

С.Д. Погорелый, М.И. Трибрат, Ю.В. Бойко, Д.Б. Грязнов

Реализация алгоритма Флойда–Уоршалла для программно-аппаратной платформы *CUDA*

Проведен анализ методики реализации алгоритма Флойда–Уоршалла для программно-аппаратной платформы *CUDA*. Выполнено сравнение времён работы алгоритма на видеоадаптере и центральном процессоре компьютера. Выявлены возможные пути сокращения времени работы алгоритма на видеоадаптере.

An analysis of the methods of implementation of the Floyd–Warshall algorithm for the Software-Hardware platform *CUDA* is conducted. A comparison of the running time on the graphics card and a central processing unit is made. Possible ways to reduce the running time of the algorithm on the graphics card are exposed.

Проведено аналіз методів реалізації алгоритму Флойда–Уоршалла для програмно-апаратної платформи *CUDA*. Виконано порівняння часу роботи алгоритму на відеоадаптері і центральному процесорі комп'ютера. Виявлено можливі шляхи скорочення часу роботи алгоритму на відеоадаптері.

Введение. Актуальная задача компьютерных сетей – задача маршрутизации и выбора оптимальных маршрутов. Предлагаемый алгоритм был разработан в 1962 году Р. Флойдом и С. Уоршаллом и представляет собой динамический алгоритм для нахождения кратчайших расстояний между всеми парами вершин взвешенного ориентированного графа, эффективно работающий на плотных графах и временную полиномиальную сложность его $O(N^3)$, где N – количество вершин графа [1].

Актуальной задачей остается минимизация времени работы алгоритма, так как процесс маршрутизации запускается при любом изменении топологии сети и в течение суток может быть использован миллионы раз.

Цель исследования – реализация предлагаемого алгоритма с использованием новой программно-аппаратной платформы *CUDA* [2], сравнение времени работы реализации на видеоадаптере и центральном процессоре, выявление путей сокращения времени работы алгоритма на видеоадаптере и создание методик и рекомендаций по реализации алгоритмов для работы с графами. Метод исследования разработан и описан в [3].

Объектом исследования выступает не только методика реализации алгоритма на программно-аппаратной платформе *CUDA*, а и ее

обобщение для реализации подобных алгоритмов на видеоадаптерах.

Современные программно-аппаратные платформы позволяют значительно увеличить скорость работы алгоритма, используя в том числе и распараллеливание его работы [4].

Формирование параллельной версии алгоритма

Процедура, выполняющая последовательный алгоритм, выглядит так:

```
for (k = 0; k < N; k++)
  for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
      D[i][j] = min(D[i][j], D[i][k] + D[k][j]),
```

 (1)

где матрица $D[i][j]$ содержит веса кратчайших путей; i, j, k – параметры циклических процессов.

Очевидно, что время выполнения алгоритма равно $O(N^3)$, однако, теоретически его можно свести к $O(N)$, реализовав один из подходов к распараллеливанию.

Для корректной работы параллельной реализации алгоритма необходимо разделить вычисления на независимые одну от другой части, выделив информационные зависимости.

На k -й итерации элементы $D[i][k]$ и $D[k][j]$ матрицы $D[i][j]$ остаются неизменными. Допустим, в выражении (1) $i = k$, откуда следует:

$$D[k][j] = \min(D[k][j], D[k][k] +$$

$$+ D[k][j]), D[k][j] = D[k][j], \quad (2)$$

так как $D[k][k] = 0$.

Аналогично для $j = k$ в выражении (1) получим:

$$D[i][k] = \min(D[i][k], D[i][k] + D[k][k]), D[i][k] = D[i][k], \quad (3)$$

так как $D[k][k] = 0$.

Используя программно-аппаратную платформу *CUDA*, позволяющую запускать одновременно тысячи потоков для математических расчетов на видеоадаптере, можно реализовать изложенный подход к распараллеливанию работы алгоритма.

Теоретически можно сократить время работы алгоритма с $O(N^3)$ до $O(N)$, заменив циклы по i и j на значение двумерного индекса каждого потока (*threadIdx.x*, *threadIdx.y*) [5], оставив только итерации по индексу k . При этом каждый поток будет выполнять только одну операцию на итерации k :

$$D[i][j] = \min(D[i][j], D[i][k] + D[k][j]). \quad (4)$$

Увеличение размерности матрицы весов влечет за собой увеличение количества потоков: при размерности 1024×1024 число потоков будет 1048576 и т.д.

Экспериментальные результаты

Для экспериментального подтверждения предложенного подхода использовался один узел кластера [6] на основе процессора *Intel Core 2 Duo E6550* и видеоадаптера *NVIDIA GeForce 8600 GT* (частота работы *CPU* и *GPU* соответственно 2,33 GHz и 0,516 GHz). Алгоритм выполнялся сначала на *GPU*, затем, для сравнения, на *CPU*.

С помощью генератора случайных чисел была сгенерирована основная матрица весов, которая считывалась в глобальную память видеоадаптера. Тестируемая матрица получалась путем вырезания части соответствующей размерности из основной матрицы весов. Шаг вырезания тестируемой матрицы выбран равным 16, так как 16 потоков – это размер *half warp*.

Алгоритм реализован на использовании глобальной памяти и разделяемой памяти видеоадаптера. Глобальная память имеет наибольшие

размеры (для данного видеоадаптера 256 Мб), необходимые для одноразовой загрузки, сравнения и изменения весов графа на итерации k . Разделяемая память имеет значительно меньший объем (для данного видеоадаптера 16 Кб), однако скорость обмена информацией с *GPU* значительно выше, чем скорость обмена *GPU* и глобальной памяти.

Для синхронизации нитей в ядре программы графического процессора на итерации k использовалась явная синхронизация центральным процессором (*CPU explicit synchronization* [7]).

В ходе эксперимента обнаружена существенная особенность поведения стандартного *CUDA* таймера для измерений. При старте таймера измерения времени работы алгоритма на *GPU* (*CUDA timer GPU*) непосредственно перед запуском ядра программы (*kernel*) и остановки сразу после выполнения расчетов (т.е. без учета времени выделения памяти и копирования данных из оперативной памяти в глобальную память видеоадаптера и обратно), таймер выдавал некорректные результаты.

Некорректные результаты были выявлены при анализе экспериментальных данных с таймеров, измерявших время копирования матрицы смежности на видеоадаптер и с него. Таймеры измерений времени работы частей алгоритма (рис. 1) таковы:

- *CUDA Total timer GPU* – полное время работы алгоритма с учетом копирования матрицы смежности на видеоадаптер и с него;
- *CUDA timer to* – время копирования матрицы из оперативной памяти в глобальную память видеоадаптера;
- *CUDA timer GPU* – время выполнения алгоритма на *GPU*;
- *CUDA timer from* – время копирования результата из глобальной памяти видеоадаптера в оперативную память.

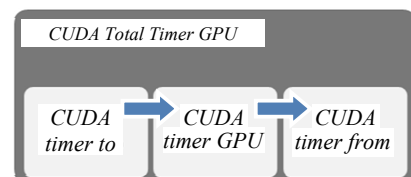


Рис. 1. Порядок запуска таймеров измерения времени работы частей параллельного алгоритма

Показатели времени копирования данных при учете пропускной способности шины *PCIe* ×16 [8] были необъяснимо велики, зависели от конфигурации потоков и менялись в зависимости от нагрузки на *GPU*, а также время выполнения алгоритма на *GPU* слабо зависело от размерности матрицы.

Такие результаты категорически нельзя использовать для дальнейшего анализа. Для измерения времени работы программы только на *GPU* (без накладных расходов на копирование, выделение памяти и др.) необходимо использовать таймер событий (*CUDA event timer*), который в этом случае даст достоверные результаты.

Для достижения целей использовались три типа таймеров для измерений (рис. 2):

- Таймер аппаратно-программной платформы (*CUDA timer GPU* и *CUDA Total timer GPU*);
- Таймер событий аппаратно-программной платформы (*CUDA event timer GPU*);
- Стандартный таймер *Windows* (рис. 2).

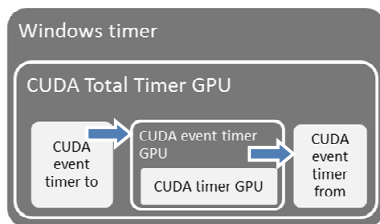


Рис. 2. Измененный порядок запуска таймеров измерения времени работы частей параллельного алгоритма

Показатели таймеров представлены на рис. 3, что и подтверждает корректность их работы, и, соответственно, полученных результатов.

Как видно из рисунка, все три типа таймеров показывают схожие результаты, однако при малых размерностях матрицы весов стандартный таймер *Windows* показывает увеличенные значения временных интервалов в сравнении с остальными таймерами. При увеличении размерности матрицы разницы в показании таймеров можно пренебречь.

Поскольку цель работы – сравнение времени выполнения алгоритма на *CPU* и *GPU*, то сравнивалось время выполнения алгоритма на *CPU* (t^{CPU}) и полное время работы на *GPU* (t^{GPU}) с учетом копирования данных из оперативной памяти в глобальную память видеоадаптера,

вычисления на графическом процессоре и копирования данных обратно в оперативную память узла кластера.

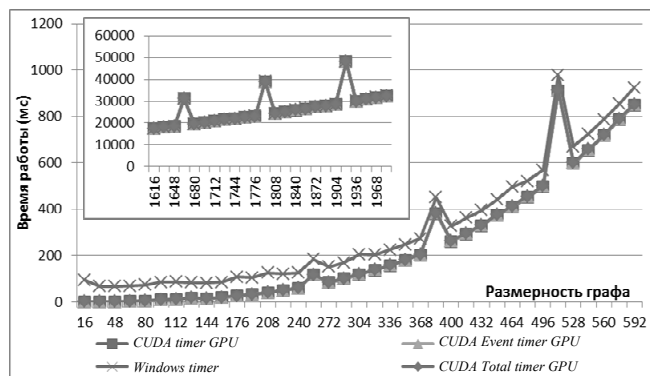


Рис. 3. Различия показаний разных типов таймеров на малых и больших (верхний левый угол) размерностях матрицы весов

Время считывания тестируемой матрицы весов с жесткого диска в оперативную память узла не учитывалось, так как оно одинаково как для центрального процессора, так и для видеоадаптера. Учитывая изложенное, были получены следующие результаты (рис. 4).

При размерности матрицы весов 208×208 и больше время выполнения алгоритма на видеоадаптере в 3,3 раза меньше, чем на центральном процессоре (рис. 5).

Анализ полученных экспериментальных результатов показывает, что ожидаемое время выполнения алгоритма $O(N)$, практической реализацией не достигнуто. Это объясняется тем, что каждый поток на итерации k обращается к глобальной памяти видеоадаптера, которая самая большая по объему, но заведомо наиболее медленная.

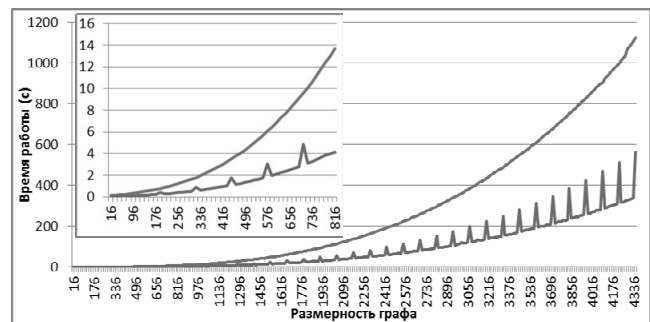


Рис. 4. Полное время работы параллельной реализации алгоритма на видеоадаптере (t^{GPU}) и время выполнения алгоритма на центральном процессоре (t^{CPU}) (верхний левый угол – первые 14 с работы алгоритма)



Рис. 5. Отношение времени работы (t^{CPU}/t^{GPU}) на центральном процессоре и параллельной реализации на видеоадаптере

Увеличение времени выполнения алгоритма на видеоадаптере с шагом увеличения размерности матрицы весов, равным 128, объясняется тем, что на конкретном видеоадаптере имеется четыре мультипроцессора, каждый из которых физически параллельно выполняет 32 потока (один *warp*). При ситуации, когда одновременно все потоки всех мультипроцессоров обращаются к глобальной памяти, происходит возрастание латентности и, соответственно, замедление доступа к данным в глобальной памяти (рис. 6). Подобные пики будут наблюдаться и на других видеоадаптерах, однако расстояние между ними будет обусловлено количеством мультипроцессоров конкретного графического процессора.



Рис. 6. Изменение отношения времени работы на центральном процессоре (t^{CPU}) и параллельной реализации алгоритма на видеоадаптере (t^{GPU})

Для сокращения времени выполнения алгоритма необходимо уменьшить количество обращений к глобальной памяти и, по возможности, максимально использовать разделяемую память. Использование разделяемой памяти без уменьшения обращений к глобальной памяти только замедлит выполнение алгоритма, так как данные будут по-прежнему копироваться каждым потоком из глобальной в разделяемую память, которая в данном случае используется как буфер (рис. 7) [9].

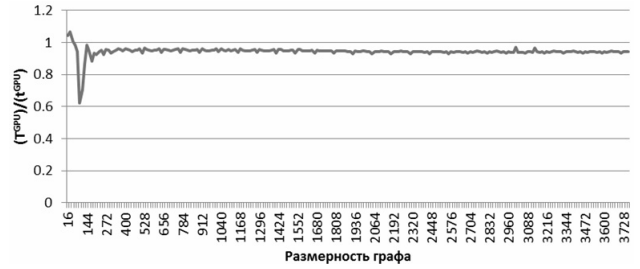


Рис. 7. Отношение времени выполнения алгоритма при использовании разделяемой памяти в качестве буфера (T^{GPU}) и глобальной памяти (t^{GPU})

К накладным расходам можно отнести время копирования матрицы весов из оперативной памяти в глобальную память видеоадаптера (рис. 8), а также время на явную синхронизацию центральным процессором.

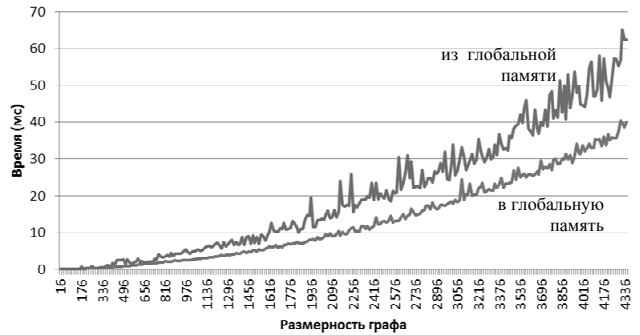


Рис. 8. Время копирования данных по шине *PCI-e* в глобальную память видеоадаптера и из нее

Закключение. В реализации алгоритма Флойда–Уоршалла для программно-аппаратной платформы *CUDA*, для сокращения времени выполнения алгоритма использован способ выделения информационных зависимостей, при котором теоритическое время выполнения алгоритма сократилось с $O(N^3)$ до $O(N)$ в результате создания числа потоков, равного числу элементов матрицы весов графа.

Получено ускорение работы параллельного алгоритма в 3,3 раза в сравнении с последовательным алгоритмом, выполненным на *CPU*.

При конфигурировании программной реализации таким образом, что все физически параллельно исполняемые потоки (*warp*, 32 потока) каждого мультипроцессора графического процессора одновременно начнут обращаться к глобальной памяти, время обращения к ней возрастет.

Окончание на стр. 72.