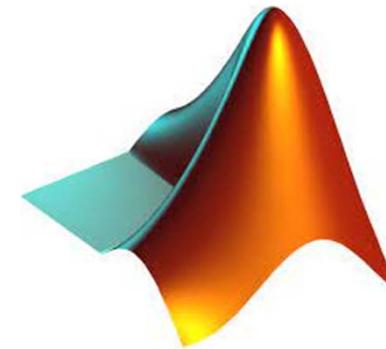


Комп'ютерне моделювання задач прикладної математики

Паралельні обчислення в MatLab, Python, C++.

Лекція 5

Паралельні обчислення в MatLab



Паралельне обчислення в циклі **for** (**parfor**)

Розберемо чим відрізняється **parfor** від **for**.

Якщо в тому, як працює цикл для проблем не виникає, то з **parfor** є питання.

Наприклад, події, на відміну стандартного циклу **for**, в **parfor** ітерації циклу можуть виконуватися не послідовно. Наприклад, може бути ситуація, де тіло циклу з індексом $i = 205$ може йти раніше, ніж $i = 160$. Це виникає через різницю продуктивності обчислювальних потоків. Через це ж, оскільки завдання виконуються в різних пулах, неможливо скористатися значеннями та результатами обчислення з попередніх ітерацій.

Паралельні обчислення в MatLab

Паралельне обчислення в циклі for (parfor)

```
n = 15;  
tic;  
for i = 1:500  
a = fibonacci(n);  
end  
time = toc  
Час виконання 7.67 секунд
```

```
tic;  
for i = 1:10  
a = rand(3000, 3000);  
a = tf(a);  
end  
time = toc  
Час виконання 56.3 секунд
```

```
n = 15;  
tic;  
parfor i = 1:500  
a = fibonacci(n);  
end  
time = toc  
Час виконання 5.06 секунд
```

```
tic;  
parfor i = 1:10  
a = rand(3000, 3000);  
a = tf(a);  
end  
time = toc  
Час виконання 34 секунди
```

Паралельні обчислення в Python

Різниця між потоками та процесами.

- Потік **threading** - це незалежна послідовність виконання якихось обчислень. Потік **thread** ділить виділену пам'ять ядра процесора, а також його процесорний час з усіма іншими потоками, які створюються програмою в рамках одного ядра процесора. Програми на мові **Python** мають один основний потік. Можна створити їх більше і дозволити **Python** перемикатися між ними. Це перемикання відбувається дуже швидко і здається, що вони працюють паралельно.
- Поняття процесу в **multiprocessing** - так само незалежну послідовність виконання обчислень. На відміну від потоків **threading**, процес має власне ядро і, отже, виділену йому пам'ять, яке не використовується спільно з іншими процесами. Процес може клонувати себе, створюючи два або більше екземплярів в одному ядрі

Python

Алгоритм планирования доступа потоков к общим данным.

- потоки используют одну и ту же выделенную память. Когда несколько потоков работают одновременно, то нельзя угадать порядок, в котором потоки будут обращаться к общим данным. Результат доступа к совместно используемым данным зависит от **алгоритма планирования**, который решает, какой поток и когда запускать. Если такого алгоритма нет, то конечные данные могут быть не такими как ожидаешь.
 - Например, есть общая переменная $a = 2$. Теперь предположим, что есть два потока, `thread_one` и `thread_two`. Они выполняют следующие операции:

```
a = 2  
# функция 1 потока  
def thread_one():  
    global a  
    a = a + 2  
  
# функция 2 потока  
def thread_two():  
    global a  
    a = a * 3
```

Доступ к общей переменной a:

- 1 - `thread_one`, 2 - `thread_two`:
 $2 + 2 = 4$;
 $4 * 3 = \mathbf{12}$.
- 1 - `thread_two`, 2 - `thread_one`:
 $2 * 3 = 6$;
 $6 + 2 = \mathbf{8}$.
- одновременный
 $2 + 2 = \mathbf{4}$; или $2 * 3 = \mathbf{6}$;

У [Python](#) есть одна особенность, которая усложняет параллельное выполнение кода. Она называется [GIL](#), сокращенно от [Global Interpreter Lock](#). [GIL](#) гарантирует, что в любой момент времени работает только один поток. Из этого следует, что с потоками невозможно использовать несколько ядер процессора.

[GIL](#) был введен в [Python](#) потому, что управление памятью [CPython](#) не является потокобезопасным. Имея такую блокировку [Python](#) может быть уверен, что никогда не будет **условий гонки**.

Исследование разных подходов

Определим функцию `heavy()`, которую будем использовать для сравнения различных вариантов вычислений - вложенный цикл, который выполняет возведение в степень.

```
def heavy(n):
    for x in range(1, n):
        for y in range(1, n):
            x**y
```

Это функция связана со скоростью ядра процессора производить математические вычисления. Если понаблюдать за операционной системой во время выполнения функции, то можно увидеть загрузку ЦП близкую к 100%.

Будем запускать эту функцию по-разному, тем самым исследуя различия между обычной однопоточной программой Python, многопоточностью и многопроцессорностью.

- **Однопоточный режим работы**

Каждая программа `Python` имеет по крайней мере один основной поток. Ниже представлен пример кода для запуска функции `heavy()` в одном основном потоке одного ядра процессора, который производит все операции последовательно и будет служить эталоном с точки зрения скорости выполнения:

```
import time
def sequential(n):
    for i in range(n):
        heavy(500, i)
    print(f"{n} циклов вычислений закончены")

if __name__ == "__main__":
    start = time.time()
    sequential(80)
    end = time.time()
    print("Общее время работы: ", end - start)
```

Вывод

```
# 80 циклов вычислений закончены
# Общее время работы: 23.573118925094604
```

Использование потоков `threading`.

Использование нескольких потоков для выполнения функции `heavy()`: 80 циклов вычислений путем разделения вычислений на 4 потока в каждом из которых 20 циклов:

```
def heavy(n, i, thread):
    for x in range(1, n):
        for y in range(1, n):
            x**y
```

```
def sequential(calc, thread):
    print(f"Запускаем поток № {thread}")
    for i in range(calc):
        heavy(500, i, thread)
    print(f"{calc} циклов вычислений закончены. Поток № {thread}")
```

```
def threaded(threads, calc):
    # threads - количество потоков
    # calc - количество операций на поток
    threads = []
    # делим вычисления на 4 потока
    for thead in range(threads):
        t = threading.Thread(target=sequential, args=(calc, thead))
        threads.append(t)
        t.start()
    # Подождем, пока все потоки завершат свою работу
    for t in threads: t.join()
```

```
import time
import threading
if __name__ == "__main__":
    start = time.time()
    threaded(4, 20)
    end = time.time()
    print("Время: ", end - start)
```

Вывод
Время: 43.3375225067138

Однопоточный режим работы, оказался почти в 2 раза быстрее, потому что один поток не имеет **накладных расходов на создание потоков** (в нашем случае создается 4 потока) и переключение между ними.

Использование потоков threading.

```
def heavy():
    # имитации операций
    time.sleep(2)
```

```
def sequential(calc, thread):
    print(f"Запускаем поток № {thread}")
    for i in range(calc):
        heavy(500, i, thread)
    print(f"{calc} циклов вычислений закончены. Поток № {thread}")
```

```
def threaded(theads):
    threads = []
    # делим операции на 80 потоков
    for thead in range(theads):
        t = threading.Thread(target=heavy)
        threads.append(t)
        t.start()
    # Подождем, пока все потоки завершат свою работу
    for t in threads: t.join()
    print(f"{theads} циклов имитации операций закончены")
```

Даже если воображаемый ввод-вывод делиться на 80 потоков и все они будут спать в течение двух секунд, то код все равно завершиться чуть более чем за две секунды, т. к. многопоточной программе нужно время на планирование и запуск потоков.

```
import time
import threading
if __name__ == "__main__":
    start = time.time()
    threaded(4, 20)
    end = time.time()
    print("Время: ", end - start)
```

Вывод
80 циклов имитации операций
закончены
Время: 2.00872588157653

Использование потоков `threading`.

Если бы у Python не было GIL, то вычисления функции `heavy()` происходили быстрее, а общее время выполнения программы стремилось к времени выполнения однопоточной программы. Причина, по которой многопоточный режим в данном примере не будет работать быстрее однопоточного - это вычисления, связанные с процессором и заключаются в GIL!

Если бы функция `heavy()` имела много блокирующих операций, таких как сетевые вызовы или операции с файловой системой, то применение многопоточного режима работы было бы оправдано и дало огромное увеличение скорости!

!!! Каждый процессор поддерживает определенное количество потоков на ядро, заложенное производителем, при которых он работает оптимально быстро. Нельзя создавать безгранично много потоков. При увеличении числа потоков на величину, большую, чем заложил производитель, программа будет выполняться дольше или вообще поведет себя непредсказуемым образом (вплоть до зависания).

Далее рассмотрим параллельную обработку с использованием модуля `multiprocessing`. Модуль `multiprocessing` во многом повторяет API модуля `threading`, по этому изменения в коде будут незначительны.

Использование многопроцессорной обработки multiprocessing.

Для того, что бы произвести 80 циклов вычислений функции `heavy()`, узнаем сколько процессор имеет ядер, а потом поделим циклы вычислений на количество ядер.

```
def heavy(n, i, proc):
    for x in range(1, n):
        for y in range(1, n):
            x**y
    print(f"Цикл № {i} ядро {proc}")
```

```
def sequential(calc, proc):
    print(f"Запускаем поток № {proc}")
    for i in range(calc):
        heavy(500, i, proc)
    print(f"{calc} циклов закончены. Проц № {thead}")
```

```
def processesed(procs, calc):
    # procs - количество ядер; calc - количество операций на ядро
    processes = []
    # делим вычисления на количество ядер
    for proc in range(procs):
        p = multiprocessing.Process(target=sequential, args=(calc, proc))
        processes.append(p)
        p.start()
    for p in processes: p.join() # Ждем, пока все ядра завершат свою работу
```

```
import time
import multiprocessing
if __name__ == "__main__":
    start = time.time()
    # узнаем количество ядер у процессора
    n_proc = multiprocessing.cpu_count()
    # вычисляем сколько циклов вычислений будет приходится # на 1 ядро, что бы в сумме получилось 80 или чуть больше
    calc = 80 // n_proc + 1
    processesed(n_proc, calc)
    end = time.time()
    print(f"Всего {n_proc} ядер в процессоре")
    print(f"На каждом ядре {calc} циклов")
    print(f"Итого {n_proc*calc} циклов за: ", end - start)
```

Использование многопроцессорной обработки `multiprocessing`.

Вывод

```
# ...
# ...
# ...
# Всего 6 ядер в процессоре
# На каждом ядре 14 циклов
# Итого 84 циклов вычислений за: 5.0251686573028564
```

Код выполнился почти в 5 раз быстрее чем линейный (последовательный). Это прекрасно демонстрирует линейное увеличение скорости вычислений от количества ядер процессора.

Выводы:

- Использовать модуль `threading` для программ, связанных с вводом-выводом, что бы значительно повысить производительность.
- Использовать модуль `multiprocessing` для решения проблем, связанных с операциями ЦП. Этот модуль использует весь потенциал всех ядер в процессоре.

Параллелизм в C+11

Linux и потоки

- Linux поддерживает многозадачность и многопоточность, т.е. в системе одновременно может работать несколько задач (процессов), и каждая из задач может выполняться в несколько потоков.
- Поток выполнения – элемент кода программы, выполняемый последовательно.
- Большинство приложений – однопоточные программы.
- Многопоточная программа в один момент времени может выполняться в нескольких отдельных потоках. В случае, если задача выполняется на многопроцессорной машине, то все ее потоки могут выполняться одновременно, повышая таким образом производительность выполнения задачи.
- Производительность многопоточного приложения можно улучшить даже на однопроцессорной системе. Например, если один из потоков приложения блокируется каким-то системным вызовом или ожидает поступления данных, в это время выполняется другой поток.

Стандарт C+11

- Одно из самых больших изменений в языке – поддержка многопоточности.
- До появления стандарта C+11 многопоточные программы можно было создавать при помощи средств ОС (pthreads на Unix-системах) или при помощи библиотек OpenMP, MPI.

Создание потока в C++11

Рассмотрим программу HelloWorld с потоками.

```
1 #include <iostream>
2 #include <thread>
3
4 //This function will be called from a thread
5
6 void call_from_thread() {
7     std::cout << "Hello, World" << std::endl;
8 }
9
10 int main() {
11     //Launch a thread
12     std::thread t1(call_from_thread);
13
14     //Join the thread with the main thread
15     t1.join();
16
17     return 0;
18 }
```

В новом стандарте C++11 многопоточность осуществлена в классе `thread`, который определен в файле `thread.h`.

Для того чтобы создать новый поток нужно создать объект класса `thread` и инициализировать, передав в конструктор имя функции, которая должна выполняться в потоке.

Метод `join` синхронизирует потоки и возвращает выполнение программе, когда поток заканчивается, после чего объект класса `thread` можно безопасно уничтожить.

Создание потока в C++11

```
#include <iostream>
#include <thread>
```

файл thread.h.

```
//This function will be called from a thread
```

```
void call_from_thread() {
    std::cout << "Hello, World" << std::endl;
}
```

```
int main() {
    //Launch a thread
    std::thread t1(call_from_thread);
```

```
    //Join the thread with the main thread
    t1.join();
```

```
    return 0;
}
```

- Компиляция при помощи g++
`g++ -std=c++11 -fmain.cpp`
- Функция “call_from_thread” выполняет какую-либо работу независимо от функции `main`.
- Функция `main` создает поток, и ждет, пока он завершится в точке `t1.join()`.
- Если не подождать дочерний поток, возможна ситуация, когда функция `main` завершится первой, и вся программа закончит свою работу, удалив запущенный поток вне зависимости от того завершилась ли функция “call_from_thread”, или нет.

Несколько потоков в C++11

```
1 ...  
2 static const int num_threads = 10;  
3 ...  
4 int main() {  
5     std::thread t[num_threads];  
6  
7     //Launch a group of threads  
8     for (int i = 0; i < num_threads; ++i) {  
9         t[i] = std::thread(call_from_thread);  
10    }  
11  
12     std::cout << "Launched from the main\n";  
13  
14     //Join the threads with the main thread  
15     for (int i = 0; i < num_threads; ++i) {  
16         t[i].join();  
17     }  
18  
19     return 0;  
20 }
```

Конкуренция
в гонке потоков

Обычно необходимо запустить несколько потоков, чтобы сделать какую-то работу параллельно.

Для этого нужно создать массив потоков.

В функции `main` создается массив из 10 потоков и функция ожидает завершения их работы.

!!! функция `main` — тоже поток, который обычно называют главным потоком, так что этот код запускает не 10, а 11 потоков.

```
Hello World  
Hello WorldHello World  
Hello World  
Hello World  
Hello World  
  
Hello World  
Hello World  
Hello World  
Launched from the main  
  
Hello World
```

Вывод в параллельном коде

```
3  using namespace std;
4  static const int num_threads = 10;
5  void call_from_thread() {
6      cout << "Hello, World" << endl;
7  }
8  int main() {
9      thread t[num_threads];
10     for (int i = 0; i < num_threads; ++i) {
11         t[i] = std::thread(call_from_thread);
12     }
13     cout << "Launched from the main\n";
14     for (int i = 0; i < num_threads; ++i) {
15         t[i].join();
16     }
17     return 0;
18 }
```

Потоковая функция с параметрами

C++ позволяет создать потоковую функцию с любым количеством параметров.

Добавим целочисленный параметр (int) в потоковую функцию.

```
3  using namespace std;
4  static const int num_threads = 10;
5  void call_from_thread(int tt) {
6      for (int i = 0; i < num_threads; ++i)
7          cout << tt ;
8          cout << endl;
9  }
10 int main() {
11     cout << " Main thread " << endl;
12     thread t[num_threads];
13     for (int i = 0; i < num_threads; ++i) {
14         t[i] = std::thread(call_from_thread, i);
15         t[i].join();
16     }
17     return 0;
18 }
```

```
Main thread
0000000000
1111111111
2222222222
3333333333
4444444444
5555555555
6666666666
7777777777
8888888888
9999999999
```

```
using namespace std;
static const int num_threads = 10;
void call_from_thread(int tt) {
    for (int i = 0; i < num_threads; ++i)
        cout << tt ; cout << endl;
}
int main() {
    cout << " Main thread " << endl;
    thread t[num_threads];
    for (int i = 0; i < num_threads; ++i) {
        t[i] = std::thread(call_from_thread, i);
    }
    for (int i = 0; i < num_threads; ++i)
        t[i].join();
    return 0;
}
```

```
Main thread
1011111111
2222222222
0000000000
4444444444
3333333333
5555555555
6666666666
7777777777
8888888888
9999999999
```



Где реально
многопоточность?

Задача программиста – проследить, чтобы потоки не блокировали друг друга, пытаясь получить доступ к одним и тем же данным.

Потоковая функция с параметрами

```
1 #include <iostream>
2 #include <thread>
3
4 static const int num_threads = 10;
5
6 //This function will be called from a thread
7
8 void call_from_thread(int tid) {
9     std::cout << "Launched by thread " << tid << std::endl;
10 }
11
12 int main() {
13     std::thread t[num_threads];
14
15     //Launch a group of threads
16     for (int i = 0; i < num_threads; ++i) {
17         t[i] = std::thread(call_from_thread, i);
18     }
19
20     std::cout << "Launched from the main\n";
21
22     //Join the threads with the main thread
23     for (int i = 0; i < num_threads; ++i) {
24         t[i].join();
25     }
26
27     return 0;
28 }
```

Launched by Thread 0
Launched by Thread 2
Launched by Thread 3
Launched by Thread Launched by Thread 45
Launched by Thread 6
Launched by Thread 1
Launched by Thread 7
Launched by Thread Launched from the main

Launched by Thread 9
8

Видно, что вывод из разных потоков смешивается –потому что все 11 потоков пытаются получить доступ к одному ресурсу – `stdout` потоку вывода.

Вспомни C++!

Передача аргументов по значению и по ссылке

```
#define N 5
#define M 100
int a[N][M];
void max_thread(int i) {
    int max = INT_MIN;
    for (int j = 0; j < M; j++) {
        if (a[i][j] > max) {
            max = a[i][j];
        }
    }
    std::cout << "Max in " << i << " is: " << max << std::endl;
}
int main() { ...
// Вызов потоков в цикле ...
    t[i] = std::thread(max_thread, i);
}
```

Параметр

```
void max_thread(int *a) {
    int max = INT_MIN;
    for (int j = 0; j < M; j++) {
        if (a[j] > max) {
            max = a[j];
        }
    }
    std::cout << "Max is: " << max << std::endl;
}
int main() {
    int a[N][M];
    ... // Ввод значений
    std::thread t[N];
    for (int i = 0; i < N; i++) {
        t[i] = std::thread(max_thread, a[i]);
    }
    for (int i = 0; i < N; i++) {
        t[i].join();
    }
    return 0;
}
```

Вызов

Передача параметров в потоке

По значению

```
10 void threadFunction(int a)
11 {
12     a++;
13 }
14
15 int main()
16 {
17     int a = 1;
18     std::thread thr(threadFunction, a);
19     thr.join();
20     std::cout << a << std::endl;
21     return 0;
22 }
```

Результат 1

По ссылке

```
10 void threadFunction(int &a)
11 {
12     a++;
13 }
14
15 int main()
16 {
17     int a = 1;
18     std::thread thr(threadFunction, std::ref(a));
19     thr.join();
20     std::cout << a << std::endl;
21     return 0;
22 }
```

Результат 2

OpenMP

OpenMultiProcessing



OpenMP (Open Multi-Processing) – открытый стандарт для распараллеливания программ на языках Си, Си++ и Фортран. Дает описание совокупности директив компилятора, библиотечных процедур и переменных окружения, которые предназначены для программирования многопоточных приложений на многопроцессорных системах с общей памятью.

Интерфейс OpenMP задуман как стандарт для программирования на масштабируемых [SMP-системах](#) (SSMP, ccNUMA, etc.) в модели общей памяти (shared memory model).

В стандарт OpenMP входят спецификации набора директив компилятора, процедур и переменных среды.

- Открытый стандарт для распараллеливания программ на языках С, С++ и Фортран
- Очень важен в области High Performance Computing (HPC).
- Требует поддержки со стороны компилятора.
- Одна программа для последовательного компьютера (отладка) и параллельного.
- Инкрементальное распараллеливание (циклы).

MPI (Message Passing Interface)

Системы с распределенной памятью.

Message Passing Interface (MPI, интерфейс передачи сообщений) – программный интерфейс (API) для передачи информации, который позволяет обмениваться сообщениями между процессами, выполняющими одну задачу.



- MPI – наиболее распространённый стандарт интерфейса обмена данными в параллельном программировании, существуют его реализации для большого числа компьютерных платформ. Используется при разработке программ для кластеров и суперкомпьютеров.
- Стандартизацией MPI занимается MPI Forum. В стандарте MPI описан интерфейс передачи сообщений, который должен поддерживаться как на платформе, так и в приложениях пользователя. В настоящее время существует большое количество бесплатных и коммерческих реализаций MPI. Существуют реализации для языков Фортран 77/90, Java, Си и Си++.
- MPI ориентирован на системы с распределенной памятью, когда затраты на передачу данных велики.
- Технологии OpenMP и MPI могут использоваться совместно, чтобы оптимально использовать в кластере многоядерные системы.



CUDA

(Compute Unified Device Architecture)

CUDA

(Compute Unified Device Architecture)



- CUDA Программирование массивно параллельных систем требует специальных систем/языков. CUDA - система (библиотеки и расширенный С) для программирования GPU
- CUDA –архитектура параллельных вычислений от NVIDIA, позволяющая существенно увеличить вычислительную производительность благодаря использованию GPU (графических процессоров).
- CUDA SDK позволяет программистам реализовывать на специальном упрощённом диалекте языка программирования С алгоритмы, выполнимые на графических процессорах NVIDIA, и включать специальные функции в текст программы на С.
- Архитектура CUDA даёт разработчику возможность по своему усмотрению организовывать доступ к набору инструкций графического ускорителя и управлять его памятью.

Основные понятия CUDA

- **Хост (host)** – узел с CPU и его память
- **Устройство (device)** – графический процессор и его память
- **Ядро (kernel)** – это фрагмент программы, предназначенный для выполнения на GPU
- Пользователь самостоятельно запускает с CPU ядра на GPU
- Перед выполнением ядра пользователь копирует данные из памяти хоста в память GPU
- После выполнения ядра пользователь копирует данные из памяти GPU в память хоста



Пример программы CUDA

Hello world

```
#include <stdio.h>
__global__ void kernel() {
}
int main() {
    kernel<<<1,1>>>0;
    printf("Hello, World!\n");
    return 0;
}
```

В одном файле находится код предназначенный для исполнения как CPU (host), так и GPU (device).

Функция `kernel` передается компилятору, обрабатывающему код для устройства.

`<<<x,y>>>` – вызов кода (с созданием x параллельных блоков и у потоков), выполняемого GPU.

О ключевых словах

Ключевое слово `_device_` означает, что код исполняется GPU. Такие функции можно вызывать из других функций с ключевыми словами `_device_` и `_global_`.

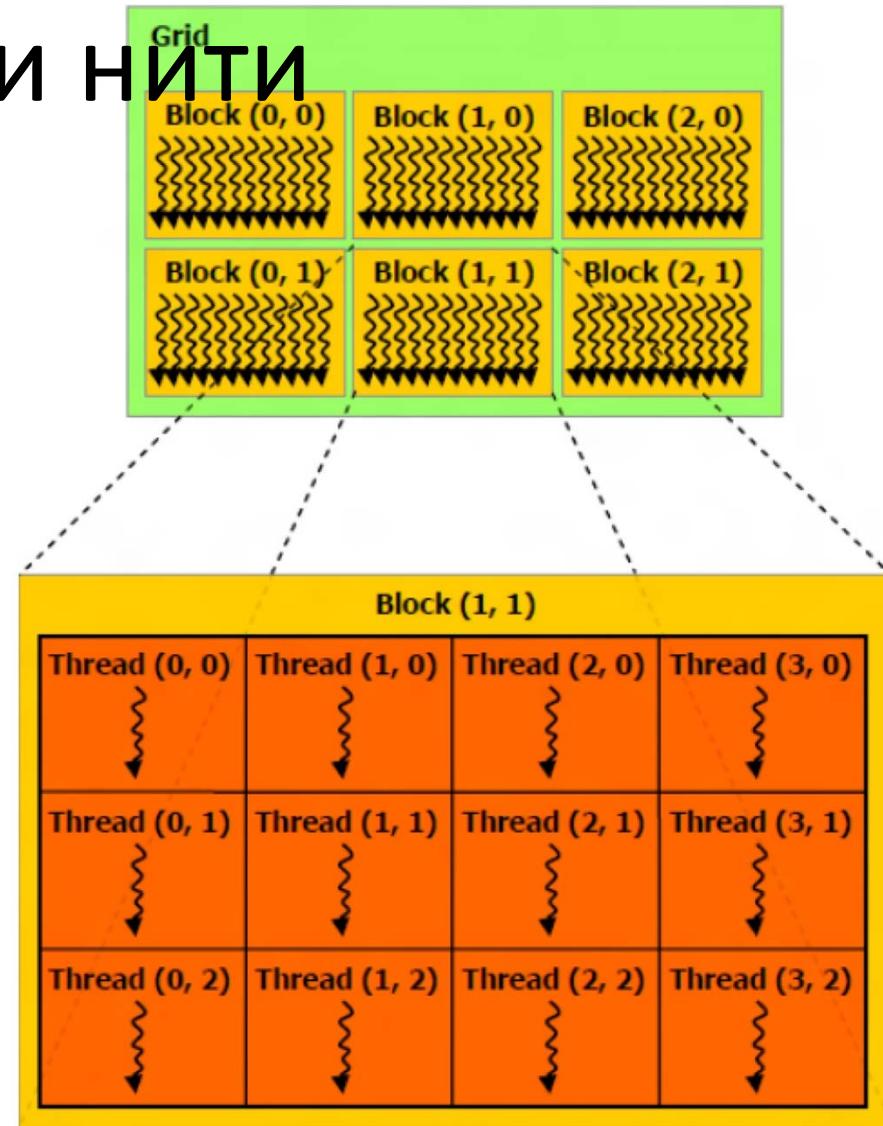
Функции `_global_` вызываются только из кода CPU.



БЛОКИ И НИТИ

При вызове `__global__` функции в конструкции
`<<< >>>` в общем случае указываются два трехмерных вектора типа `dim3`, обозначающие размерность решетки блоков и размерности блока.

Пример для решетки блоков размерностью $(2, 3, 1)$ и размерности блока размерностью $(3, 4, 1)$.

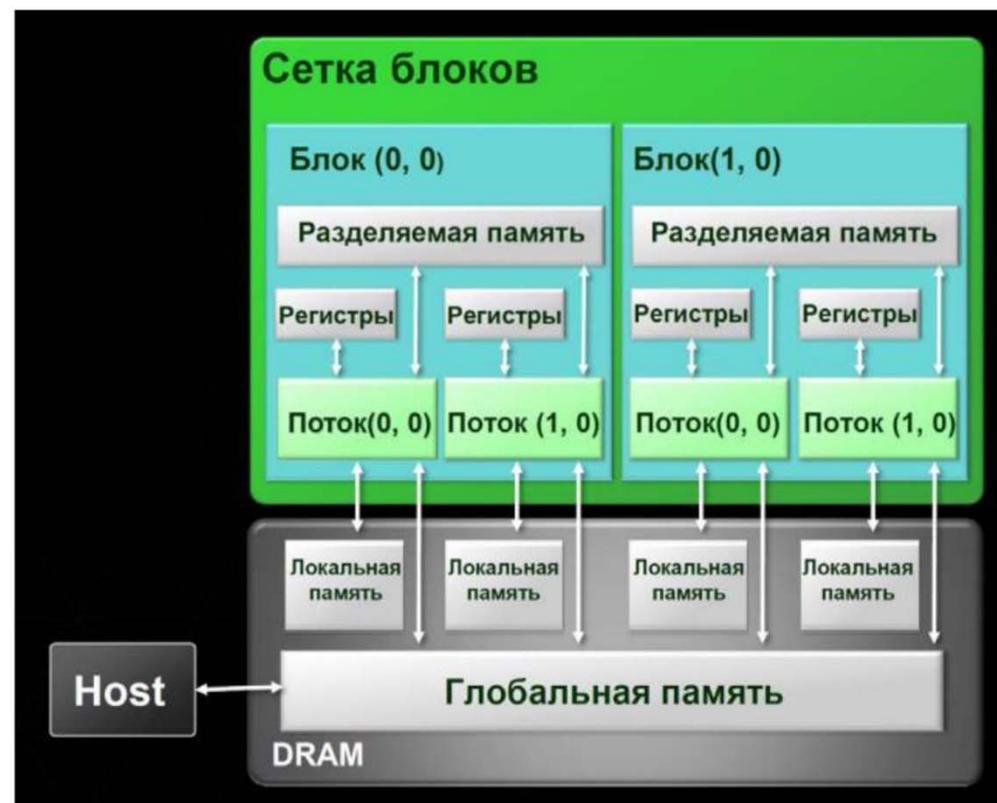


О размерности GPU

Каждая видеокарта имеет свои ограничения для максимальных размерностей решеток блока и размерности блока (512 x 512 x 64 для размерности блока и 65536 x 65536 x 1 для решетки блока).

Существует предел количества потоков на блок, поскольку все потоки блока располагаются на одном процессорном ядре GPU и должны разделять ограниченные ресурсы памяти этого ядра.

Размерность блока можно узнать с помощью встроенной переменной `blockDim`, размерность сетки блоков – с помощью `gridDim`.



Получение информации об устройствах

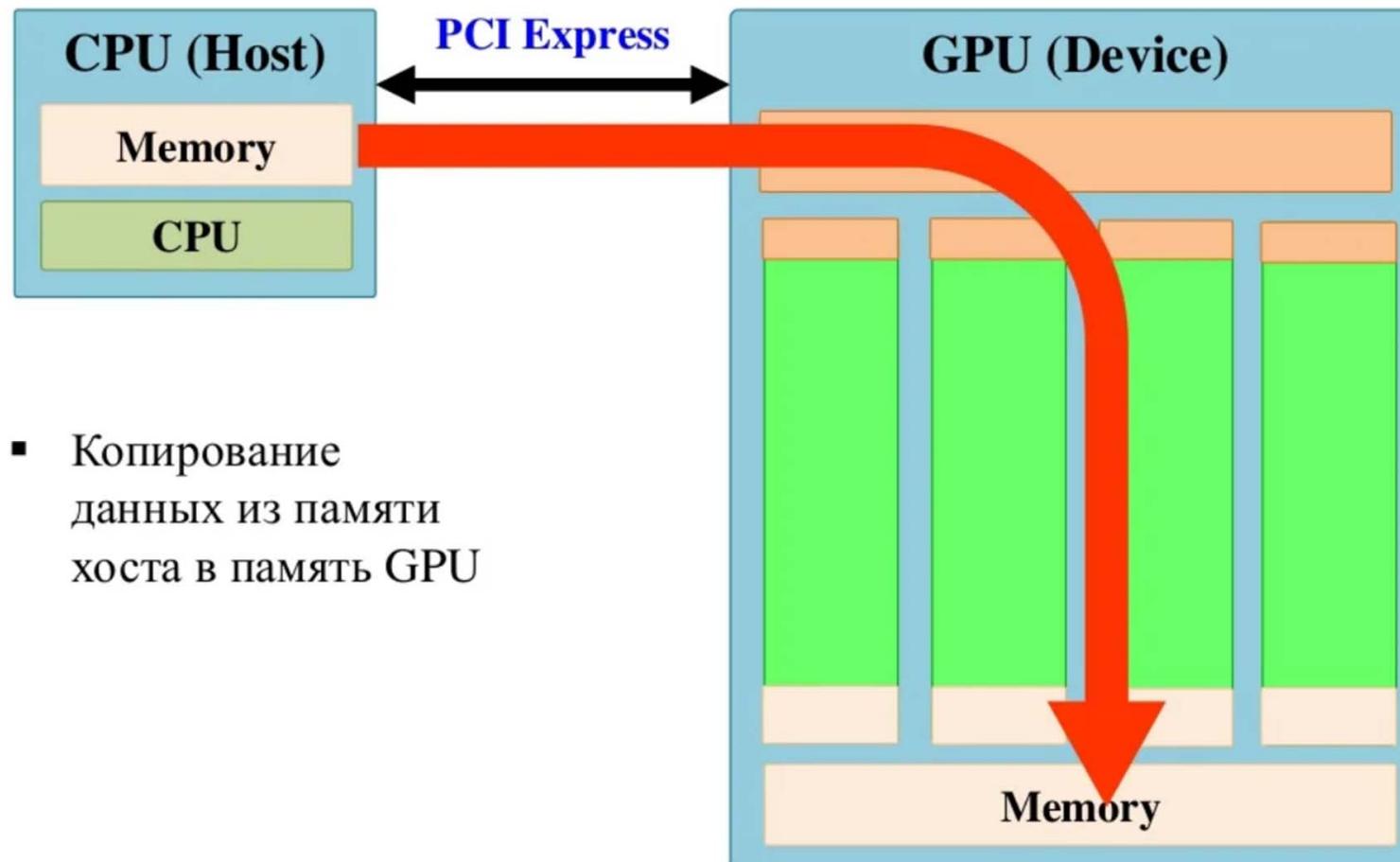
```
Int main()
{
    cudaDeviceProp prop;
    int count;
    cudaGetDeviceCount(&count);
    for(int i= 0; i< count; i++)
    {
        cudaGetDeviceProperties(&prop, i);
        // Сделать что-то со свойствами устройства
    }
}
```

Структура `cudaDeviceProp` содержит поля с информацией:

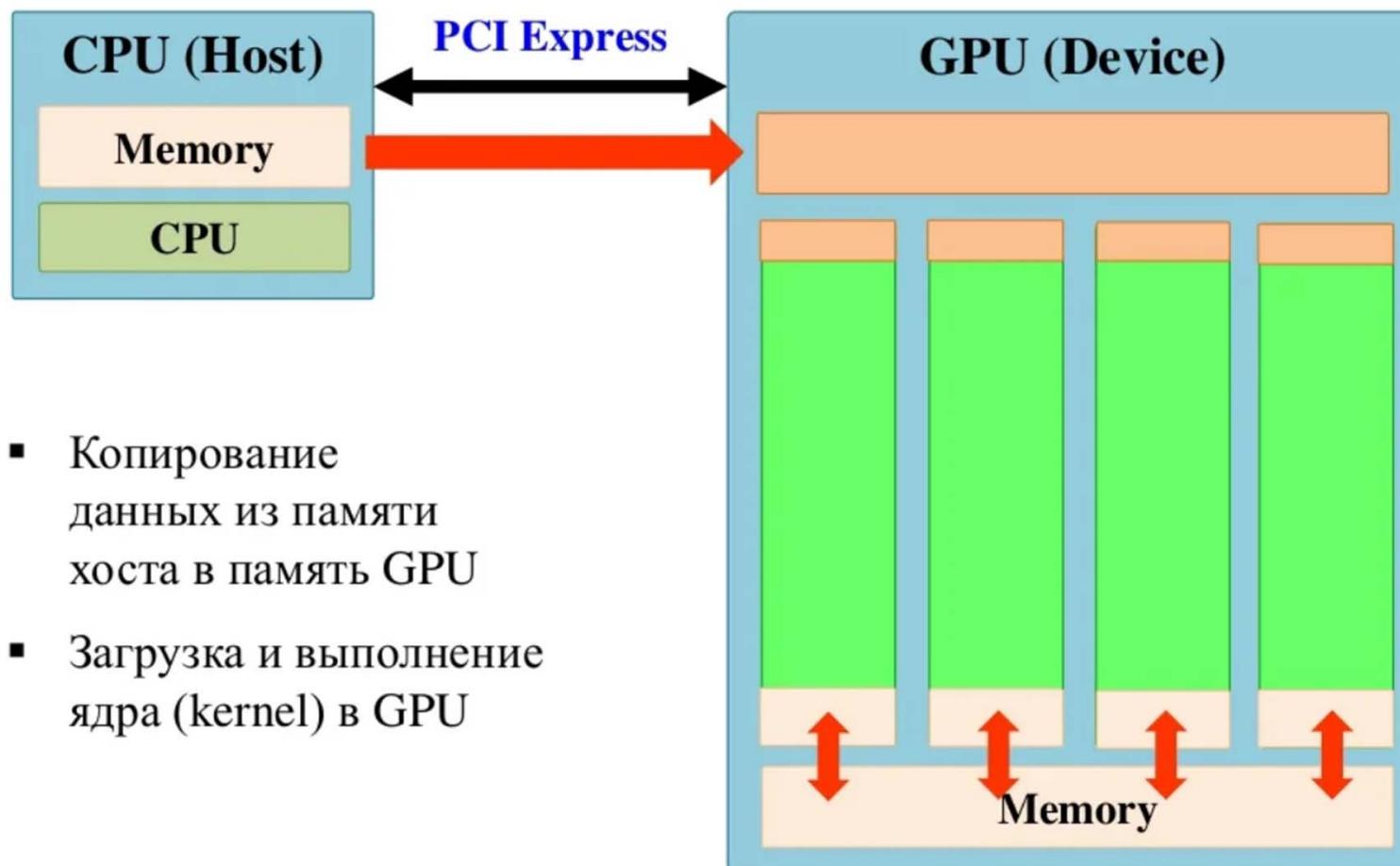
- имя GPU,
- объём памяти,
- максимальное количество нитей в блоках и т.д.



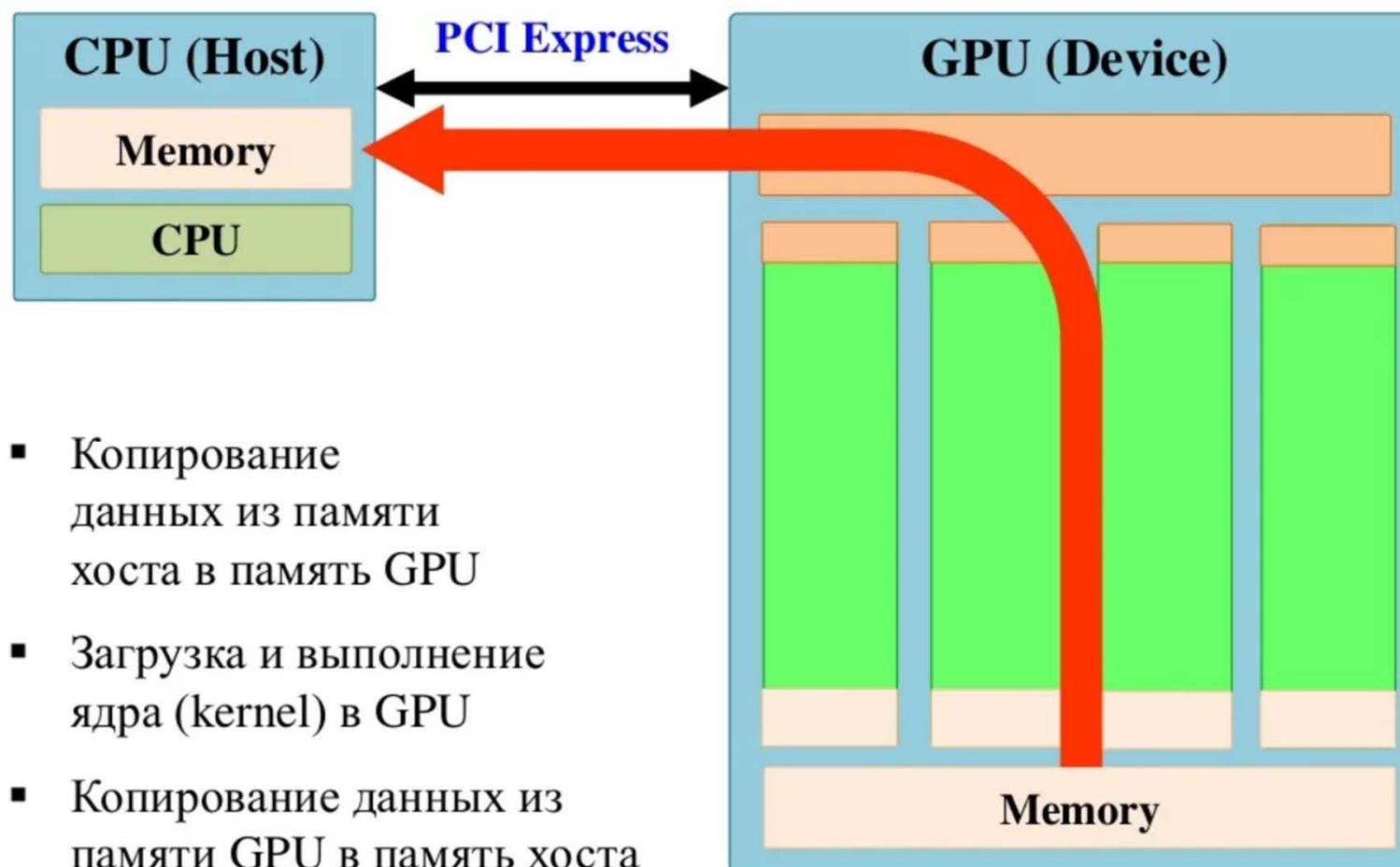
Выполнение программы CUDA



Выполнение программы CUDA



Выполнение программы CUDA



Правила работы с указателями на память

- Разрешается передавать указатели на память, выделенную cudaMalloc() функциям, исполняемым GPU.
- Разрешается использовать указатели на память, выделенную cudaMalloc() для чтения и записи в эту память в коде, исполняемом GPU.
- Разрешается передавать указатели на память, выделенную cudaMalloc() функциям, исполняемым CPU.
- Не разрешается использовать указатели на память, выделенную cudaMalloc(), для чтения и записи в эту память в коде, который исполняется CPU.
- Поэтому область памяти GPU копируется функцией cudaMemcpy() в память CPU. Последний параметр – константа, описывающая направление копирования памяти:
 - cudaMemcpyDeviceToHost
 - cudaMemcpyHostToDevice
 - cudaMemcpyDeviceToDevice
- Функции работы с памятью GPU возвращают коды ошибок.



Пример CUDA

сложение векторов

```
__global__ void vecadd_gpu(float *a, float *b, float *c)
{
    // Каждый поток обрабатывает один элемент
    int i = threadIdx.x;
    c[i] = a[i] + b[i];
}
```

- Запускается один блок из n потоков ($n \leq 1024$)
`vecadd_gpu<<<1, n>>>(deva, devb, devc);`
- Каждый поток вычисляет один элемент массива c

```
int threadsPerBlock = 256; /* Device specific */
int blocksPerGrid = (n + threadsPerBlock - 1) /
                     threadsPerBlock;
vecadd_gpu<<<blocksPerGrid, threadsPerBlock>>>(deva,
    devb, devc, n);
```

- Будет запущена группа блоков, в каждом блоке по фиксированному количеству потоков
- Потоков может быть больше чем элементов в массиве

```
int main()
{
    int i, n = 100;
    float *a, *b, *c;
    float *deva, *devb, *devc;

    a = (float *)malloc(sizeof(float) * n);
    b = (float *)malloc(sizeof(float) * n);
    c = (float *)malloc(sizeof(float) * n);
    for (i = 0; i < n; i++) {
        a[i] = 2.0;
        b[i] = 4.0;
    }
    // Выделяем память на GPU
    cudaMalloc((void **)&deva, sizeof(float) * n);
    cudaMalloc((void **)&devb, sizeof(float) * n);
    cudaMalloc((void **)&devc, sizeof(float) * n);

    // Копируем из памяти узла в память GPU
    cudaMemcpy(deva, a, sizeof(float) * n,
               cudaMemcpyHostToDevice);
    cudaMemcpy(devb, b, sizeof(float) * n,
               cudaMemcpyHostToDevice);

    vecadd_gpu<<<1, n>>>(deva, devb, devc);

    cudaMemcpy(c, devc, sizeof(float) * n,
               cudaMemcpyDeviceToHost);
    cudaFree(deva); cudaFree(devb); cudaFree(devc);
    free(a); free(b); free(c);

    return 0;
}
```



Пример CUDA

CUDA C

NVIDIA.

Standard C Code

```
void saxpy_serial(int n,
                  float a,
                  float *x,
                  float *y)
{
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}

// Perform SAXPY on 1M elements
saxpy_serial(4096*256, 2.0, x, y);
```

Parallel C Code

```
__global__
void saxpy_parallel(int n,
                     float a,
                     float *x,
                     float *y)
{
    int i = blockIdx.x*blockDim.x +
            threadIdx.x;
    if (i < n) y[i] = a*x[i] + y[i];
}

// Perform SAXPY on 1M elements
saxpy_parallel<<<4096, 256>>>(n, 2.0, x, y);
```

<http://developer.nvidia.com/cuda-toolkit>



Дякую за увагу