УДК 004.3

СТВОРЕННЯ І ДОСЛІДЖЕННЯ ПАРАЛЕЛЬНИХ СХЕМ АЛГОРИТМУ ДЖОНСОНА В ТЕХНОЛОГІЇ GPGPU

С.Д. Погорілий, М.С. Слинько

Запропоновано застосування алгоритму Джонсона для знаходження найкоротших шляхів між усіма парами вершин зваженого орієнтованого графа. Виконано його формалізацію у термінах модифікованих систем алгоритмічних алгебр Глушкова. Обгрунтовано доцільність використання технології GPGPU для пришвидшення роботи алгоритму. Отримано низку схем паралельної версії алгоритму, оптимізовану для використання в технології GPGPU. Запропоновано підходи до реалізації отриманих схем з використанням архітектури обчислень NVIDIA CUDA. Виконано експериментальне дослідження підвищення продуктивності при проведенні обчислень на відеоадаптері.
Ключові слова: NVidia CUDA, GPGPU, SSSP, APSP, Thrust, САА схема, алгоритм Джонсона.

Предложено применение алгоритма Джонсона для нахождения кратчайших путей между всеми парами вершин взвешенного ориентированного графа. Выполнена его формализация в терминах модифицированных систем алгоритмических алгебр Глушкова. Обоснована целесообразность использования технологии GPGPU для ускорения работы алгоритма. Получен ряд схем параллельной версии алгоритма, оптимизированной под использование в технологии GPGPU. Предложено подходы к реализации полученных схем с использованием архитектуры вычислений NVIDIA CUDA. Выполнено экспериментальное исследование повышения производительности при проведении вычислений на видеоадаптере.
Ключевые слова: NVidia CUDA, GPGPU, SSSP, APSP, Thrust, САА схема, алгоритм Джонсона.

Johnson's all pairs shortest path algorithm application in an edge weighted, directed graph is considered. Its formalization in terms of Glushkov's modified systems of algorithmic algebras was made. The expediency of using GPGPU technology to accelerate the algorithm is proved. A number of schemas of parallel algorithm optimized for using in GPGPU were obtained. Suggested approach to the implementation of the schemes obtained using computing architecture NVIDIA CUDA. An experimental study of improved performance by using GPU for computations was made.
Key words: NVidia CUDA, GPGPU, SSSP, APSP, Thrust, SAA scheme, Johnson's algorithm.

Вступ

Багато застосувань теорії графів існують у галузі мережевого аналізу. Однією із найважливіших і найактуальніших у сьогоденні областей застосування задачі пошуку найкоротших шляхів у ґрафах є маршрутизація пакетів у комп'ютерних мережах [1]. Теорія графів використовується в хімії та молекулярній біології для задач моделювання. Графи – зручна модель для розв'язання таких задач, як дизайн VLSI мікросхем, філогенетика, видобуток даних, біоінформатика, аналіз мереж тощо.

В роботі розглядається проблема знаходження найкоротших шляхів між усіма парами вершин зваженого орієнтованого графа (APSP), яка потребує великих обчислювальних потужностей. Наприклад, алгоритм Флойда – Уоршелла (1962) має складність, $O(|V^2|)$, де $|V|$ – кількість вершин у графі. Різні алгоритми пропонують різні підходи до вирішення задачі: наприклад, алгоритм Данцига полягає у послідовному обчисленні за допомогою рекурентної процедури підматриць найкоротших шляхів зростаючої розмірності. Втім, найбільш очевидним шляхом є ітеративне застосування алгоритму знаходження всіх найкоротших шляхів від однієї вершини (SSSP) для кожної з вершин графу. В роботі розглядається алгоритм Джонсона ([2]), який полягає у ітеративному застосуванні алгоритму Дейкстри до кожної з вершин графу. Нотацію алгоритмів подано у системі алгоритмічних алгебр Глушкова (САА).

Архітектури сучасних GPU від NVIDIA

Графічні адаптери спочатку використовувалися лише для обробки зображень. Основною особливістю сучасних графічних адаптерів є наявність набору потокових мультипроцесорів (Streaming multiprocessor, SM), які є універсальними пристроями обробки даних, що уможливлює широке використання GPU для обчислення широкого кола задач (технологія GPGPU – General Purpose GPU).

CUDA (Computed Unified Device Architecture) – це архітектура паралельних обчислень, розроблена компанією NVIDIA для спрощення програмування GPGPU за рахунок використання високорівневих API. В моделі програмування CUDA CPU визначають як «хост», а GPU – як пристрій (обчислювальний) [3]. Фактично, пристрій CUDA є багатоядерним процесором, що використовується для обчислення задач, та який може бути задіяним лише з CPU. За таксономією Флінна, принцип роботи CUDA пристроїв є близьким до принципу SIMD (Singe Instruction – Multiple Data), з деякими уточненнями: кожен потік не лише виконує одну й ту ж саму інструкцію над своєю підмножиною даних, але й може мати власний шлях виконання в залежності від заданих умов. Розробники CUDA називають такий підхід SIMT (Single Instruction, Multiple Thread), тобто одна інструкція виконується одночасно багатьма потоками, при чому поведінка кожного окремого потоку нічим не обмежується. Таким чином, основною програмною обчислювальною одиницею є потік. Блок – це набір одночасно виконуваних потоків, що можуть взаємодіяти за допомогою спільної пам'яті та бар'єрної синхронізації. Сітка – це набір блоків, що виконують одну і ту ж саму функцію-ядро [3, 4].

CUDA визначає розширення мови C/C++, яке дозволяє створювати код, що виконуватиметься на відеоадаптері, визначаючи С функції-ядра (kernels). Кожен потік можна однозначно визначити за допомогою ряду контекстних змінних.

1. gridDim – векторна змінна, що зберігає розмірності сітки.
2. blockIdx – векторна змінна, що зберігає індекс (3-вимірний) блоку в сітці.
3. blockDim – векторна змінна, що зберігає розмірність блоку.
4. threadIdx – векторна змінна, що зберігає індекс (3-вимірний) потоку в блоці.
5. warpSize – змінна, що зберігає розмір warp групи [5].

Наприклад, наступна функція-ядро додає 2 числа:

```
__global__ void add(int* a, int* b, int* c){
  *c = *a + *b;
}
int main(void){
  ... //ініціалізація змінних, виділення пам'яті на GPU
  add<<<1, 1>>>(a, b, c); //запуск 1-потокової 1-блокової програми на GPU
  return 0;
}
```

Система Алгоритмічних Алгебр

Для формалізованого представлення алгоритмів функціонування абстрактної моделі ЕОМ В.М. Глушков запропонував математичний апарат систем алгоритмічних (мікропрограмних) алгебр (САА).

Фіксована САА являє собою двоосновну алгебраїчну систему, основами якої є множина операторів і множина умов [6]. Операції САА поділяються на логічні і операторні. До логічних відносяться узагальнені булеві операції і операція лівого множення оператора на умову (призначена для прогнозування обчислювального процесу), а до операторних – основні конструкції структурного програмування (послідовне виконання, циклічне виконання тощо).

Теорема Глушкова: Для довільного алгоритму існує (в загальному випадку не єдина) САА, в якій цей алгоритм може бути представлений регулярною схемою. Тобто, якщо визначити основи САА для конкретного алгоритму, можна представити цей алгоритм у вигляді схеми і проводити подальші трансформації й оптимізації вже не з алгоритмом, а з його САА схемою. Деякі основні операції алгоритмічної алгебри та їх аналоги у процедурних мовах програмування наведено в табл. 1, 2:

Таблиця 1. Сигнатура операцій САА

Сигнатура операцій алгоритмічної алгебри (САА)	**Відповідні оператори мови Паскаль**
Композиція: A * B (або $A \times B$)	A;B;
α-диз'юнкція $_{\alpha}(A \vee B)$	if α then A else B;
α-ітерація $_{\alpha}\{A\}$	while α do A;
Обернена α-ітерація $\{A\}_{\alpha}$	repeat A until not α

Таблиця 2. Сигнатура операцій САА-М

Сигнатура паралельних операцій САА-М	**Позначення операції**
Асинхронна диз'юнкція $A \mid\mid B$ (або $A \overset{\bullet}{\vee} B$)	Бінарна операція паралельного виконання операторів A і B на підструктурах певної моделі
Фільтрація $\underline{u}$	Унарна операція, що породжує оператори-фільтри
Синхронна диз'юнкція $A \vee B$	Бінарна операція синхронного застосування операторів A і B

Алгоритм Джонсона. Ідея алгоритму полягає у багаторазовому застосуванні алгоритму Дейкстри до кожної з вершин графу. Таким чином можна отримати найкоротші шляхи між будь-якими парами вершин. Недоліком алгоритму Дейкстри є неможливість його застосування для графів з від'ємними вагами ребер. Якщо ж в графі є ребра з від'ємними вагами (але відсутні цикли негативних ваг), можна спробувати звести задачу до випадку невід'ємних ваг, замінивши функцію ваги w на нову функцію ваги $\hat{w}$ таку, щоб.

1. Найкоротші шляхи не змінилися: для будь-якої пари вершин $u, v \in V$ найкоротший шлях з u в v з точки зору функції w є також найкоротшим шляхом з точки зору $\hat{w}$ і навпаки.

2. Ваги всіх ребер з точки зору функції $\hat{w}$ є невід'ємними [2].

Введемо наступні позначення:

$G(V,E)$ – вхідний граф;

V – початкова розмірність графа;

$isEdge(i,j)$ – предикат, значенням якого буде істина, якщо існує ребро з вершини i в j;

d_{ij} – довжина найкоротшого знайденого шляху з i в j;

w_{ij} – довжина ребра з вершини i в j;

$\{h_i\}, i = \overline{0, V+1}$ – множина найкоротших відстаней від доданої вершини до всіх вершин графа;

$setWay(i,j,val)$ – встановлення значення (val) шляху d_{ij};

$setEdge(i,j,val)$ – встановлення значення (val) ребра з вершини i в j;

$add(Q,i)$ – додавання елементу i в колекцію Q;

$erase(Q,i)$ – видалення елементу i з колекції Q;

$eraseAll(Q,u)$ – видалення всіх елементів колекції u з колекції Q;

$size(Q)$ – отримання кількості елементів у колекції Q.

Тому першим кроком алгоритму є побудова графа $G' = (V', E')$, де # $V' = V \cup \{s\}$, а s – деяка нова вершина. При цьому мають бути створені ребра # $\{(s,v) : v \in V \cup w_{sv} = 0\}$. Оператор *extendGraph* розширює заданий граф додатковою вершиною (з порядковим індексом V), додаючи ребра вагою 0 від доданої вершини до усіх існуючих:

$$extendGrap\,h =_{i<V} \{setEdge(V,i,0)\}. \quad (1)$$

Для формалізації алгоритмів введемо такі оператори:

оператор *init* проводить ініціалізацію перед виконанням алгоритмів Дейкстри та Беллмана – Форда, встановлюючи найкоротші відстані від початкової вершини (q) до усіх інших вершин, окрім q, як ∞:

$$init(q) =_{i<V} \{_{i=q} (setWay(q,i,0) \vee setWay(q,i,\infty)) \times (++i)\}; \quad (2)$$

оператор *RelaxEdge* проводить релаксацію ребра з вершини u в v. Умовою релаксації є мінімум шляху від початкової вершини q до вершин, що розглядаються:

$$RelaxEdge(q,u,v) =_{isEdge(u,v)} (_{d_{uv} > d_{qu} + w_{uv}} (setWay(q,v,d_{qu} + w_{uv}) \vee E) \vee E); \quad (3)$$

оператор *RelaxEdges* проводить релаксацію всіх ребер у графі:

$$RelaxEdges(q) =_{u<V} \{_{v<V} \{RelaxEdge(q,u,v) \times (++v)\} \times (++u)\}. \quad (4)$$

Алгоритм Беллмана – Форда. Для коректного виконання алгоритму вхідний граф не може мати цикли негативної ваги. Для перевірки наявності такого циклу використовується алгоритм Беллмана – Форда для вершини s. Масив найкоротших шляхів від вершини s до усіх інших вершин, отриманий як результат виконання алгоритму Беллмана – Форда (який буде містити елементи, відмінні від 0 у випадку негативних ваг ребер), дозволяє перейти до вагової функції $\hat{w}$.

Згідно алгоритму Беллмана – Форда, після $(V-1)$ циклів релаксації усіх ребер встановлені шляхи d_{uv} є найкоротшими. Тому виконання умови релаксації після $(V-1)$ циклів релаксації усіх ребер означає наявність в графі цикла негативних ваг [2], що і перевіряє умова *cnc*:

$$cnc(q) = \bigwedge_{u=1}^{u<V} \bigwedge_{v=1}^{v<V} (isEdge(u,v) \wedge d_{qv} < d_{qu} + w_{uv}). \quad (5)$$

Таким чином, для алгоритма Беллмана – Форда з початковою вершиною q можна сформувати наступну схему з використанням апарату систем алгоритмічних алгебр Глушкова (САА):

$$BellmanFor\,d(q) = init(q)\times_{i<V-1}\{\mathrm{Re}\,laxEdges(q)\times(++i)\}\times cnc(q). \quad (6)$$

Результатом виконання алгоритму Беллмана – Форда є булеве значення, що вказує на наявність циклів негативної ваги у графі. Також алгоритм визначає найкоротші шляхи від доданої вершини *"q"* до усіх існуючих вершин графа (задає значення $\{h_i\}$).

Якщо у графі немає циклів негативної ваги, можна трансформувати ваги його ребер:

$$UpdateWeights=_{i<V+1}\{_{j<V+1}\{_{isEdge(i,j)}(w_{ij}=(w_{ij}+h_i-h_j)\vee E)\times(++j)\}\times(++i)\}. \quad (7)$$

Алгоритм Дейкстри. Алгоритм Дейкстри є по суті послідовним і знаходить найкоротші шляхи від заданої вершини до всіх інших вершин у графі за $O(V\log V+E)$ час. В алгоритмі Дейкстри виділяють 2 типи вершин: встановлені та невстановлені. Встановленими вершинами називають ті, для яких визначений шлях від початкової вершини і всі вихідні ребра яких релаксуються (або вже релаксовані). На початку алгоритму до масиву встановлених вершин додається початкова вершина, і релаксуються ребра, що з неї виходять. На наступній ітерації визначається вершина U, довжина до якої від початкової вершини є найменшою серед невстановлених вершин. Ця вершина переноситься в масив встановлених вершин і її вихідні ребра релаксуються. Ітерації повторюються, поки всі вершини не будуть перенесені в масив встановлених вершин [2].

Введемо наступні додаткові позначення для формалізації послідовної версії алгоритму Дейкстри:

Q – множина нерозглянутих (невстановлених) вершин;

$ExtractMin(Q,q,u)$ – оператор знаходження вершини u такої, що не була розглянута раніше та відстань від початкової вершини до якої (d_{qu}) є найменшою:

$$ExtractMin\,(Q,q,u) = (mv = d_{q,Q_0})\times(mi=0)\times_{i\in Q}\{(v=Q_i)\times_{mv<d_{qv}}(mv=d_{qv}\times$$

$$\times(mi=i)\times(++i)\vee E)\times(v=Q_i)\times erase(Q,mi)\}. \quad (8)$$

Алгоритм Дейкстри ($D(q)$) можна формалізувати наступним чином:

$$SubD1(Q) = ExtractMin(Q,q,u)\times_{v<V}\{RelaxEdge(q,u,v)\times(++v)\},$$

$$D(q) = init(q)\times_{Q\neq\varnothing}\{SubD1(Q)\}. \quad (9)$$

Результат виконання алгоритму Дейкстри дає масив найкоротших шляхів від однієї вершини графу до всіх інших в рамках нової вагової функції $\hat{w}$. Щоб перейти до початкової вагової функції w, потрібно виконати операцію зворотньої трансформації ваг:

$$to(D,i)=_{j<V}\{setWay(i,j,D_j+h_j-h_i)\times(++j)\}, \quad (10)$$

де $\{D_j\}, j=\overline{0,V}$ – множина найкоротших шляхів з вершини i до всіх інших вершин графу (отриманих алгоритмом Дейкстри).

Використовуючи позначення, можна сформувати послідовну регулярну схему алгоритму Джонсона:

$$\alpha = BellmanFord(V),$$

$$SubJohns\,1(V)=_{i<V}\{to(D(i),i)\times(++i)\},$$

$$SubJohns\,2(V)=_{\alpha}(UpdateWeig\,hts\times SubJohns\,1(V)\vee E),$$

$$Johnson\;=extendGrap\;h(V)\times SubJohns\;2(V). \quad (11)$$

З точки зору оптимізації і прискорення алгоритму, основний акцент має ставитися на прискорення алгоритму Дейкстри, оскільки саме в цьому виконуються основні обчислення.

Створення паралельних схем

Класичний підхід до розпаралелювання алгоритмів APSP заключається в паралельному виконанні багатьох ітерацій алгоритму SSSP. Такий підхід є доцільним для виконання обчислень на CPU. При використанні технології GPGPU та моделі CUDA велику роль у підвищенні продуктивності має рівень різноманітності гілок виконання потоків. За моделлю SIMT різні потоки можуть мати різні шляхи виконання інструкцій. Втім, на апаратному рівні різні шляхи виконання виконуються послідовно, навіть якщо певна умова розгалуження виконується лише для одного потоку. Розробники CPU архітектур доклали багато зусиль для створення компенса-

ційних заходів, зокрема, спекулятивного виконання та передбачення переходів. На GPU таких оптимізацій не було виконано, тому використання розгалужень, шляхи виконання яких сильно відрізняються, призводить до суттєвої втрати продуктивності [7].

В роботі запропоновано паралельні схеми, що ґрунтуються на паралельному виконанні обчислень всередині ітерацій SSSP. Це дає можливість звести рівень розгалуження гілок до мінімуму.

Паралельні схеми алгоритму Дейкстри. Алгоритм Дейкстри складається з двох циклів – зовнішній на кожній ітерації обирає вершину для релаксації всіх ребер, що виходять з неї; внутрішній цикл – власне цикл релаксації ребер. Паралелізація зовнішнього циклу означає, що на кожній ітерації обирається більш ніж одна вершина, ребра якої можуть бути релаксовані без впливу на коректність результатів алгоритму. Паралелізація внутрішнього циклу означає, що вихідні ребра обраної вершини релаксуються одночасно.

Паралелізація зовнішнього циклу. На кожній ітерації алгоритм Дейкстри виконує пошук вершини u такої, що не була розглянута раніше та відстань від початкової вершини до якої (d_{qu}) є найменшою. Чим більша кількість обраних вершин на кожній ітерації – тим більший рівень паралелізму.

Таким чином, якщо при послідовній версії обиралася одна вершина u (з умови мінімальності d_{qu}), то в паралельній версії u – множина вершин v, для яких справджується нерівність:

$$d_{qv} \le d_{qu} + \Delta\,, \tag{12}$$

де Δ – довжина найменшого ребра в графі [8]. Такий варіант було обрано для економії пам'яті та обчислювальної потужності за рахунок малого зниження продуктивності. Для збільшення швидкості виконання алгоритму (за разунок додаткової пам'яті) можна замінити Δ на Δ_v, де Δ_v – довжина найменшого ребра, що виходить з вершини v. Така заміна дасть можливість корегувати умову «прийнятності» для кожної окремої вершини, що дозволить збільшити рівень паралелізму. Тому оператор (8) набуває наступного вигляду:

$$ExtractMin(Q,q,u) = ExtractLimit(Q,q,m)\times_{i<size(Q)} \{_{d_{qi}<d_{qm}+\Delta}(add(u,l)\vee E)\times$$

$$\times(++i)\}\times eraseAll(Q,u). \tag{13}$$

ExtractLimit – оператор знаходження вершини, відстань від початкової вершини до якої є найменшою (аналог (8) для послідовної схеми алгоритму, але без видалення знайденої вершини з Q). Таким чином, в схемі оператора (13) обробку кожної вершини в Q можна проводити паралельно, синхронізуючи потоки бар'єром $B_Q(i)$, який затримує виконання потоків, поки їх не набереться кількість Q (внутрішня організація бар'єра наведена в [9]):

$$PExtractMin(Q,q,u) = ExtractLimit(Q,q,m)\times$$

$$\times \overset{size(Q)}{\underset{i=1}{\bullet\vee}} (_{d_{qi}<d_{qm}+\Delta}(add(u,l)\vee E)\times B_Q(size(Q)))\times eraseAll(Q,u). \tag{14}$$

Таким чином, алгоритм Дейкстри, розпаралелений по зовнішньому циклу, на кожній ітерації бере в роботу більш ніж одну вершину та паралельно для кожної з них відбувається релаксація всіх ребер, що з неї виходять. Після цього процесу гілки синхронізуються бар'єром, алгоритм знаходить новий набір вершин для обробки і починається наступна ітерація. Формалізувати цю схему можна наступним чином:

$$PSubD1(u) = \overset{size(u)}{\underset{l=1}{\bullet\vee}} (_{v<V}(RelaxEdge(q,u_l,v)\times(++v)\}\times B_u(size(u))),$$

$$Dijkstra(q) = init(q)\times_{Q\neq\varnothing}\{PExtractMin(Q,q,u)\times PSubD1(u)\}. \tag{15}$$

Як видно з формалізованої схеми, кількість потоків дорівнює кількості вершин у графі. Структура алгоритму наведена далі:

```
thrust::device_vector<int> Q(V);
INIT<<<blocks, threads>>>(Q, M, C, q);
while(Q.size() != 0)
{
 RELAX_WITH_MASK<<<blocks, threads>>>(G, M, C);
 threshold = ExtractMin(Q, C);
 UPDATE_MASK<<<blocks, threads>>>(Q, C, M, threshold);
}
```

На кожній ітерації релаксація вихідних ребер відбувається не для всіх вершин, а з певною маскою:

```
RELAX_WITH_MASK(G, M, C)
{
 tid = threadx.Id;
 if(M[tid] == true)
 {
        <для всіх ребер [edgeId] що виходять з вершини tid виконати>
        atomicMin(C[edgeId], G[edgeId] +  C[tid]);
 }
}
```

Як показано вище, вершини переносяться у масив встановлених, якщо відстань від початкової вершини до них є меншою за поріг:

```
UPDATE_MASK(Q, C, M, threshold)
{
 tid = thread.Id;
 if(C[Q[tid]] <= threshold)
 {
        M[Q[tid]] = true;
        Q[tid] = -1; // після виконання елементи '-1' будуть видалені
 }
}
```

Розробникам застосувань під GPU потрібно вирішувати не лише алгоритмічні задачі, але й задачі вирішення конфліктів банків, оптимізації розгалужень тощо. При досить складних задачах кроки оптимізації є неочевидними та погіршують читабельність та зрозумілість коду. Розробники CUDA створили бібліотеку шаблонів на C++ для CUDA GPU застосувань, аналогічну до STL. *Thrust* вирішує низку проблем, пов'язаних з оптимізацією застосувань під конкретні архітектури, або ж використання різних підходів для пристроїв, що підтримують різні версії CUDA. Розробники можуть повністю концентруватися на вирішенні високорівневих задач, делегуючи *Thrust* вибір обчислювальної реалізації.

Основними контейнерами *Thrust* є вектори: *host_vector* та *device_vector*. Вміст *host_vector* зберігається в загальній пам'яті комп'ютера, а вміст *device_vector* зберігається в пам'яті GPU. Контейнери реалізують низку стандартних операцій та дають можливість глибокого розширення [10].

З формальної схеми видно, що у векторі *Q* зберігаються індекси невстановлених вершин. Алгоритм Дейкстри на кожній ітерації потребує знаходження індексу вершини, відстань від початкової вершини до якої є найменшою. За допомогою *Thrust* задача вирішується наступним шляхом:

```
struct compare_unsettled_value
{
 const int* data;
 compare_unsettled_value(int* ptr) { data = ptr; }
 __host__ __device__
        bool operator()(int lhs, int rhs){
               return this->data[lhs] < this->data[rhs];
 }
};

ExtractMin(Q, C) //Q - device_vector<int>
{
  ...

  thrust::detail::normal_iterator<device_ptr<int>> minIndex = thrust::min_element(
thrust::device, Q.begin(), Q.end(), compare_unsettled_value(C);
  ...
}
```

У вищенаведеному блоці коду використовується бібліотечний метод *min_element*, що виконує згортку масиву (заданого межами *Q.begin*() – *Q.end*()) операцією знаходження мінімуму. Оскільки в масиві зберігаються не значення відстані, а індекси, вводиться структура *compare_unsettled_value*, яка за заданими індексами проводить порівняння відстаней відповідних вершин. Всі обчислення проводяться на GPU, про що свідчить явно задана політика *thrust::device* [10].

Паралелізація внутрішнього циклу. Як було сказано вище, паралелізація внутрішнього циклу означає одночасну релаксацію всіх ребер обраної вершини. Таким чином, оператор (13) залишається такою ж, як і в послідовній версії ((8)), але змінюється логіка роботи з оператором (3): наразі кожен потік представляє собою ребро, яке може бути релаксовано незалежно. Після паралельної обробки відбувається бар'єрна синхронізація по V потокам, після якої обробка поточної вершини вважається закінченою і алгоритм обирає вершину для наступної ітерації.

$$PSubD2(u) = \overset{V}{\underset{v=1}{\dot{\vee}}} (RelaxEdge(q,u,v) \times B_v(v)),$$

$$Dijkstra(q) = init(q) \times_{Q \neq \varnothing} \{ExtractMin(Q,u) \times PSubD2(u)\}. \quad (16)$$

Паралелізація алгоритму Беллмана – Форда. Оскільки алгоритм Беллмана – Форда виконується в алгоритмі Джонсона лише раз, навіть при ідеальній паралельній версії великого виграшу отримати не вдасться. Втім, алгоритм може опрацьовувати кожну вершину паралельно, тому оператор (4) у паралельному виконанні має наступний вигляд:

$$RelaxEdges(q) = \overset{V}{\underset{u=1}{\dot{\vee}}} (_{u<V} \{RelaxEdge(q,u,v) \times (++v) \times B_v(u)\}). \quad (17)$$

Схема всього алгоритму залишається без змін.

Аналіз отриманих результатів

Обчислювання виконувалися з використанням процесора Intel® Core™ i7-4770 3.4ГГц, 8 Гб оперативної пам'яті DDR3 та GeForce GTX 770 (Kepler) GPU. Граф представлено у вигляді матриці суміжності (рисунок).

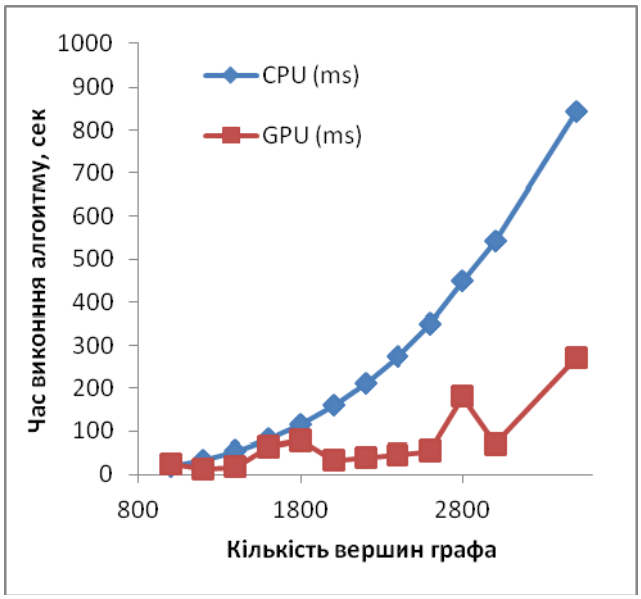

Рисунок. Залежність часу виконання алгоритму Джонсона від кількості вершин графа

Висновки

1. Основною задачею оптимізації алгоритму Джонсона є підвищення швидкодії алгоритму Дейкстри при зростанні розмірності вихідного графа за рахунок використання технології GPGPU.
2. Паралельне виконання ітерацій алгоритму Дейкстри є недоцільним для GPGPU, тому основними принципами розглянутих паралельних версій алгоритму Дейкстри є паралелізація за вершинами (кожен CUDA потік представляє собою окрему вершину) та паралелізація за ребрами (кожен CUDA потік представляє собою ребро, що релаксується).
3. Використання CAA/CAA-M Глушкова дозволяє абстрагуватися від конкретної програмної платформи та особливостей її реалізації, приділяючи більше уваги самій схемі алгоритму і концепціям та методам її трансформації.
4. Експериментально підтверджено перевагу використання розглянутих паралельних схем алгоритму Джонсона при застосуванні архітектури CUDA. Розглянута GPGPU реалізація досягає приросту швидкості виконання у 2–7 разів порівняно з CPU реалізацією.

1. *Погорілий С.Д., Білоус Р.В.* Генетичний алгоритм розв'язання задачі маршрутизації в мережах. // Матеріали VII Міжнародної науково-практичної конференції з програмування УкрПРОГ'2010. – 2010. – № 2–3. – С. 171–177.
2. *Томас Х. Кормен, Чарльз И. Лейзерсон, Рональд Л. Ривест, Клиффорд Штайн.* Алгоритмы: Построение и анализ. Третье издание. – М.: Вильямс, 2013. – 1328 с.
3. NVidia (2015). *CUDA C Programming Guide* [Online] September 2015. Available from: http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf [Accessed: 22-nd Dec 2015].
4. *Боресков А.В., Харламов А.А.* Основы работы с технологией CUDA. – М.: ДМК Пресс, 2010. – 232 с.
5. *Погорілий С. Д., Вітель Д. Ю., Верещинський О. А.* Новітні архітектури відеоадаптерів. Технологія GPGPU. Частина 1. // Реєстрація, зберігання і оброб. даних. – 2012. – Т. 14, № 4.
6. *Anisimov A.V., Pogorilyy S.D., Vitel D.Yu.* About the Issue of Algorithms formalized Design for Parallel Computer Architectures. Applied and Computational Mathematics. – 2013, Vol. 12, № 2. – P. 140–151.
7. *Tianyi D.H. & Abdelrahman T.S.* Reducing Branch Divergence in GPU Programs. Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units. New York, NY, USA. – 2011.
8. *Crauser A. et al.* A parallelization of Dijkstra's shortest path algorithm. Mathematical Foundations of Computer Science. – 1998. – P. 722-731.
9. *Погорілий С.Д., Мар'яновський В.А., Бойко Ю.В., Верещинський О.А.* Дослідження паралельних схем алгоритму Данцига для обчислювальних систем зі спільною пам'яттю // Математичні машини і системи. – 2009. – № 4.
10. *Bell N. & Hoberock J.* Thrust: A Productivity-Oriented Library for CUDA [Online] 2011. Available from: https://thrust.googlecode.com/files/Thrust%20-%20A%20Productivity-Oriented%20Library%20for%20CUDA.pdf [Accessed: 22-nd Dec 2015].

References

1. Pogorilyy S.D. & Bilous R.V. (2010) Genetic algorithm for solving routing problem in networks. In 7-th international scientific programming conference UKRPROG'2010. Kyiv. p.p. 171-177.
2. Cormen T. H. et al. (2013) Introduction to algorithms, 3rd Ed. Cambridge: MIT Press.
3. NVidia (2015). CUDA C Programming Guide [Online] September 2015. Available from: http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf [Accessed: 22-nd Dec 2015].
4. Boreskov A.V. & Kharlamov A.A (2010) CUDA technology fundamentals. Moscow: DMC Press.
5. Pogorilyy S.D., Vitel D. Yu. & Vereschinsky O.A. (2012) Modern video adapter architectures. GPGPU technology. Part 1. Data registration, storage and processing. 14 (4).
6. Anisimov A.V., Pogorilyy S.D. & Vitel D.Yu. (2013) About the Issue of Algorithms formalized Design for Parallel Computer Architectures. Applied and Computational Mathematics. 12 (2). p.p. 140-151 .
7. Tianyi, D. H. & Abdelrahman, T. S. (2011). Reducing Branch Divergence in GPU Programs. Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units. New York, NY, USA.
8. Crauser, A. et al (1998). A parallelization of Dijkstra's shortest path algorithm. Mathematical Foundations of Computer Science, P. 722–731.
9. Pogorilyy S.D., Maryanovsky V.A., Boyko Yu.V. & Vereshinsky O.A. (2009) Research of Danzig algorithm parallel schemes for computing systems with shared memory. Mathematical Machines and Systems. 4.
10. Bell, N. & Hoberock, J. (2011). Thrust: A Productivity-Oriented Library for CUDA [Online] 2011. Available from: https://thrust.googlecode.com/files/Thrust%20-%20A%20Productivity-Oriented%20Library%20for%20CUDA.pdf [Accessed: 22-nd Dec 2015].

Про авторів:

Погорілий Сергій Дем'янович,
доктор технічних наук, професор факультету радіофізики, електроніки та комп'ютерних систем.
Індекс Гірша: Google Scholar – 5, Scopus – 2.
Кількість наукових публікацій у вітчизняних виданнях – 200.
Кількість наукових публікацій в зарубіжних виданнях – 45.
http://orcid.org/відсутній.

Слинько Максим Сергійович,
студент факультету радіофізики, електроніки та комп'ютерних систем.
Кількість публікацій – 0.
http://orcid.org/0000-0001-9667-8729.

Місце роботи авторів:

Київський національний університет імені Тараса Шевченка.
03022, Київ, проспект Академіка Глушкова, 2, корпус 5.
Тел.: (044) 526 0522,
E-mail: sdp@univ.net.ua, maxim.slinko@gmail.com.