МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
«ХАРКІВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

МЕТОДИЧНІ ВКАЗІВКИ

ПРОГРАМУВАННЯ СПЕЦІАЛІЗОВАНИХ ОБЧИСЛЮВАЛЬНИХ СИСТЕМ

до виконання лабораторних робіт
з дисципліни: «Програмування спеціалізованих обчислювальних систем»

для студентів спеціальностей:
174 Автоматизація, комп'ютерно-інтегровані технології та робототехніка
172 Електронні комунікації та радіотехніка

Затверджено
редакційно-видавничою
радою університету,
протокол № 2 від 27.06. 2024 р.

Харків
НТУ «ХПІ»
2024

Методичні вказівки «Програмування спеціалізованих обчислювальних систем» для виконання лабораторних робіт з дисципліни: «Програмування спеціалізованих обчислювальних систем» для студентів спеціальностей 174 – «Автоматизація, комп'ютерно-інтегровані технології та робототехніка», 172 – «Електронні комунікації та радіотехніка» / уклад.: Сальніков Д.В., Євсеєнко О. М., Зуєв А. О. – Харків : НТУ «ХПІ». – 2024. – 45 с.

Укладачі: Д.В. Сальніков, А. О. Зуєв, О. М. Євсеєнко

Рецензент Д. А. Гапон

Кафедра автоматики та управління в технічних системах

ВСТУП

Сучасні обчислювальні системи можна розділити на декілька груп. Серед них програмовані логічні схеми (FPGA), процесорні системи (CPU) та графічні акселератори (GPU). Кожна з цих груп має свої переваги та недоліки. Завдання інженера при їх використанні полягає в правильному виборі системи для задачі та у написанні програмного забезпечення, яке максимально ефективно використовує наявне апаратне забезпечення.

Графічні прискорювачі є важливим компонентом сучасних обчислень завдяки їхній здатності обробляти дані паралельно, що призводить до більш швидких і ефективних обчислень порівняно з традиційними центральними процесорами. Це відкрило шлях для широкого спектра застосувань графічних процесорів для обчислень загального призначення, таких як підвищення продуктивності додатків, прискорення наукових симуляцій і вдосконалення алгоритмів машинного навчання.

Сучасна програмно-апаратна архітектура паралельних обчислень CUDA (Compute Unified Device Architecture) дозволяє істотно збільшити обчислювальну продуктивність завдяки використанню графічних процесорів фірми nVidia. Вона надає можливість включати в текст програм на загальних мовах програмування виклик підпрограм, що виконуються на графічних процесорах, це дає розробнику можливість на свій розсуд організовувати доступ до набору інструкцій графічного прискорювача й керувати його пам'яттю.

1. Методи оптимізації ПЗ

Тема: Оптимізація алгоритмів за допомогою вбудованих (intrinsic) функцій. Спеціалізовані інструкції процесорів.

Мета: Опанувати методи використання вбудованих (intrinsic) функцій для оптимізації програмного забезпечення процесорних архітектур.

Теоретичні відомості. Архітектура CPU

Кожен процесор являє собою комбінацію логічних схем, що послідовно зчитує команди з пам'яті та виконує їх розподіляючи дані між різними частинами процесору.

У логічних схемах, які є основою процесорів, існує невелика затримка між моментом, коли сигнал входить у схему, і моментом, коли він виходить. Ця затримка обмежує максимальну частоту, на якій може працювати процесор, оскільки сигнали потрібно обробляти послідовно. Коли процесор працює на високій частоті, затримки можуть стати настільки значними, що вони перешкоджають подальшому збільшенню швидкості обробки даних.

Щоб вирішити цю проблему та підвищити продуктивність процесора, розробники використовують конвеєризацію. Цей метод дозволяє розділити обробку інструкцій на декілька етапів, які виконуються одночасно в різних частинах процесора, тим самим збільшуючи кількість інструкцій, що обробляються за одиницю часу. Це дозволяє процесору працювати ефективніше, виконуючи різні частини інструкцій в один і той же час.

Розглянемо стадії роботи конвеєру процесора з 5 ступенями:

- IF - стадія завантаження даних інструкції з пам'яті,
- ID - стадія декодування даних інструкції,
- EX - стадія виконання інструкції,
- MEM - стадія виконання операцій з пам'яттю,
- WB - стадія зворотного запису результатів виконання,
- BB - бульбашка.

В Таблиці 1 розглянуто стадії виконання 5 інструкцій. Кожна з інструкцій проходить 5 стадій процесору. Кожна стадія виконується протягом одного такту. Таким чином при максимальній ефективності за 5 тактів буде виконано 5 інструкцій.

Таблиця 1 - 5ти ступеневий конвеєр процесора

Inst	1	2	3	4	5	6	7
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX

В деяких ситуаціях конвеєрний процесор може зупинятися (stall) і створювати так звану бульбашку (pipeline bubble, BB) в конвеєрі, що призводить до одного або кількох циклів, у яких корисні операції не виконуються.

Таблиця 2 - виконання інструкцій в конвеєрі з бульбашкою

	1	2	3	4	5	6	7
IF	I1	I2	I2	I3	I4		
ID		I1	BB	I2	I3	I4	
EX			I1	BB	I2	I3	I4
MEM				I1	BB	I2	I3
WB					I1	BB	I2

В Таблиці 2 у такті 2 процесор не може декодувати фіолетову інструкцію (I2). Це можливо, через те, що процесор визначає, що декодування залежить від результатів, отриманих під час виконання зеленої інструкції. Зелена інструкція може перейти до етапу виконання, а потім до етапу зворотного запису за розкладом, але фіолетова інструкція зупиняється на один цикл на етапі завантаження. Синя інструкція, яка мала бути отримана під час циклу 3, зупиняється на один цикл, як і помаранчева інструкція після неї.

Через бульбашку (BB) схема декодування процесора неактивна під час циклу 3. Його схема виконання неактивна під час циклу 4, а схема зворотного запису — під час циклу 6.

Коли бульбашка виходить із конвеєра (у такті 6), нормальне виконання відновлюється. Але загалом виконання затримується на 1 такт. Щоб повністю виконати чотири інструкції, показані кольорами, знадобиться 9 тактів (такти з 1 по 9), а не 8.

Таким чином для досягнення максимальної ефективності програмного забезпечення необхідно мінімізувати кількість бульбашок в конвеєрі процесору.

Спеціалізовані інструкції процесорів

Сучасні процесорні архітектури мають спеціалізовані інструкції. Найбільш поширені серед них так звані

SIMD (Single instruction, multiple data) інструкції. Такі інструкції включено до складу розширення Intel SSE/AVX, ARM NEON та інших.

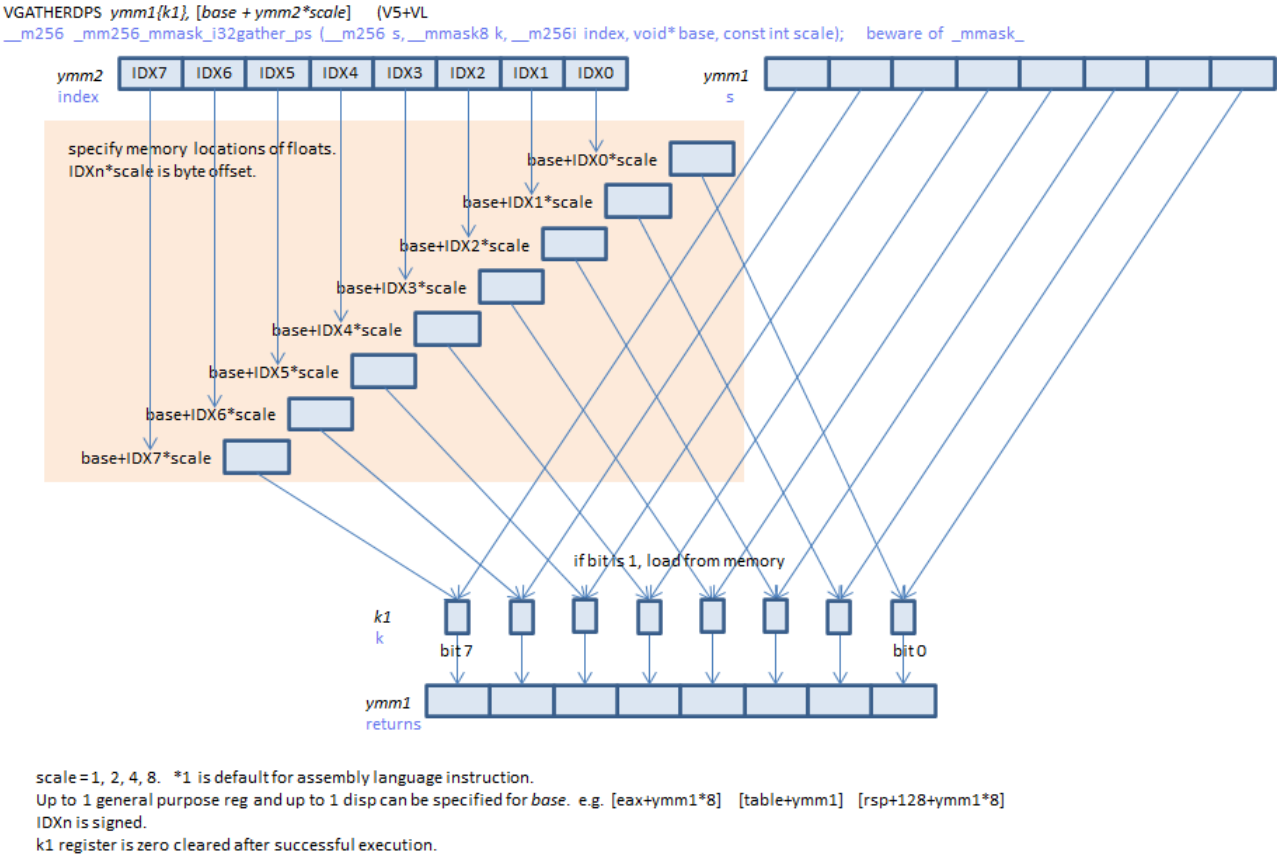


Рисунок 1.1 - Інструкція VGATHERDPS процесорів Intel з розширенням SSE/AVX

SIMD інструкції оперують декількома значеннями одночасно. Наприклад Intel SSE/AVX має спеціалізовані регістри розміром 256 (більш нові процесори 512) біт. Таким чином процес може виконувати операції над 8 числами з плаваючою точкою одинарної точності одночасно.

Крім елементарних операцій на кшталт ADD/SUB та MUL сучасні процесори миють інструкції для виконання специфічних задач. Наприклад, зображену на рис.1.1 для інструкції для операції gather.

Інструкція ***VGATHERDPS DEST, BASE, VINDEX*** використовує регістри:

- BASE - базова адреса,
- VINDEX - зміщення (номер) елементу,
- DEST - регістр для збереження результату.

За одну інструкцію виконується завантаження 8-ми 32-бітних чисел в регістр DEST.

Ход роботи

Для уникнення програмування мовами асемблеру сучасні компілятори надають спеціальні інтринсик функції. Під час компіляції такі функції замінюються ідентичними інструкціями. Використовуючі їх програміст може контролювати спосіб розрахунку тих чи інших виразів.

В той самий час такий метод не потребує ручного контролю регістрів та розподілу пам'яті.

Розглянемо наступну задачу розрахунку суми векторів.

```
#include <iostream>
#include <chrono>
#include <xmmintrin.h>

void add(float *outp, float *a, float *b, size_t len)
{
    // 1. для кожного елемента в діапазоні [0, len)
```

```
    // знайдіть суму 'a' та 'b'
    // результат запишіть в 'outp'
    //
    // 2. використайте:
    // __m128 _mm_add_ps(__m128 a, __m128 b)
}

int main(void) {
    size_t len = 100000;
    float *a = new float[len];
    float *b = new float[len];
    float *c = new float[len];

    // ініціалізуйте 'a' та 'b' випадковими числами

    // виміряйте час виконання функції
    auto start = std::chrono::high_resolution_clock::now();

    add(c, a, b, len);

    auto stop = std::chrono::high_resolution_clock::now();
    auto duration =
      std::chrono::duration_cast<std::chrono::microseconds>
            (stop - start);

    std::cout << "execution took: " << duration.count() << std::endl;

    return 0;
}
```

Створіть новий документ в середовищі Google Colab.

За допомогою директиви ‘%%writefile add.cpp’ створіть файл програми.

Реалізуйте функцію add() використовуючі класичний підхід: цикл, завантаження з пам'яті, оператор '+' та завантаження результату в пам'ять.

Скомпілюйте збережений файл та запустіть його за допомогою команд:

```
!clang++ -O2 add.cpp -o add
!./add
```

Програма виведе час виконання в мікросекундах. Збережіть зазначений час для подальшого порівняння.

Процесори Intel мають спеціалізовані SIMD інструкції. Зокрема:

__m128 _mm_add_ps(__m128 a, __m128 b)

__m256 _mm256_add_ps(__m256 a, __m256 b)

__m512 _mm512_add_ps(__m512 a, __m512 b)

Використовуючи ці функції створіть 3 варіанти функції add. Порівняйте їх роботу. Збережіть асемблерний код кожної функції та порівняйте кількість інструкцій в циклі за допомогою команд:

```
!clang++ -O2 -S add.cpp
!cat add.s
```

Виконайте звіт до лабораторної роботи.

Включіть асемблер виконаних функцій. Побудуйте графіки порівняння швидкості виконання програм для кількості елементів: 1000, 10000, 100000, 1000000, 10000000.

Питання для самостійного опрацювання

- Профілювання програми
- Вимірювання середньої швидкості виконання
- Спеціалізовані команди мікроконтролерних систем.
- Процесорна архітектура VLIW (very long instruction word).

2. Введення в CUDA

Тема: Розподіл та види пам'яті. Ядра CUDA. Компіляція та запуск програм CUDA.

Мета: Опанувати методи написання програм для GPU - ядер CUDA

Теоретичні відомості. Архітектура GPU

В силу того що процесор виконує код загального призначення його оптимізують для виконання великої кількості операцій стрибків для реалізації циклів, умовних блоків, тощо. Це обумовлює велику кількість логіки, що пов'язана з кешуванням даних та інструкцій процесору. Такі архітектурні особливості суттєво ускладнюють роботу конвеєру процесору та знижує можливість використання SIMD інструкцій через складність їх апаратного виконання.

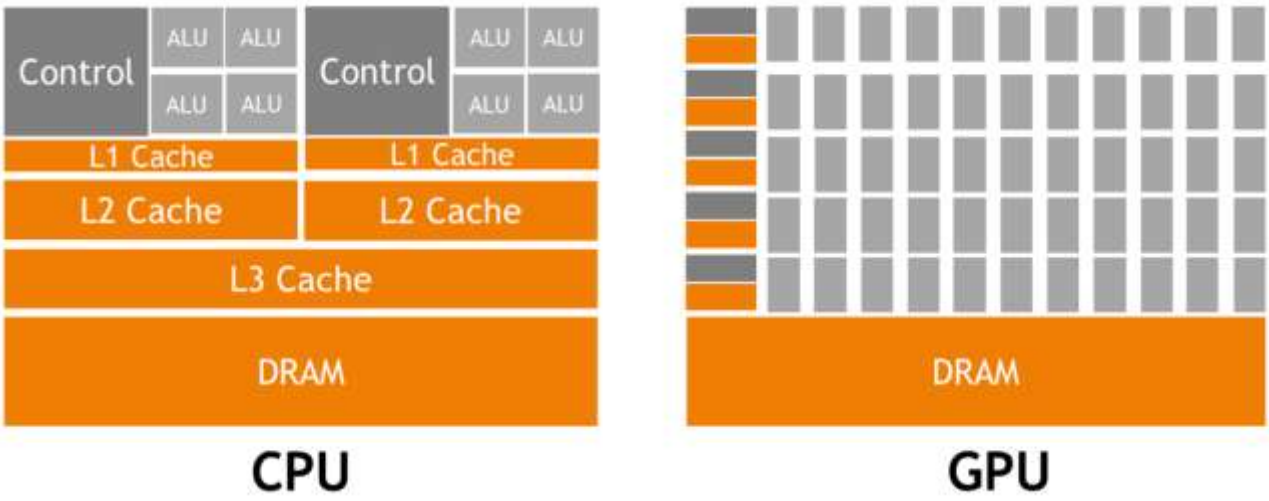


Рисунок 2.1 - Архітектура CPU та GPU.

Сучасний графічний акселератор можна представити як велику кількість досить простих процесорів, що виконують одні і ті самі команди. На відміну від поширених процесорних архітектур, графічні акселератори погано виконують операції розгалуження. Проте, через простоту кожного процесорного юніта,

GPU може включати тисячі таких юнітів, що призводить до суттєво більшої паралельності розрахунків.

Однією з ключових причин, чому CUDA стала такою популярною, є те, що апаратне та програмне забезпечення були розроблені та тісно пов’язані, щоб отримати найкращу продуктивність програми. У зв’язку з цим виникає необхідність показати взаємозв’язок між концепціями програмування програмного забезпечення CUDA та самим дизайном апаратного забезпечення.

Встановлення засобів розробки CUDA

Для встановлення необхідних бібліотек та компілятору CUDA на персональний комп'ютер скористайтесь наступними посиланнями на документацію NVIDIA.

Для Windows:

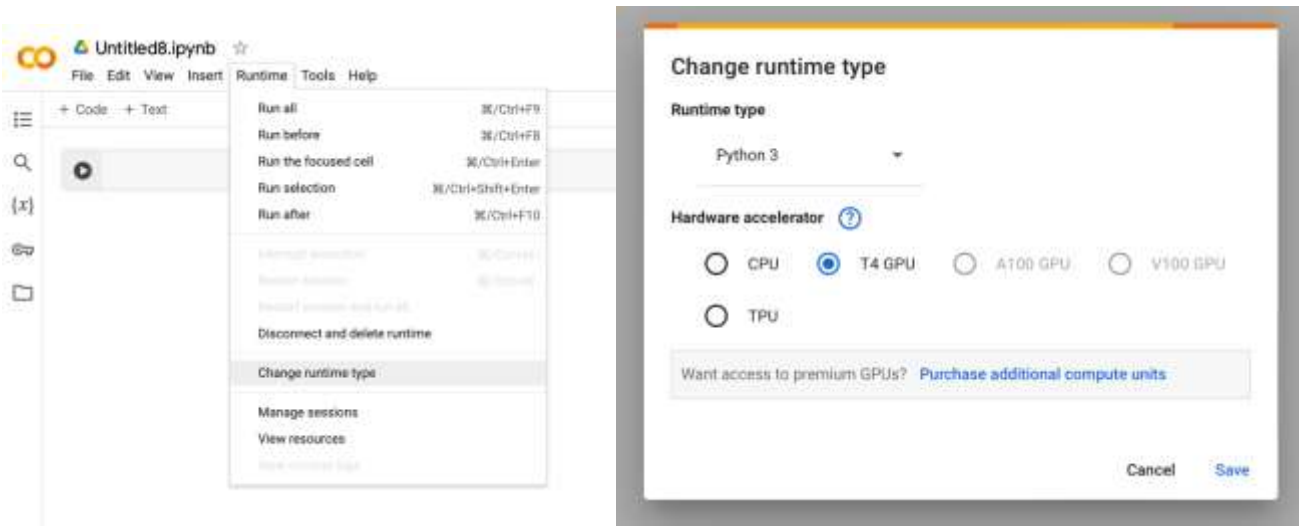
https://docs.nvidia.com/cuda/cuda-installation-guide-microsoft-windows/index.html

Для Linux:

https://docs.nvidia.com/cuda/cuda-installation-guide-linux/index.html

Налаштування Google Colab для розробки CUDA

Для виконання лабораторних робіт в онлайн середовищі Google Colab перейдіть за посиланням: https://colab.research.google.com/. Для роботи необхідно створити аккаунт Google чи використати наявний.



(a). (b)

Рисунок 2.2 - Зміна типу середовища Google Colab

a) Change runtime type, b) T4 GPU

За замовчанням середовище налаштоване для виконання коду Python та прискорювача Google TPU.

Для використання GPU необхідно перейти в меню *Runtime - Change runtime type* як показано на рис. 2.2a.

В меню що з'явилося виберіть T4 GPU (або інший за наявності) як показано на рис. 2.2b Після чого натисніть Save.

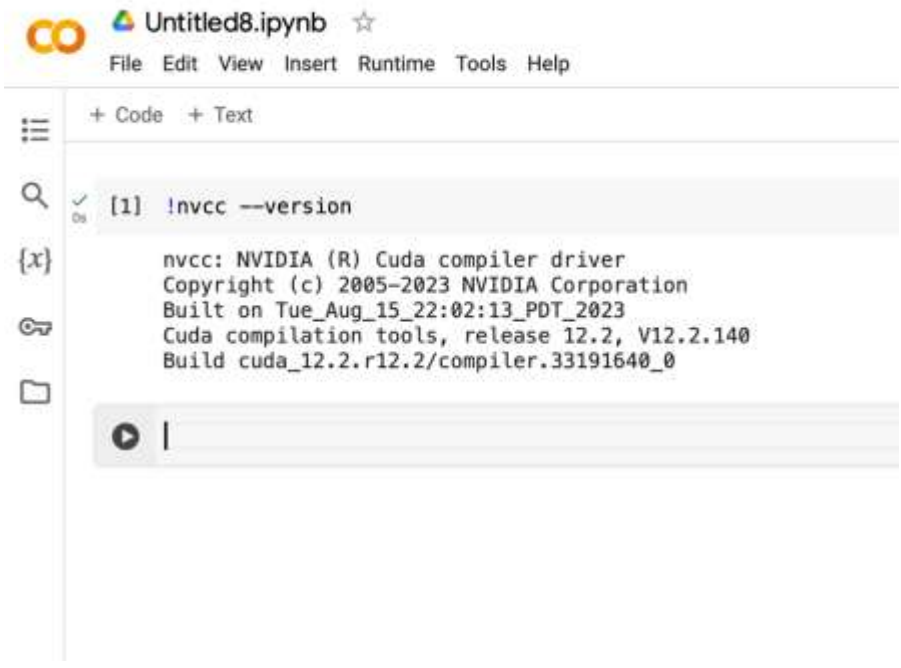


Рисунок 2.3 - Виконання секцій в Google Colab

Ход роботи

Середовище Colab являє собою певну кількість секцій які можуть бути виконані в будь-якому порядку за бажанням користувача.

Для створення нової секції коду натисніть "+ Code". В секції що з'явилася введіть код як показано на рис.. Для виконання натисніть кнопку ліворуч від секції. Секції що починаються з символу "!" виконуються як команда Linux.

Використовуючі директиву ‘%%writefile cuda_example_0.cpp’ створіть файл програми.

```
#include <stdio.h>
#include <stdlib.h>

__global__ void print_from_gpu(void) {
  printf("Hello World! from thread [%d,%d] \
    From device\n", threadIdx.x, blockIdx.x);
}

int main(void) {
  printf("Hello World from host!\n");
  print_from_gpu<<<2,3>>>();
  cudaDeviceSynchronize();
  return 0;
}
```

Скомпілюйте збережений файл та запустіть його за допомогою команд:

```
!nvcc -O2 cuda_example_0.cpp -o cuda_example_0
!./ cuda_example_0
```

Використовуючі директиву ‘%%writefile cuda_example_1.cpp’ створіть файл програми.

```
#include <cuda_runtime.h>
#include <iostream>

__global__ void kernel(uint *A, uint *B, int row) {
  auto x = threadIdx.x / 4;
  auto y = threadIdx.x % 4;
  A[x * row + y] = x;
  B[x * row + y] = y;
}

int main(int argc, char **argv) {
  uint *Xs, *Ys;
  uint *Xs_d, *Ys_d;
  uint SIZE = 4;

  Xs = (uint *)malloc(SIZE * SIZE * sizeof(uint));
  Ys = (uint *)malloc(SIZE * SIZE * sizeof(uint));

  cudaMalloc((void **)&Xs_d, SIZE * SIZE * sizeof(uint));
  cudaMalloc((void **)&Ys_d, SIZE * SIZE * sizeof(uint));

  dim3 grid_size(2, 2, 1);
  dim3 block_size(4 * 4);

  kernel<<<grid_size, block_size>>>(Xs_d, Ys_d, 4);

  cudaMemcpy(Xs, Xs_d, SIZE * SIZE * sizeof(uint),
      cudaMemcpyDeviceToHost);
  cudaMemcpy(Ys, Ys_d, SIZE * SIZE * sizeof(uint),
      cudaMemcpyDeviceToHost);

  cudaDeviceSynchronize();
```

```
  for (int row = 0; row < SIZE; ++row) {
    for (int col = 0; col < SIZE; ++col) {
      std::cout << "[" << Xs[row * SIZE + col] << "|"
                  << Ys[row * SIZE + col]
                << "] ";
    }
    std::cout << "\n";
  }

  cudaFree(Xs_d);
  cudaFree(Ys_d);
  free(Xs);
  free(Ys);
}
```

Скомпілюйте збережений файл та запустіть його за допомогою команд:

```
!nvcc -O2 cuda_example_1.cpp -o cuda_example_1
!./cuda_example_1
```

Використовуючі директиву ‘%%writefile cuda_example_2.cpp’ створіть файл програми.

```
#include <iostream>

void PrintDeviceInfo() {
    int deviceId;
    cudaGetDevice(&deviceId);

    cudaDeviceProp props{};
    cudaGetDeviceProperties(&props, deviceId);
```

```
    std::cout << "Device ID:" << deviceId << std::endl;
    std::cout << "\tName: " << props.name << std::endl;
    std::cout << "\tCompute Capability: " << props.name << std::endl;
    std::cout << "\tmemoryBusWidth: " << props.memoryBusWidth
           << std::endl;
    std::cout << "\tmaxThreadsPerBlock: "
              << props.maxThreadsPerBlock << std::endl;
    std::cout << "\tmaxThreadsPerMultiProcessor: "
              << props.maxThreadsPerMultiProcessor << std::endl;
    std::cout << "\tregsPerBlock: "
              << props.regsPerBlock << std::endl;
    std::cout << "\tregsPerMultiprocessor: "
              << props.regsPerMultiprocessor << std::endl;
    std::cout << "\ttotalGlobalMem: "
              << props.totalGlobalMem << std::endl;
    std::cout << "\tsharedMemPerBlock: "
              << props.sharedMemPerBlock << std::endl;
    std::cout << "\tsharedMemPerMultiprocessor: "
              << props.sharedMemPerMultiprocessor << std::endl;
    std::cout << "\ttotalConstMem: "
              << props.totalConstMem << std::endl;
    std::cout << "\tmultiProcessorCount: "
              << props.multiProcessorCount << std::endl;
    std::cout << "\twarpSize: " << props.warpSize << std::endl;
};
int main() {
    PrintDeviceInfo();
    return 0;
}
```

Скомпілюйте збережений файл та запустіть його за допомогою команд:

```
!nvcc -O2 cuda_example_2.cpp -o cuda_example_2
```

```
!./cuda_example_2
```

Збережіть отримані дані. Виконайте звіт до лабораторної роботи. Зробіть висновок щодо оптимальної кількості потоків та блоків.

Питання для самостійного опрацювання

- Поділ задач у багатопроцесорних системах.
- Типи середовищ Google Colab.
- Спеціалізовані команди мікроконтролерних систем.
- Процесорна архітектура VLIW (very long instruction word).

3. Вимірювання швидкості програмних засобів

Тема: Профілювання за допомогою системи NVIDIA Nsight Systems

Мета: Опанувати методи вимірювання швидкості роботи програмних засобів. Отримати навички використання засобів профілювання NVIDIA Nsight Systems

Ход роботи

Використовуючі директиву ‘%%writefile cuda_profile_example.cpp’ створіть файл програми.

```
#include <assert.h>
#include <stdio.h>

// Standard CUDA API functions
#include <cuda_runtime_api.h>

// Error checking macro
#define cudaCheckError(code)                                                \
  {                                                                         \
    if ((code) != cudaSuccess) {                                            \
      fprintf(stderr, "Cuda failure %s:%d: '%s' \n", __FILE__, __LINE__, \
              cudaGetErrorString(code));                                    \
    }                                                                       \
  }

// Host function for array addition
void add_loop(float *dest, int n_elts, const float *a, const float *b)
{
  for (int i = 0; i < n_elts; i++) {
    dest[i] = a[i] + b[i];
  }
}
```

```
// Device kernel for array addition.
__global__ void add_kernel(float *dest, int n_elts, const float *a,
                           const float *b)
{
  int index = blockIdx.x * blockDim.x + threadIdx.x;
  if (index >= n_elts) return;

  dest[index] = a[index] + b[index];
}

int main()
{
  const int ARRAY_LENGTH = 100;

  // Generate some data on the host
  float host_array_a[ARRAY_LENGTH];
  float host_array_b[ARRAY_LENGTH];
  float host_array_dest[ARRAY_LENGTH];

  for (int i = 0; i < ARRAY_LENGTH; i++) {
    host_array_a[i] = 2 * i;
    host_array_b[i] = 2 * i + 1;
  }

  // Allocate device memory
  float *device_array_a, *device_array_b, *device_array_dest;
  cudaCheckError(cudaMalloc(&device_array_a, sizeof(host_array_a)));
  cudaCheckError(cudaMalloc(&device_array_b, sizeof(host_array_b)));
  cudaCheckError(cudaMalloc(&device_array_dest, sizeof(host_array_dest)));

  // Transfer data to device
  cudaCheckError(cudaMemcpy(device_array_a, host_array_a,
      sizeof(host_array_a), cudaMemcpyHostToDevice));
  cudaCheckError(cudaMemcpy(device_array_b, host_array_b,
      sizeof(host_array_b), cudaMemcpyHostToDevice));

  // Calculate lauch configuration
```

```
  const int BLOCK_SIZE = 128;
  int n_blocks = (ARRAY_LENGTH + BLOCK_SIZE - 1) / BLOCK_SIZE;

  // Add arrays on device
  add_kernel<<<BLOCK_SIZE, n_blocks>>>(device_array_dest, ARRAY_LENGTH,
                                       device_array_a, device_array_b);

  // Meanwhile, add arrays on the host, for comparison
  add_loop(host_array_dest, ARRAY_LENGTH, host_array_a, host_array_b);

  // Copy result back to host and compare
  float host_array_tmp[ARRAY_LENGTH];
  cudaCheckError(cudaMemcpy(host_array_tmp, device_array_dest,
                            sizeof(host_array_tmp), cudaMemcpyDeviceToHost));
  for (int i = 0; i < ARRAY_LENGTH; i++) {
    assert(host_array_tmp[i] == host_array_dest[i]);
    printf("%g + %g = %g\n", host_array_a[i], host_array_b[i],
           host_array_tmp[i]);
  }

  return 0;
}
```

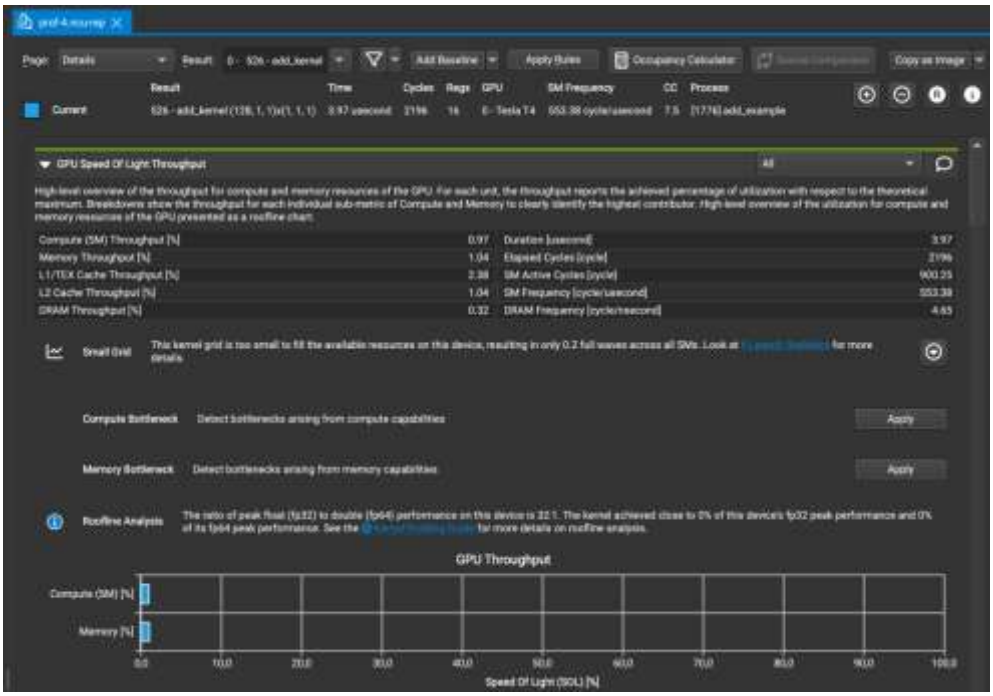


Рисунок 3.1 – GPU Speed Of Light Throughput

Скомпілюйте збережений файл та запустіть його за допомогою команд:

```
!nvcc -O2 cuda_profile_example.cpp -o cuda_profile_example
!./cuda_profile_example
```

Запустіть программу профілювання за допомогою наступних команд.

```
!rm -rf prof.ncu-rep
!ncu -o prof --set full ./cuda_profile_example
```

Завантажте файл prof.ncu-rep та відкрийте за допомогою NVIDIA Nsight Compute.

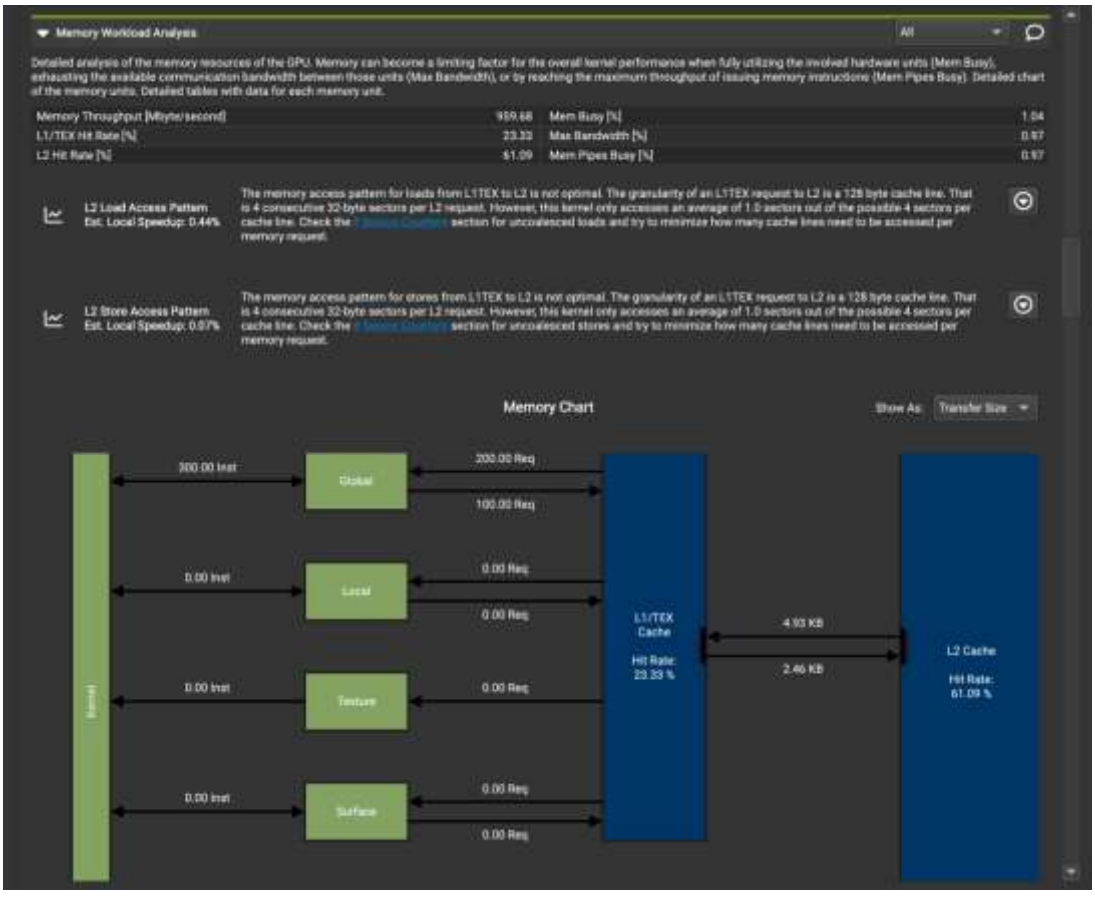


Рисунок 3.2 – GPU Memory Workload Analysis

На сторінці “Details” Перегляньте розділі звіту профілювання програми. Зокрема звіт пропускної здатності (Рис. 3.1), звіт профілювання підсистеми пам’яті (Рис 3.2) та графік граничної пропускної здатності (Рис 3.3).

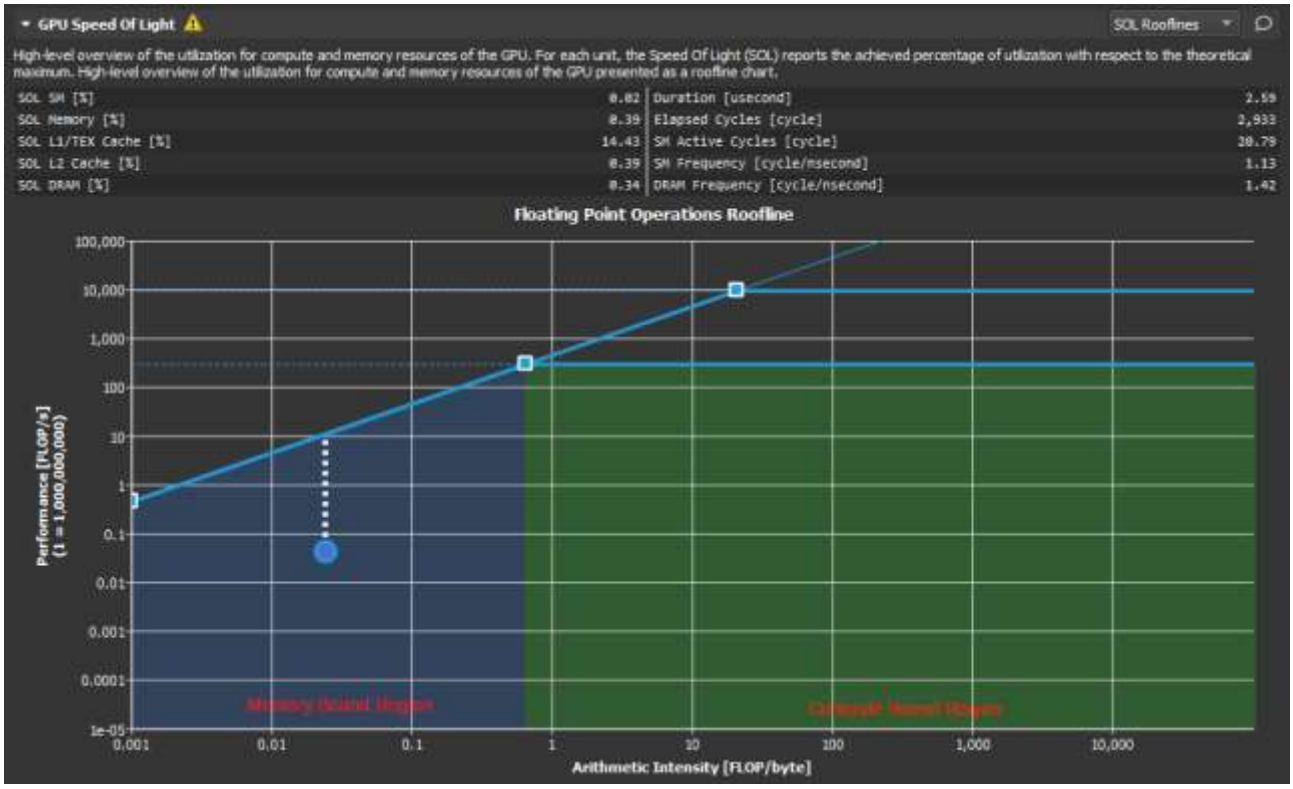


Рисунок 3.3 – GPU Bound region

Повторіть 5 експериментів змінюючи змінні `ARRAY_LENGTH` та `BLOCK_SIZE` згідно таблиці 1.

Таблиця 3.1 – Варіанти

Варіант	ARRAY_LENGTH	BLOCK_SIZE	Варіант	ARRAY_LENGTH	BLOCK_SIZE
1	100-10000	10-1000	11	400-25000	40-5000
2	200-20000	5-500	12	1400-25000	10-3000
3	300-30000	20-2000	13	600-35000	30-1000
4	400-40000	30-3000	14	200-20000	30-1000
5	500-50000	10-3000	15	2400-250000	40-100
6	600-30000	40-1000	16	10-15000	40-1000
7	500-20000	40-5000	17	20-3000	5-500
8	400-25000	30-1000	18	300-30000	100-300
9	300-35000	50-2000	19	100-10000	10-1000
10	10-15000	40-100	20	500-20000	30-3000

Збережіть кількість запитів в пам’ять, пропускну здатність пам’яті та графіки. Оформіть звіт.

Питання для самостійного опрацювання

- Вплив кількості блоків на продуктивність програми.
- Архітектурні особливості графічних прискорювачів різних поколінь.
- Спеціалізовані команди графічних прискорювачів.
- Система NVIDIA Nsight System

4. Техніки оптимізації виконання програм (Частина 1)

Тема: Дослідження продуктивності алгоритму множення матриць.

Мета: Дослідити особливості алгоритму множення матрирць за допомогою програмного пакету NVIDIA Nsight Systems. Порівняти продуктивність власного алгоритму з алгоритмом бібліотеки CuDNN.

Ход роботи

Розглянемо алгоритм множення матриць (general matrix multiplication), що є одним з найбільш вживаним в області розрахунків штучного інтелекту.

Математично алгоритм можна представити наступним чином:

$$C = \alpha AB + \beta C, \quad (1)$$

де A, B – вхідні матриці;

C – результат множення;

α та β – скалярні коефіцієнти.

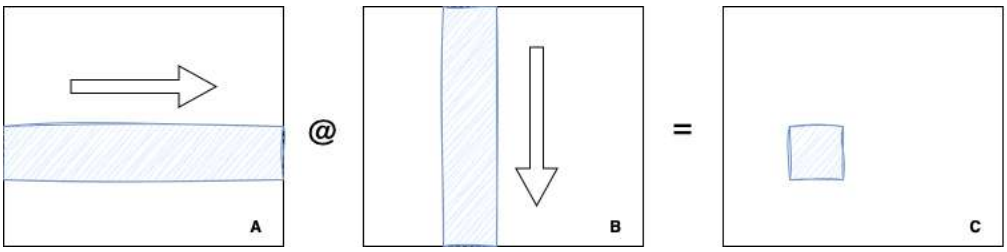


Рисунок 4.1 – Діаграма розташування даних в пам`яті

Розташування динних в пам’яті розглянуто на рисунку 4.1. Для обчислення одного елементу вихідної матриці необхідно зчитати один рядок та один стовпець даних з вхідних матриць, провести обчислення та виконати збереження результату до пам’яті.

Розглянемо найпростіший алгоритм множення з використанням графічного прискорювача.

Використовуючи директиву ‘%%writefile cuda_gemm.cpp’ створіть файл програми.

```
#include <sys/time.h>
#include <iostream>
#include <cuda_runtime.h>
#include <cublas_v2.h>

#define CEIL_DIV(M, N) ((M) + (N)-1) / (N)

void randomize_matrix(float *mat, int N) {
    struct timeval time;
    gettimeofday(&time, NULL);
    srand(time.tv_usec);
    for (int i = 0; i < N; i++) {
        float tmp = (float) (rand() % 5) + 0.01 * (rand() % 5);
        tmp = (rand() % 2 == 0) ? tmp : tmp * (-1.);
        mat[i] = tmp;
    }
}

#define cudaCheckError(code)                                          \
  {                                                                   \
    if ((code) != cudaSuccess) {                                      \
      fprintf(stderr, "Cuda failure %s:%d: '%s' \n", __FILE__, __LINE__, \
              cudaGetErrorString(code));                              \
    }                                                                 \
  }

void sgemm_cpu(int M, int N, int K, float alpha, const float *A,
                            const float *B, float beta, float *C)
{
    // A(M, K) * B(K, N) = C(M, N)
    for (int m=0; m<M; m++)
    {
        for (int n=0; n<N; n++)
        {
```

```
            float tmp = 0.0;
            for (int i = 0; i < K; ++i) {
                tmp += A[m * K + i] * B[i * N + n];
            }

            C[m * N + n] = alpha * tmp + beta * C[m * N + n];
        }
    }
}

__global__ void sgemm_naive(int M, int N, int K, float alpha, const float *A,
                            const float *B, float beta, float *C) {
  // compute position in C that this thread is responsible for
  const uint x = blockIdx.x * blockDim.x + threadIdx.x;
  const uint y = blockIdx.y * blockDim.y + threadIdx.y;

  // `if` condition is necessary for when M or N aren't multiples of 32.
  if (x < M && y < N) {
    float tmp = 0.0;
    for (int i = 0; i < K; ++i) {
      tmp += A[x * K + i] * B[i * N + y];
    }
    // C = α*(A@B)+β*C
    C[x * N + y] = alpha * tmp + beta * C[x * N + y];
  }
}

int main(int argc, char** argv)
{
    if (argc != 2) {
        std::cout << "Please select a kernel\n";
        return -1;
    }

    cudaSetDevice(0);

    int kernelNum = std::atoi(argv[1]);
```

```
std::cout << "using kernel #" << kernelNum << std::endl;

float *A = NULL, *B = NULL, *C = NULL, *C_ref = NULL;  //host matrices
float *dA = NULL, *dB = NULL, *dC = NULL;              //device matrices

// A(M, K) * B(K, N) = C(M, N)
int M = 32;
int K = 32;
int N = 64;

float alpha = 1.0, beta = 0.; //two arbitary input parameters，C=α*AB+β*C

A = (float *) malloc(sizeof(float) * M * K);
B = (float *) malloc(sizeof(float) * K * N);
C = (float *) malloc(sizeof(float) * M * N);
C_ref = (float *) malloc(sizeof(float) * M * N);

randomize_matrix(A, M*N);
randomize_matrix(B, K*N);
randomize_matrix(C, M*N);

cudaCheckError(cudaMalloc((void **) &dA, sizeof(float) * M * N));
cudaCheckError(cudaMalloc((void **) &dB, sizeof(float) * K * N));
cudaCheckError(cudaMalloc((void **) &dC, sizeof(float) * M * N));

cudaCheckError(cudaMemcpy(dA, A,
  sizeof(float) * M * N, cudaMemcpyHostToDevice));
cudaCheckError(cudaMemcpy(dB, B,
  sizeof(float) * K * N, cudaMemcpyHostToDevice));
cudaCheckError(cudaMemcpy(dC, C,
  sizeof(float) * M * N, cudaMemcpyHostToDevice));


// run reference
sgemm_cpu(M, N, K, alpha, A, B, beta, C_ref);

float elapsed_time;
```

```
cudaEvent_t beg, end;
cudaEventCreate(&beg);
cudaEventCreate(&end);

cudaEventRecord(beg);

switch (kernelNum)
{
case 0:
{
    // create as many blocks as necessary to map all of C
    dim3 gridDim(CEIL_DIV(M, 32), CEIL_DIV(N, 32), 1);
    // 32 * 32 = 1024 thread per block
    dim3 blockDim(32, 32, 1);
    // launch the asynchronous execution of the kernel on the device
    // The function call returns immediately on the host
    sgemm_naive<<<gridDim, blockDim>>>(M, N, K, alpha, dA, dB, beta, dC);
}
    break;

case 1:
{
    cublasHandle_t handle;
    if (cublasCreate(&handle)) {
        printf("Create cublas handle error.\n");
        exit(EXIT_FAILURE);
    };
    cublasSgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N,
        N, M, K, &alpha, dB, N, dA, K, &beta, dC, N);
}
break;

default:
    std::cout << "undefined kernel\n";
    return -2;
    break;
};
```

```
    cudaEventRecord(end);
    cudaEventElapsedTime(&elapsed_time, beg, end);
    std::cout << "elapsed time: " << elapsed_time << std::endl;

    cudaDeviceSynchronize();

    cudaCheckError(cudaMemcpy(C, dC,
      sizeof(float) * M * N, cudaMemcpyDeviceToHost));


    //compare ref
    for (int i=0; i<M*N; i++)
    {
        float diff = std::fabs(C[i] - C_ref[i]);
        if (diff > 1e-3)
        {
            std::cout << "error at #" << i << ": "
                  << C[i] << " vs " << C_ref[i] << "|" << diff << std::endl;
            return -3;
        }
    }

    std::cout << "PASS.\n";

    std::cout << "free cuda mem." << std::endl;
    cudaFree(dA);
    cudaFree(dB);
    cudaFree(dC);
    std::cout << "free host mem." << std::endl;
    free(A);
    free(B);
    free(C);
    free(C_ref);

    return 0;
}
```

Скомпілюйте збережений файл та запустіть його за допомогою команд:

```
!nvcc -O2 cuda_gemm.cpp -o cuda_gemm
!./cuda_gemm 0
!./cuda_gemm 1
```

Запустіть програму профілювання за допомогою наступних команд.

```
!rm -rf gemm_prof.ncu-rep
!ncu -o gemm_prof --set full ./cuda_gemm 0
!ncu -o gemm_prof --set full ./cuda_gemm 1
```

Завантажте файл gemm_prof.ncu-rep та проаналізуйте за допомогою NVIDIA Nsight Compute.

Виконайте профілювання для наступних розмірів матриць та додайте дані профілювання у вигляді графіка в звіт.

Таблиця 4.1. Розміри матриць для аналізу

N	128	256	512	1024	2048	4096	256	512	2048
M	128	256	512	1024	2048	4096	2048	512	512
K	128	256	512	1024	2048	4096	512	2048	512

Питання для самостійного опрацювання

- Вплив розміру матриць на продуктивність
- Розрахунок пропускної здатності алгоритму
- Оптимізація шляхом використання інтринсик функцій

5. Техніки оптимізації виконання програм (Частина 2)

Тема: Дослідження продуктивності алгоритму множення матриць із оптимізованим доступом до пам’яті.

Мета: Дослідити особливості алгоритму множення матрирць за допомогою програмного пакету NVIDIA Nsight Systems. Порівняти продуктивність алгоритму з алгоритмом бібліотеки CuDNN та власним алгоритмом.

Ход роботи

Алгоритм що було реалізовано в роботі 4 виконує зчитування цілого рядку матриці A в кожному потоці GPU. В той самий час кожен потік зчитує дані однієї колонки матриці B.

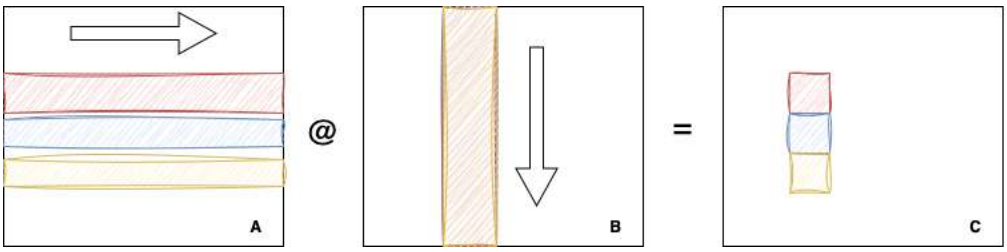


Рисунок 5.1 – Діаграма розташування даних в пам`яті (алгоритм 1)

На рисунку 5.1 зображено роботу 3 потоків GPU. Заштриховані області позначають області пам’яті що зчитує кожен потік.

Для підвищення швидкодії оптимізуємо операції роботи з пам’яттю. А саме порядок операцій.

GPU підтримує завантаження з пам’яті даних розміром 32, 64 та 128 Байт. Таким чином, якщо кожен потік GPU виконує завантаження одного 32бітного (4 Байти) значення, є можливість об’єднати завантаження 32-ох значень (32x4B=128B) в одну транзакцію пам’яті.

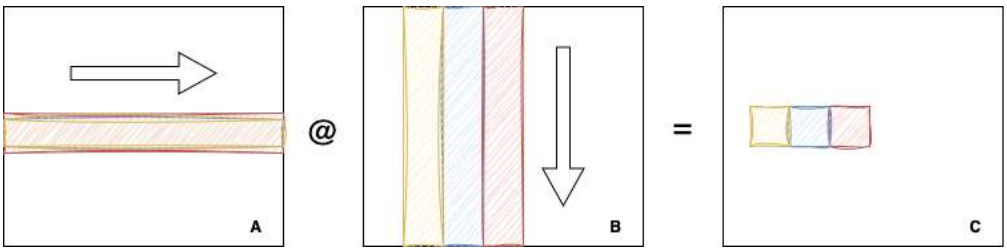


Рисунок 5.2 – Діаграма розташування даних в пам’яті (алгоритм 2)

На рисунку 5.2 зображено роботу 3 потоків GPU. Заштриховані області позначають області пам’яті що зчитує кожен потік.

В даному випадку з матриці A зчитаються дані що розташовані підряд. Кожен потік зчитує один і той самий рядок. Тому транзакції будуть об’єднані.

В той самий час кожен потік вичитує дані розташовані в пам’яті підряд з матриці B. Таки чином транзакції будуть об’єднані.

Модифікуйте програму розроблену в роботі 4 для виконання наступного алгоритму множення.

```
template <const uint BLOCKSIZE>
__global__ void sgemm_global_mem_coalesce(int M, int N, int K, float alpha,
                                          const float *A, const float *B,
                                          float beta, float *C) {
  const int cRow = blockIdx.x * BLOCKSIZE + (threadIdx.x / BLOCKSIZE);
  const int cCol = blockIdx.y * BLOCKSIZE + (threadIdx.x % BLOCKSIZE);

  // if statement is necessary to make things work under tile quantization
  if (cRow < M && cCol < N) {
    float tmp = 0.0;
    for (int i = 0; i < K; ++i) {
      tmp += A[cRow * K + i] * B[i * N + cCol];
    }
    C[cRow * N + cCol] = alpha * tmp + beta * C[cRow * N + cCol];
  }
}
```

Викликати алгоритм можна наступним чином:

```
dim3 gridDim(CEIL_DIV(M, 32), CEIL_DIV(N, 32));
dim3 blockDim(32 * 32);
sgemm_global_mem_coalesce<32>
    <<<gridDim, blockDim>>>(M, N, K, alpha, A, B, beta, C);
```

Скомпілюйте збережений файл та запустіть його за допомогою команд:

```
!nvcc -O2 cuda_gemm_2.cpp -o cuda_gemm_2
!./cuda_gemm_2
```

Запустіть програму профілювання за допомогою наступних команд.

```
!rm -rf gemm_prof_2.ncu-rep
!ncu -o gemm_prof_2 --set full ./cuda_gemm
```

Завантажте файл gemm_prof_2.ncu-rep та проаналізуйте за допомогою NVIDIA Nsight Compute.

Виконайте профілювання для розмірів матриць з таблиці 4.1 та додайте дані профілювання у вигляді графіка в звіт.

Питання для самостійного опрацювання

- Вплив розміру матриць на продуктивність
- Види пам’яті GPU
- Memory-bound та compute-bound програмне забеспечення
- Розрахунок пропускної здатності алгоритму

6. Техніки оптимізації виконання програм (Частина 3)

Тема: Дослідження продуктивності алгоритму множення матриць з використанням кеш пам’яті.

Мета: Дослідити особливості алгоритму множення матриць за допомогою програмного пакету NVIDIA Nsight Systems. Порівняти продуктивність алгоритму з алгоритмом бібліотеки CuDNN та власним алгоритмом.

Ход роботи

На рисунку 6.1 зображено діаграму множення матриць. Для обчислення одного вихідного значення кожен потік GPU виконує зчитування замальованої області даних.

Для підвищення швидкодії необхідно оптимізувати доступ до пам’яті.

Водночас з глобальною пам’яттю в GPU є набагато менший за об’ємом регіон пам’яті - спільна пам'ять (shared memory, SMEM). Оскільки спільна пам’ять розташована на кристалі, вона має набагато нижчу затримку та вищу пропускну здатність, ніж глобальна пам’ять. Залежно від покоління GPU кожен блок може отримати доступ до певної кількості КБ спільної пам’яті.

В даній роботі спільна пам’ять використовується для зберігання частини даних матриць A та B. Завдяки цьому дані зчитуються підряд оптимальними транзакціями та зберігається в спільну пам’ять.

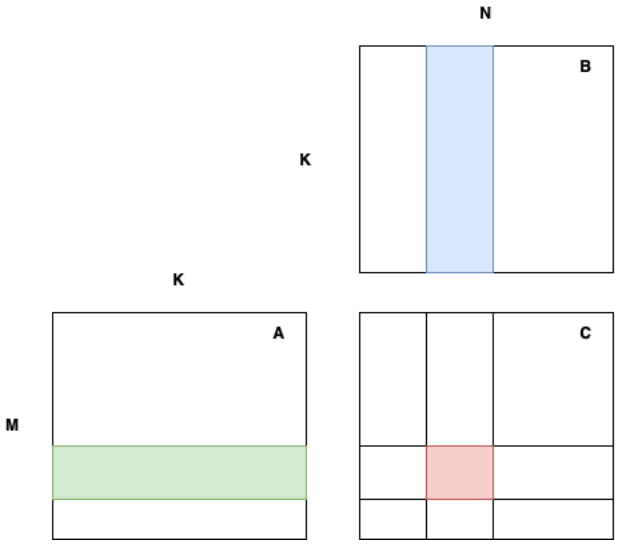


Рисунок 6.1 – Алгоритм множення матриць з кеш пам’яттю

Модифікуйте програму розроблену в роботі 4 для виконання наступного алгоритму множення.

```
template <const int BLOCKSIZE>
__global__ void sgemm_shared_mem_block(int M, int N, int K, float alpha,
                                       const float *A, const float *B,
                                       float beta, float *C) {
  // the output block that we want to compute in this threadblock
  const uint cRow = blockIdx.x;
  const uint cCol = blockIdx.y;

  // allocate buffer for current block in fast shared mem
  // shared mem is shared between all threads in a block
  __shared__ float As[BLOCKSIZE * BLOCKSIZE];
  __shared__ float Bs[BLOCKSIZE * BLOCKSIZE];

  // the inner row & col that we're accessing in this thread
  const uint threadCol = threadIdx.x % BLOCKSIZE;
  const uint threadRow = threadIdx.x / BLOCKSIZE;

  // advance pointers to the starting positions
  A += cRow * BLOCKSIZE * K;                    // row=cRow, col=0
  B += cCol * BLOCKSIZE;                        // row=0, col=cCol
  C += cRow * BLOCKSIZE * N + cCol * BLOCKSIZE; // row=cRow, col=cCol

  float tmp = 0.0;
  for (int bkIdx = 0; bkIdx < K; bkIdx += BLOCKSIZE) {
    // Have each thread load one of the elements in A & B
    // Make the threadCol (=threadIdx.x) the consecutive index
    // to allow global memory access coalescing
    As[threadRow * BLOCKSIZE + threadCol] = A[threadRow * K + threadCol];
    Bs[threadRow * BLOCKSIZE + threadCol] = B[threadRow * N + threadCol];

    // block threads in this block until cache is fully populated
    __syncthreads();
    A += BLOCKSIZE;
    B += BLOCKSIZE * N;
```

```
    // execute the dotproduct on the currently cached block
    for (int dotIdx = 0; dotIdx < BLOCKSIZE; ++dotIdx) {
      tmp += As[threadRow * BLOCKSIZE + dotIdx] *
             Bs[dotIdx * BLOCKSIZE + threadCol];
    }
    // need to sync again at the end, to avoid faster threads
    // fetching the next block into the cache before slower threads are done
    __syncthreads();
  }
  C[threadRow * N + threadCol] =
      alpha * tmp + beta * C[threadRow * N + threadCol];
}
```

Скомпілюйте збережений файл та запустіть його за допомогою команд:

```
!nvcc -O2 cuda_gemm_3.cpp -o cuda_gemm_3
!./cuda_gemm_3
```

Запустіть програму профілювання за допомогою наступних команд.

```
!rm -rf gemm_prof_3.ncu-rep
!ncu -o gemm_prof_3 --set full ./cuda_gemm
```

Завантажте файл gemm_prof_3.ncu-rep та проаналізуйте за допомогою NVIDIA Nsight Compute.

Виконайте профілювання для розмірів матриць з таблиці 4.1 та додайте дані профілювання у вигляді графіка в звіт.

Питання для самостійного опрацювання

- Вплив розміру матриць на продуктивність
- Розрахунок пропускної здатності алгоритму
- Структурна модель пам’яті GPU

7. Використання бібліотек CUDA

Тема: Створення програмних засобів з використанням бібліотеки CuDNN

Мета: Опанувати методи використання бібліотек спеціалізованих функцій що використовують графічні процесори.

Ход роботи

Нейронні мережі, комп'ютерне бачення, штучний інтелект і системи рекомендацій призвели до таких проривів, як автономні транспортні засоби та інтелектуальні голосові помічники. Фреймворки глибокого навчання NVIDIA з прискоренням графічного процесора прискорюють час навчання цим технологіям, скорочуючи багатоденні сесії лише до кількох годин.

cuDNN надає основні бібліотеки, необхідні для високопродуктивних розрахунків з низькою затримкою для глибоких нейронних мереж на графічних прискорювачах NVIDIA.

За допомогою директиви *'%%writefile cudnn_example.cpp'* створіть файл програми.

```
#include <iostream>
#include <cuda_runtime.h>
#include <cudnn.h>

int main(int argc, char** argv)
{
    int numGPUs;
    cudaGetDeviceCount(&numGPUs);
    std::cout << "Found " << numGPUs << " GPUs. Use #0." << std::endl;
    cudaSetDevice(0);
    int device;
    struct cudaDeviceProp devProp;
    cudaGetDevice(&device);
```

```
cudaGetDeviceProperties(&devProp, device);
std::cout << "Compute capability: "
  << devProp.major << "." << devProp.minor << std::endl;

cudnnHandle_t handle_;
cudnnCreate(&handle_);
std::cout << "Create cuDNN handle" << std::endl;

// create the tensor descriptor
cudnnDataType_t dtype = CUDNN_DATA_FLOAT;
cudnnTensorFormat_t format = CUDNN_TENSOR_NCHW;
int n = 1, c = 1, h = 1, w = 10;
int NUM_ELEMENTS = n*c*h*w;
cudnnTensorDescriptor_t x_desc;
cudnnCreateTensorDescriptor(&x_desc);
cudnnSetTensor4dDescriptor(x_desc, format, dtype, n, c, h, w);

// create the tensor
float *x;
cudaMallocManaged(&x, NUM_ELEMENTS * sizeof(float));
for(int i=0;i<NUM_ELEMENTS;i++) x[i] = i * 1.00f;
std::cout << "Original array: ";
for(int i=0;i<NUM_ELEMENTS;i++) std::cout << x[i] << " ";

// create activation function descriptor
float alpha[1] = {1};
float beta[1] = {0.0};
cudnnActivationDescriptor_t sigmoid_activation;
cudnnActivationMode_t mode = CUDNN_ACTIVATION_SIGMOID;
cudnnNanPropagation_t prop = CUDNN_NOT_PROPAGATE_NAN;
cudnnCreateActivationDescriptor(&sigmoid_activation);
cudnnSetActivationDescriptor(sigmoid_activation, mode, prop, 0.0f);
```

```
    cudnnActivationForward(
        handle_, sigmoid_activation, alpha,
        x_desc, x, beta, x_desc, x
    );

    cudnnDestroy(handle_);
    std::cout << std::endl << "Destroy cuDNN handle." << std::endl;
    std::cout << "Output array: ";
    for(int i=0;i<NUM_ELEMENTS;i++) std::cout << x[i] << " ";
    std::cout << std::endl;
    cudaFree(x);
    return 0;
}
```

Виконайте компіляцію коду прогами за допомогою команди:

```
!nvcc cuda_example.cpp -o cuda_example  -lcudnn
```

Запустіть знегерорваний файл за допомогою наступної команди:

```
!./cuda_example
```

Збережіть вивід програми та додайте в звіт до роботи.

Змініть функцію за варіантом та порівняйте результат з референтним.

Таблиця 7.1 – Варіанти

Варіант	Вхідний масив	Функція активації	Варіант	Вхідний масив	Функція активації
1	[0:1:100]	Sigmoid	11	[10:0.33;100]	Sigmoid
2	[0:0.1;20]	Relu	12	[10:0.9;1500]	Relu
3	[0:0.2;30]	Tanh	13	[30:0.5;250]	Tanh
4	[0:0.5;300]	Clipped relu	14	[20:0.3;150]	Clipped relu

5	[0:0.1;15]	Elu	15	[10:0.1;150]	Elu
6	[10:0.1;150]	Sigmoid	16	[0:0.1;15]	Sigmoid
7	[20:0.3;150]	Relu	17	[0:0.5;300]	Relu
8	[30:0.5;250]	Tanh	18	[0:0.2;30]	Tanh
9	[10:0.9;1500]	Clipped relu	19	[0:0.1;20]	Clipped relu
10	[10:0.33;100]	Elu	20	[0:1:100]	Elu

Питання для самостійного опрацювання

- Бібліотека CuBLAS, її призначення та використання
- Види зв’язування бібліотек
- Графи операцій в бібліотеці CuDNN

Контрольні питання

1. Яка область використання розрахунків на GPU? Чим це вигідніше ніж розрахунки з використанням мікропроцесорних архітектур?

2. Що таке векторні інструкції? Які векторні інструкції ви знаєте? Назвість мікропроцесорні архітектури що їх використовують.

3. Як працює конвеєр мікропроцесору? Як структура конвеєру впливає на швидкодію?

4. Опишіть архітектуру графічних прискорювачів. Чим вона відрізняється від архітектури процесорів?

5. Що таке warp, sm, block та thread у сучасних графічних прискорювачах?

6. Чи можливо використовувати векторні інструкції для програмування GPU? Якщо так, які векторні інструкції?

7. Як розраховується теоретична можлива швидкодія алгоритму? Який вплив на розрахунок має кількість арифметичних операцій алгоритму?

8. Які види пам’яті мають графічні прискорювачі?

9. За допомогою якого програмного забезпечення аналізують швидкодію GPU алгоритмів? Опишіть алгоритм процедури аналізу.

10. Як виконується синхронізація потоків GPU між собою? Чи може поток отримати доступ до даних іншого потоку?

11. Як порядок доступу до даних в пам’яті вплаває на швидкодію? Яким чином можна оптимізувати порядок? Поясніть на прикладі алгоритму множення матриць.

12. Чим відрізняють різні GPU один від одного? Які з параметрів та яким чином пов’язані з швидкодією?

13. Якими мовами програмування можна створювати програми для GPU? Які недоліки та переваги вони мають?

14. Які методи досягнення максимальної продуктивності алгоритму ви знаєте? Чи всі з них можна застосувати до будьякої програми?

15. Які бібліотеки для GPU ви знаєте? Яким чином їх можна використовувати?

8. Список рекомендованої літератури

1. Зуєв А.О., Гапон Д.А., Денисенко М.А. Методичні вказівки до виконання самостійних робіт «Основи програмування на мові С++» Х.: 2022, 45с. наказ НТУ “ХПІ” №3 26.10.2022р. поз. 342. http://repository.kpi.kharkov.ua/handle/KhPI-Press/60574

2. Stroustrup, Bjarne. The C++ programming language / Bjarne Stroustrup.— Fourth edition. Published by Pearson Education, Inc. 2013. 1360 p.

3. Грицюк Ю., Рак Т. Програмування мовою С++. Навчальний посібник. / Юрій Грицюк, Тарас Рак. Львів. Вид-во ЛДУ БЖД 2011. 290 с.

4. Jaegeun Han. Learn CUDA Programming / Jaegeun Han, Bharatkumar Sharma. - Published by Packt Publishing Ltd. 2019. 502 p. ISBN 978-1-78899-624-2

5. Bhaumik Vaidya. Hands-On GPU-Accelerated Computer Vision with OpenCV and CUDA

- Published by Packt Publishing Ltd. 2018. 380 p. ISBN: 978-1-78934-829-3

Додаткова література:

1. CUDA C++ Programming Guide https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

2. Jason Sanders. CUDA by Example: An Introduction to General-Purpose GPU Programming / Jason Sanders, Edward Kandrot. Published by Addison-Wesley Professional; 1st edition, 2010, 320p. ISBN: 978-0131387683

Зміст

Навчальне видання

Методичні вказівки

Програмування спеціалізованих обчислювальних систем

до виконання лабораторних робіт
з дисципліни «Програмування спеціалізованих
обчислювальних систем»
для студентів спеціальностей
«Автоматизація, комп'ютерно-інтегровані технології та робототехніка»,
«Електронні комунікації та радіотехніка»

Укладачі:
САЛЬНИКОВ Дмитро Валентинович
ЄВСЕЄНКО Олег Миколайович
ЗУЄВ Андрій Олександрович

Відповідальний за випуск Зуєв А.О.

Роботу до видання рекомендував Пугановський О.В.

В авторській редакції

План 2024 р., поз 536

Підп. до друку Формат 60x84 1/16.
Папір офсет. Друк ризографічний. Ум. друк. арк. 0,5.
Обл.вид. арк. Наклад 50 прим. Замовлення №

Видавничий центр НТУ «ХПІ», вул. Кирпичова, 2, м. Харків, 61002 Свідоцтво суб'єкта видавничої справи ДК № 5478 від 21.08.2017Друкарня НТУ «ХПІ», 61002, Харків, вулиця Кирпичова, 2