALL(cudaMemcpy(h_c, d_c, sizeof(float)*Size,
                                cudaMemcpyDeviceToHost) );

// звільнення пам'яті
CUDA_SAFE_CALL(cudaFree(d_a));
CUDA_SAFE_CALL(cudaFree(d_b));
CUDA_SAFE_CALL(cudaFree(d_c));
```

Структуру будь-якої програми з використанням CUDA можна уявити аналогічно розглянутому вище прикладу. Таким чином, можна запропонувати наступну послідовність дій:

1) ініціалізація CUDA;

2) виділення відеопам'яті для зберігання даних програми;

3) копіювання необхідних для роботи функції даних з оперативної пам'яті в відеопам'ять;

4) виклик функції CUDA;

5) копіювання повертаються даних з відеопам'яті в оперативну;

6) звільнення відеопам'яті.

Приклад функції, здійсненною на відео карті

```
extern "C" __global__ void MatrVectMul(float *d_c, float *d_a, float *d_b, int Size)
{

          int i = blockIdx.x*blockDim.x+threadIdx.x;
          int k;
          d_c[i]=0;
          for (k=0;k<Size;k++)
            {
               d_c[i]+=d_a[i*Size+k]*d_b[k];
            }
}
```

де: `threadIdx.x` – ідентифікатор потоку в блоці з координування x, `blockIdx.x` – ідентифікатор блоку в гріді по координаті x, `blockDim.x` – кількість потоків в одному блоці.

Таким чином виходить унікальний ідентифікатор потоку (в даному випадку `i`), який можна використовувати в програмі, працюючи з усіма потоками як з одновимірним масивом.

Важливо пам'ятати, що функція, призначена для виконання на відеокарті, не повинна звертати до оперативної пам'яті. Подібне звернення призведе до помилки. Якщо необхідно працювати з будь-яким об'єктом в оперативній

пам'яті, попередньо його треба скопіювати в відеопам'ять, і звертатися з функції CUDA до цієї копії.

Серед основних особливостей CUDA слід зазначити відсутність підтримки подвійної точності (типу *double*). Також для функцій CUDA встановлено максимальний час виконання, відсутня рекурсія, не можна оголосити функцію зі змінним числом аргументів.

Слід зазначити наступне, що функція, яка працює на відеокарті, повинна виконуватися не більше 1 секунди. Інакше, функція буде не завершена, і програма завершиться з помилкою.

Для синхронізації потоків в блоці існує функція *__syncthreads ()*, яка чекає, поки всі запущені потоки відпрацюють до цієї точки. Функція *__syncthreads ()* необхідна, коли дані, що обробляються одним потоком, потім використовуються іншими потоками.

1.3.3 Вимірювання часу в CUDA

Для вимірювання часу виконання окремих частин програми і аналізу швидкості виконання тієї чи іншої ділянки коду зручно використовувати вбудовані в CUDA функції по виміру часу.

```
unsigned int timer;

// створення таймера
cutCreateTimer(&timer);

// запуск таймера
cutStartTimer(timer);

… // код, час виконання якого необхідно заміряти

// зупинка таймера
cutStopTimer( timer);

// отримання часу
printf("time: %f (ms)\n", cutGetTimerValue(timer));

// видалення таймера
cutDeleteTimer( timer);
```

1.4 Порядок виконання роботи

1.4.1 Створення проекту CUDA в VS

Після установки CUDA SDK, CUDA Toolkit і VS-інтегратора в меню створення проекту з'явиться новий пункт. Щоб створити проект CUDA треба вибрати закладку Visual C ++, CUDA, CUDAWinApp.

Вибрати тип проекту (за замовчуванням створюється консольне) та інші параметри. В результаті буде створений проект, який виводить на екран рядок «Hello CUDA!».

Щоб додаток могло запуститися, в поточному каталозі повинен бути присутнім файл cutil32D.dll для роботи в режимі Debug і файл cutil32.dll для роботи в режимі Release (наприклад, для проекту з назвою lab4 необхідно покласти ці файли в директорію lab4 / lab4). Причому, шлях до файлу проекту не повинен містити російських букв.

1.4.2 Завдання

Написати програму відповідно до варіанту, при цьому реалізувати 2 функції: одну для виконання на процесорі, другу для виконання на відеокарті. Потім порівняти результати (повернені значення) і швидкість роботи.

Завдання №1.

Варіанти завдань:

Розробіть програму, для обчислення значення функції f (x) на відрізку [1, N + 1] з кроком h = N / k, де N - номер варіанта і складіть таблицю:

k	час розрахунку на CPU	час розрахунку на GPU
N		
$N*10^2$		
$N*10^4$		
$N*10^6$		
...		

a = N, b = N*2, k=N/2, z = N^2

1. $y = \sqrt{x-1} + \frac{1}{x-3};$
2. $y = \frac{1}{k * \sqrt{2\pi}} * e^{\frac{-(x-a)^2}{2k^2}};$
3. $y = \frac{1}{2b} * e^{\frac{-|x-a|}{b}};$
4. $y = \frac{\sin x}{2\cos^2 x} - \cos x - \frac{3}{2} tgx;$
5. $y = \cos x(\ln\left|2 - e^{-|a+x|}\right|);$
6. $y = \frac{e^{\sin^2 x} + \ln|tgx|}{\sin x};$
7. $y = a * e^{-ax} * \sin x;$
8. $y = \frac{\sin^3\left|ax^3 + bx^2 - ab\right|}{\sqrt{\left|ax^3 + bx^2 - ab\right|}};$
9. $y = \frac{\sqrt{|\cos x|}}{\sqrt{1+x^2}};$
10. $y = \frac{1 + \sin\sqrt{x+1}}{\cos(12z-4)};$
11. $y = x - \frac{x^3}{3} + \frac{x^5}{5};$
12. $y = \frac{x^2 - 7x + 10}{x^2 - 8x + 12};$
13. $y = \frac{\cos x}{a - 2x} + 16x \cdot \cos(xz) - 2;$
14. $y = \left|x^2 - x^3\right| - \frac{7x}{x^3 - 15x};$
15. $y = e^{-x} - \cos x + \sin(2xz);$

Завдання №2.

Варіанти завдань:

1. Розробіть програму для знаходження мінімального значення серед елементів вектора.

2. Розробіть програму для знаходження максимального значення серед елементів вектора.

3. Розробіть програму для знаходження суми всіх елементів вектора.

4. Розробіть програму для знаходження добутку всіх елементів вектора.

5. Розробіть програму для обчислення скалярного добутку двох векторів.

6. Розробіть програму рішення задачі пошуку максимального значення серед мінімальних елементів рядків матриці.

7. Розробіть програму рішення задачі пошуку мінімального значення серед максимальних елементів рядків матриці.

8. Розробіть програму для вирішення завдання транспонування матриці.

9. Розробіть програму для знаходжень добутку вектора на матрицю.

10. Розробіть програму для знаходження мінімального значення серед елементів матриці.

11. Виконати додавання двох матриць однакового розміру.

12. Знайти суму максимальних елементів рядків матриці.

13. Знайти площу опуклого багатокутника, заданого координатами вершин.

14. Дана матриця дійсних чисел. Перетворити матрицю таким чином, щоб елементи її рядків йшли по спадаючій.

15. Дана матриця дійсних чисел. Перетворити матрицю таким чином, щоб елементи її стовпців йшли по спадаючій.

1.5 Зміст звіту

1. Титульний аркуш, назва роботи.
2. Мета роботи.
3. Завдання до виконання лабораторної роботи відповідно до варіанту.
4. Опис алгоритму розв'язання задачі у вигляді блок-схем або словесне. Опис використовуваних паралельних методів обчислень.
5. Програма у вигляді вихідних кодів (з пояснювальними коментарями), а також в відкомпілювався вигляді для демонстрації на ЕОМ.
6. Приклади роботи програми на тестових даних.
7. Висновки по роботі.

1.6 Контрольні питання

1. Що мається на увазі під термінами *kernel*, *host* та *device*?

2. Поясніть поняття *grid, thread block, thread*.

3. На яких відеокартах доступна технологія CUDA?

4. Як ви вважаєте, чи будуть перераховані вище завдання виконуватися швидше для `Size = 5` на відеокарті, ніж на процесорі і чому?

5. Що спільного в технологіях OpenMP і Nvidia CUDA і чим вони відрізняються (з точки зору програмування)?

2. Встановлення CUDA Toolkit, основні принципи написання CUDA-коду

2.1 Мета роботи: Написання програми з використанням технології CUDA. Навчитися компілювати і запускати програми, що містять CUDA- код.

2.2 Теоретична частина

На сьогодні ідея застосування графічних процесорів та технології CUDA є перспективною та популярною. CUDA широко застосовується для вирішення задач обробки зображень, машинного навчання та інженерних розрахунків, часто дозволяючи порівняно недорого та без громіздкого обладнання забезпечити задовільну продуктивність.

Звичайно, існують й інші технології, які також дозволяють виконувати обчислення на графічних процесорах, зокрема, **OpenCL**. Проте, як показують експерименти та дослідження, функція-ядро OpenCL виконується повільніше на ***13-63%***, а під час *end-to-end* тестування на ***16-67%*** повільніше.

Можливості CUDA

Розглянемо основні можливості технології.

- стандартна мова програмування C для паралельної розробки застосунків з використанням GPU;
- стандартні бібліотеки для швидкого перетворення Фур'є та базові пакети програм лінійної алгебри;
- спеціальний драйвер CUDA зі швидкою передачею даних між GPU та CPU;
- драйвер CUDA взаємодіє з OpenGL і DirectX;
- підтримка операційних систем Linux, Windows і Mac.

Набір інструментів CUDA

Інструмент CUDA Toolkit – середовище розробки для GPU з підтримкою CUDA, засноване на мові C. Це середовище включає:

- C-компілятор nvcc;
- бібліотеки FFT і BLAS для GPU;
- налагоджувач для GPU;
- драйвер CUDA Runtime;

2.3 Порядок виконання роботи

2.3.1 Встановлення CUDA

Для того, щоб встановити програмне забезпечення CUDA переходимо на сайт Nvidia, обираємо платформу та операційну систему. Після цього розпочнеться завантаження останньої версії CUDA Toolkit- **9.0**.

Версію ПЗ потрібно обирати в залежності від того, яка у вас відеокарта. Тобто яку версію CUDA підтримує ваш графічний процесор і чи підтримує взагалі. Перейшовши за цим посиланням ви можете побачити усі відеокарти, що підтримують технологію CUDA. Під час встановлення ПЗ є момент, коли потрібно обрати або перезапис графічного драйвера новим, або ж залишити драйвер, що вже встановлений в системі. Рекомендовано обрати перезапис, якщо досить «старий» драйвер.

Для перевірки правильності встановлення CUDA, можете відкрити приклад проекту Visual Studio `deviceQuery`, що знаходиться за шляхом `C:\ProgramData\NVIDIA Corporation\CUDA Samples\версія_CUDA\1_Utilities\deviceQuery`. Скомпілювавши та запустивши проект, ви побачите схоже вікно:

```
deviceQuery.exe Starting...

 CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: "GeForce GTX 950"
  CUDA Driver Version / Runtime Version          8.0 / 8.0
  CUDA Capability Major/Minor version number:    5.2
  Total amount of global memory:                 2048 MBytes (2147483648 bytes)
  ( 6) Multiprocessors, (128) CUDA Cores/MP:     768 CUDA Cores
  GPU Max Clock rate:                            1329 MHz (1.33 GHz)
  Memory Clock rate:                             3305 Mhz
  Memory Bus Width:                              128-bit
  L2 Cache Size:                                 1048576 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D=(65536, 65536), 3D=(4096, 4096, 4096)
  Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 2 copy engine(s)
  Run time limit on kernels:                     Yes
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  CUDA Device Driver Mode (TCC or WDDM):         WDDM (Windows Display Driver Model)
  Device supports Unified Addressing (UVA):      Yes
  Device PCI Domain ID / Bus ID / location ID:   0 / 2 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 8.0, CUDA Runtime Version = 8.0, NumDevs = 1, Device0 = GeForce GTX
950
Result = PASS
```

Рисунок 2.1 - Результат виконання deviceQuery

У вікні виводиться вся основна інформація про відеокарту, її обчислювальні можливості та інші цікаві показники. Усі інші приклади

програм розміщені за шляхом `C:\ProgramData\NVIDIA Corporation\CUDA Samples\версія_CUDA`.

2.3.2 Програмна модель

Програму на CUDA можна логічно розділити на дві частини, перша частина (керуюча) виконується на CPU, друга частина (обчислювальна) виконується на GPU. CPU в термінології CUDA називається **host**, GPU – **device**. Частина коду, яка повинна виконуватися на GPU, називається ядром (**kernel**), вона описується у вигляді функції, пізніше ми розглянемо приклади програм.

Розширення мови С, що входять у CUDA складаються з:

- специфікаторів функцій, які описують, де буде виконуватися функція і звідки вона може бути викликана;
- специфікаторів змінних, що задають тип пам'яті, який використовується для даної змінної;
- директив, що використовуються для запуску ядра і задає як дані, так і ієрархію потоків;
- вбудованих змінних, що містять інформацію про поточний потік;
- runtime, що містить в собі додаткові типи даних.

Програми зберігаються у файлі з розширенням `.cu`.

Всі програмні коди компілюються використовуючи CUDA API. Спочатку компілюється код, що призначений виключно для центрального процесора, а інший код, призначений для графічного процесора, компілюється в проміжну мову PTX (щось схоже до байт-коду в Java) для виявлення можливих помилок. Після чого, компілюється в «зрозумілу» для CPU/GPU мову.

Специфікатори (модифікатори)

Перед функціями в `.cu` файлі можуть стояти наступні модифікатори:

- `__device__` – означає, що функція виконується тільки на відеокарті. ***З програми, що виконується на звичайному процесорі (хості), її викликати не можна***;
- `__global__` – функція – початок вашого обчислювального ядра. Виконується на відеокарті, але запускається з хоста;
- `__host__` – виконується і запускається тільки з хоста (тобто звичайна функція С).

При цьому модифікатори `__host__` і `__device__` можуть бути використані разом (це означає, що відповідна функція може виконуватися як на GPU, так і на CPU – відповідний код для обох платформ буде автоматично згенерований компілятором). Модифікатори `__global__` і `__host__` не можуть бути використані разом.

На функції, що виконуються на GPU, накладено певні обмеження: вони не можуть містити рекурсії, не можуть мати змінне число вхідних аргументів, не можуть містити статичні змінні, а також не можна взяти адресу такої функції.

Додані змінні

В мову додані наступні спеціальні змінні:

- **gridDim** – розмір grid (тип *dim3*);
- **blockDim** – розмір блоку (тип *dim3*);
- **blockIdx** – індекс поточного блоку в grid (тип *uint3*);
- **threadIdx** – індекс поточного потоку в блоці (тип *uint3*);
- **warpSize** – розмір warp (тип *int*).

Також додаються **1 / 2 / 3 / 4** – мірні вектори з базових типів: char1, char2, char3, char4, uchar1, uchar2, uchar3, uchar4, short1, short2, short3, short4, ushort1, ushort2, ushort3, ushort4, int1, int2, int3, int4, uint1, uint2, uint3, uint4 і так далі.

Директива виклику ядра

Для запуску ядра на GPU використовується наступна конструкція:

```
kernelName <<<Dg,Db,Ns,S>>> (args)
```

Тут `kernelName` – ім'я (адреса) відповідної `__global__` функції, `Dg` – змінна (або значення) типу *dim3*, що задає розмірність **grid** (в блоках), `Db` – змінна (або значення) типу *dim3*, що задає розмірність **блоку** (в потоках), `Ns` – змінна (або значення) типу *size_t*, що задає додатковий обсяг спільної пам'яті, яка повинна бути динамічно виділена (до вже статично виділеної shared-пам'яті; параметр не є обов'язковим), `S` – змінна (або значення) типу *cudaStream_t*, що задає потік (потік CUDA), в якому має викликатися ядро, за замовчуванням використовується потік 0. Через `args` позначено аргументи виклику функції `kernelName`.

Множину потоків у *блоці* та блоків у *grid* можна задавати у вигляді 1 / 2 / 3 -мірних векторів. Розміри сітки та максимально можлива кількість потоків напряму залежать від відеокарти.

Також в мову C додана функція `__syncthreads` (детальніше тут), яка здійснює синхронізацію всіх потоків блоку. ***Управління з неї буде повернуто тільки тоді, коли всі потоки даного блоку викличуть цю функцію***. Тобто, коли весь код, що йде перед цим викликом, вже виконано. Ця функція дуже зручна для організації безконфліктної роботи зі спільною пам'яттю.

Як потік знає над якими даними йому працювати?

Припустимо, що потрібно зробити деякі операції над зображенням (зберігається у змінній `fox`) розміром ***400x400*** пікселів. Зображення можна

розділити на ділянки (блоки) ***10x10*** пікселів і для опрацювання кожної ділянки запустити окремий потік.

Оскільки параметри, що передаються у ядро однакові для всіх потоків, то кожен потік повинен сам «отримати дані для себе». Щоб це зробити, потоку «потрібно розрахувати», в якому місці зображення він знаходиться.

```
const int ix = blockDim.x * blockIdx.x + threadIdx.x;

const int iy = blockDim.y * blockIdx.y + threadIdx.y;
```

`ix` та `iy` – координати, за допомогою яких можна отримати вихідні дані з масиву зображення. Як бачите, для цього якраз і застосовуються описані вище змінні.

2.3.4 Програма з використанням CUDA

Для прикладу, потрібно обчислити суму двох векторів розмірністю **N**. Для цього опишемо наступне ядро:

```
__global__ void addVector(float* vector1, float* vector2, float*
result)
{
  //Отримуємо id потоку.
  int idx = threadIdx.x;

  //Обчислюємо результат.
  result[idx] = vector1[idx] + vector2[idx];
}
```

Таким чином, розпаралелювання буде виконано автоматично під час запуску ядра. У цій функції також використовується вбудована змінна `threadIdx` (дивися вище) та її поле `x`, що дозволяє отримати координату `x` потоку в блоці. Після чого, проводимо розрахунок кожного елемента вектора в окремому потоці.

```
#define SIZE 512
__host__ int main()
{
  //Виділяємо пам'ять для векторів
  float* vec1 = new float[SIZE];
  float* vec2 = new float[SIZE];
  float* vec3 = new float[SIZE]; //результат

  //Ініціалізуємо значення векторів
  for (int i = 0; i < SIZE; i++)
  {
    vec1[i] = i;
    vec2[i] = i;
  }

  //Вказівники на створені вектори
```

```
        float* devVec1;
        float* devVec2;
        float* devVec3;

        //Виділяємо пам'ять для вектрів на відеокарті
        cudaMalloc((void**)&devVec1, sizeof(float) * SIZE);
        cudaMalloc((void**)&devVec2, sizeof(float) * SIZE);
        cudaMalloc((void**)&devVec3, sizeof(float) * SIZE);

        //Копіюємо дані в пам'ять відеокарти
        cudaMemcpy(devVec1, vec1, sizeof(float) * SIZE,
cudaMemcpyHostToDevice);
        cudaMemcpy(devVec2, vec2, sizeof(float) * SIZE,
cudaMemcpyHostToDevice);

        ...

    }
```

Для виділення пам'яті на відеокарті використовується функція `cudaMalloc`, що має наступний прототип: `cudaError_t cudaMalloc( void** devPtr, size_t count )`, де:

- **devPtr** – вказівник, який містить адресу виділеної пам'яті;
- **count** – розмір пам'яті, що виділяється.

Для копіювання даних в пам'ять відеокарти використовується функція `cudaMemcpy`, яка має наступний прототип: `cudaError_t cudaMemcpy(void* dst, const void* src,size_t count, enum cudaMemcpyKind kind)`, де:

- **dst** – вказівник, що містить адресу місця призначення копіювання (тобто destination, призначення);
- **src** – вказівник, що містить адресу джерела копіювання;
- **count** – розмір ресурсу, який необхідно скопіювати (в байтах);
- **cudaMemcpyKind** – *enum*, що вказує напрямок копіювання (може бути *cudaMemcpyHostToDevice*, *cudaMemcpyDeviceToHost*, *cudaMemcpyHostToHost*, *cudaMemcpyDeviceToDevice*).

Переходимо безпосередньо до виклику ядра.

```
..

        dim3 gridSize = dim3(1, 1, 1);     //Розмір grid'a
        dim3 blockSize = dim3(SIZE, 1, 1); //Розмір блоку

        //Викликаємо ядро
        addVector<<<gridSize,        blockSize>>>(devVec1,
 devVec2, devVec3);

        ...
```

Нам потрібно скопіювати результат розрахунку з відеопам'яті в пам'ять хоста. Але тут є одна особливість – ***асинхронне виконання***, тобто, якщо після виклику ядра почав працювати наступний блок коду (мається на увазі код хоста), то це ще не означає, що GPU виконав розрахунки. Для завершення роботи заданої функції ядра необхідно використовувати засоби синхронізації, наприклад, *events*. Тому перед копіюванням результатів на хост виконується синхронізація потоків GPU. Якщо цього не робити, то ви можете отримати неправильні результати після виконання ядра.

..

```
            cudaEvent_t syncEvent;

            cudaEventCreate(&syncEvent);      //Створюємо event
            cudaEventRecord(syncEvent, 0);    //Записуємо event
            cudaEventSynchronize(syncEvent);  //Синхронізуємо event

            //Отримуємо результати розрахунків
                cudaMemcpy(vec3,   devVec3,   sizeof(float)   *   SIZE,
  cudaMemcpyDeviceToHost);

            ...
```

Event створюється за допомогою функції `cudaEventCreate`, прототип якої має такий вигляд: `cudaError_t cudaEventCreate( cudaEvent_t* event )`, де:

- **event** – вказівник для запису дескриптора події.

Запис event виконується за допомогою функції `cudaEventRecord`, прототип якої виглядає так: `cudaError_t cudaEventRecord( cudaEvent_t event, CUstream stream )`, де:

- **event** – дескриптор event,
- **stream** – номер потоку, в якому працюємо (по-дефолту ***0***).

Синхронізація event виконується функцією `cudaEventSynchronize`. Функція очікує завершення роботи всіх потоків GPU і виклику заданого event, і тільки тоді передає управління керуючій програмі. Прототип функції виглядає так: `cudaError_t cudaEventSynchronize( cudaEvent_t event )`, де:

- **event** – дескриптор event, виклик якого очікується.

Тепер залишається вивести результат на екран і вивільнити ресурси.

..

```
            //Результати розрахунків
            for (int i = 0; i < SIZE; i++)
            {
                printf("Elem  #%i:  %.1f\n",
  i , vec3[i]);
            }
```

```
    cudaEventDestroy(syncEvent);

    cudaFree(devVec1);
    cudaFree(devVec2);
    cudaFree(devVec3);

    delete[] vec1; vec1 = 0;
    delete[] vec2; vec2 = 0;
    delete[] vec3; vec3 = 0;
```

Компілюємо:

```
$ nvcc vector.cu
```

І отримуємо помилку:

```
nvcc warning : The 'compute_20', 'sm_20', and 'sm_21' architectures are deprecated, and may be removed in a future release (U
se -Wno-deprecated-gpu-targets to suppress warning).
vector.cu
vector.cu(58): error: identifier "printf" is undefined

1 error detected in the compilation of "C:/Users/RAINBO~1/AppData/Local/Temp/tmpxft_00000608_00000000-6_vector.cpp4.ii".
```

Компіляція

Ми забули під'єднати бібліотеку **iostream**. Додаємо її у початок файлу та компілюємо знову. Після чого запускаємо:

```
Elem #469: 938.0
Elem #470: 940.0
Elem #471: 942.0
Elem #472: 944.0
Elem #473: 946.0
Elem #474: 948.0
Elem #475: 950.0
Elem #476: 952.0
Elem #477: 954.0
Elem #478: 956.0
Elem #479: 958.0
Elem #480: 960.0
Elem #481: 962.0
Elem #482: 964.0
Elem #483: 966.0
Elem #484: 968.0
Elem #485: 970.0
Elem #486: 972.0
Elem #487: 974.0
Elem #488: 976.0
Elem #489: 978.0
Elem #490: 980.0
Elem #491: 982.0
Elem #492: 984.0
Elem #493: 986.0
Elem #494: 988.0
Elem #495: 990.0
Elem #496: 992.0
Elem #497: 994.0
Elem #498: 996.0
Elem #499: 998.0
Elem #500: 1000.0
Elem #501: 1002.0
Elem #502: 1004.0
Elem #503: 1006.0
Elem #504: 1008.0
Elem #505: 1010.0
Elem #506: 1012.0
Elem #507: 1014.0
Elem #508: 1016.0
Elem #509: 1018.0
Elem #510: 1020.0
Elem #511: 1022.0
```

Результат виконання програми

Весь код виглядає наступним чином:

```
      #i
nclude
<iostream
>
               #define SIZE 512 //довжина вектора
               using namespace std;

               __global__  void  addVector(float*  vector1,  float*  vector2,
        float* result)
               {
                 //Отримуємо id потоку.
                 int idx = threadIdx.x;

                 //Обчислюємо результат.
                 result[idx] = vector1[idx] + vector2[idx];
               }

               __host__ int main()
               {
                 //Виділяємо пам'ять для векторів
                 float* vec1 = new float[SIZE];
                 float* vec2 = new float[SIZE];
                 float* vec3 = new float[SIZE]; //результат

                 //Ініціалізуємо значення векторів
                 for (int i = 0; i < SIZE; i++)
                 {
                   vec1[i] = i;
                   vec2[i] = i;
                 }

                 //Вказівники на створені вектори
                 float* devVec1;
                 float* devVec2;
                 float* devVec3;

                 //Виділяємо пам'ять для вектрів на відеокарті
                 cudaMalloc((void**)&devVec1, sizeof(float) * SIZE);
                 cudaMalloc((void**)&devVec2, sizeof(float) * SIZE);
                 cudaMalloc((void**)&devVec3, sizeof(float) * SIZE);

                 //Копіюємо дані в пам'ять відеокарти
                     cudaMemcpy(devVec1,   vec1,   sizeof(float)   *   SIZE,
        cudaMemcpyHostToDevice);
                     cudaMemcpy(devVec2,   vec2,   sizeof(float)   *   SIZE,
        cudaMemcpyHostToDevice);

                 dim3 gridSize = dim3(1, 1, 1);     //Розмір grid'a
                 dim3 blockSize = dim3(SIZE, 1, 1); //Розмір блоку

                 //Викликаємо ядро
                    addVector<<<gridSize,   blockSize>>>(devVec1,   devVec2,
        devVec3);
                 cudaEvent_t syncEvent;

                 cudaEventCreate(&syncEvent);      //Створюємо event
                 cudaEventRecord(syncEvent, 0);    //Записуємо event
                 cudaEventSynchronize(syncEvent);  //Синхронізуємо event
```

```
            //Отримуємо результати розрахунків
                cudaMemcpy(vec3,   devVec3,   sizeof(float)   *   SIZE,
    cudaMemcpyDeviceToHost);

            for (int i = 0; i < SIZE; i++)
            {
              printf("Elem #%i: %.1f\n", i , vec3[i]);
            }

            cudaEventDestroy(syncEvent);

            cudaFree(devVec1);
            cudaFree(devVec2);
            cudaFree(devVec3);

            delete[] vec1; vec1 = 0;
            delete[] vec2; vec2 = 0;
            delete[] vec3; vec3 = 0;
          }
```

Оцінка витраченого часу на виконання обчислень

Для того, щоб порахувати скільки часу зайняло виконання обрахунків на GPU, можна використати наступний фрагмент коду:

```
    cudaEvent_t start, stop;
    float gpuTime;

    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    cudaEventRecord(start, 0);

    ... //Виклик ядра

    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);
    cudaEventElapsedTime(&gpuTime, start, stop);
    printf("time spent executing by the GPU: %.2f milliseconds\
n", gpuTime);
    cudaEventDestroy(start);
    cudaEventDestroy(stop);
```

Для CPU:

```
    #include <ctime>
    ...

    double begin = clock();

    ... //Виконання операцій
```

```
        double end = clock();
        double elapsed = double(end - begin) /
CLOCKS_PER_SEC * 1000.0;
```

Запускаємо програму і бачимо результат (я збільшив кількість знаків після коми, щоб краще було видно результат).

```
time spent executing by the GPU: 0.1042240 milliseconds
time spent executing by the CPU: 0.0000000 milliseconds
```

Розмір вектора – 512

Збільшуємо розмір вектора до `1_000_000_00` (мільярд) і запускаємо знову.

```
time spent executing by the GPU: 0.1228800 milliseconds
time spent executing by the CPU: 925.0000000 milliseconds
```

Розмір вектора – 1 мільярд

Ми досягнули досить хорошого результату шляхом збільшення кількості елементів у векторі

***Оптимізація коду**:*

- старайтеся використовувати глобальну пам'ять не дуже часто, оскільки це найповільніший тип пам'яті;
- під час роботи з shared-пам'яттю уникайте конфліктів синхронізації;
- не використовуйте код, що містить багато розгалужень;
- старайтеся використовувати якнайменше пам'яті, що значно зменшить затримки.

2.4 Зміст звіту

1. Титульний аркуш, назва роботи.
2. Мета роботи.
3. Завдання до виконання лабораторної роботи.
4. Опис алгоритму розв'язання задачі у вигляді блок-схем або словесне.
5. Програма у вигляді вихідних кодів (з пояснювальними коментарями),
6. Приклади роботи програми на тестових даних.
7. Висновки по роботі.

Електронне Навчальне видання

МЕТОДИЧНІ ВКАЗІВКИ
до лабораторних робіт з дисципліни

«РОЗПІЗНАВАННЯ ОБРАЗІВ НА ОСНОВІ ТЕХНОЛОГІЙ ПРОГРАМУВАННЯ ГРАФІЧНИХ ПРОЦЕСОРІВ»

для студентів денної форми навчання
спеціальності 123 Комп’ютерна інженерія
за освітньо-професійною програмою
«Комп’ютерні інтелектуальні технології»

Упорядник СЕРДЮК Наталія Миколаївна

Відповідальний випусковий О.Г.Руденко

Авторська редакція