DOI: 10.17117/na.2017.01.03.117

Поступила (Received): 26.01.2017

<http://ucom.ru/doc/na.2017.01.03.117.pdf>

Савченко Н.В.
Исследование возможностей повышения
производительности алгоритма
с применением технологии CUDA

Savchenko N.V.
Possibility of increasing the algorithm
performance using CUDA technology

Многоядерные центральные процессоры могут исполнять лишь несколько потоков одновременно, а графические процессоры Nvidia с технологией CUDA обрабатывают сразу тысячи потоков с высоким уровнем информационной нагрузки для решения одной задачи, что позволяет приложениям работать с большей эффективностью. При этом сложность операций по обработке данных должна соответствовать цене перемещения данных на устройство и обратно

Ключевые слова: параллельные вычисления, многопоточность, технология CUDA

Савченко Наталья Викторовна

Магистрант

Новосибирский государственный технический университет

г. Новосибирск, пр. К. Маркса, 20

Multi-core CPUs can execute a few threads at the same time, and GPUs with CUDA-technology process thousands of threads with high information at once, which allows applications to run more efficiently. At the same time the complexity of the data processing operations should be consistent with the price of data movement to the GPU-device and vice versa

Key words: parallel computing, multi-threading, CUDA

Savchenko Natalya Viktorovna

Master

Novosibirsk state technical university

Novosibirsk, K. Marksa ave., 20

Тестовый алгоритм работает с массивом `input[]`, содержащим данные типа `float`. По умолчанию величина массива – 480 элементов. В результате выполнения алгоритма должен быть получен массив `output[]` с такой же размерностью. Переменная `N` (или `delta`) обозначает величину диапазона значений исходного массива, участвующих в вычислении нового элемента для каждого элемента `input[]`. Последовательная реализация на CPU, исходного алгоритма, предлагаемого к распараллеливанию:

```
for (int i = 0; i < input.length; i++) {  
    for (int j = i - N; j <= i + N; j++) {  
        if (j < 0 || j >= input.length)  
            continue;  
        for (int k = i - N; k <= i + N; k++) {
```

```

if (k < 0 || k >= input.length)
    continue;
output[i] += (input[i] - input[j]) * (input[i] - input[k]) *
(input[k] - input[j]);
    }
    }
}

```

В сводной таблице 1 представлено описание различных разработанных алгоритмов CUDA-ядра и среднее по 20 тестам время их исполнения.

Таблица 1. Сравнение различных вариантов реализации ядра

Ядро	Особенности	Время выполнения кода ядра при SIZE=480, N=100 (ms)
Kernel 1	В каждом блоке параллельно в разных потоках запускаются вычисления для различных j, и в каждом потоке последовательно перебираются все k для каждого j.	0,005835
Kernel 2	В каждом блоке параллельно в разных потоках запускаются вычисления для различных j, и в каждом потоке последовательно перебираются все k для каждого j. Промежуточные данные из разных потоков записываются в массив __shared__ float cache[512]. Суммирование промежуточных данных осуществляется по принципу редукции.	0,004763
Kernel 3	Для каждого блока в shared-памяти объявлен свой массив __shared__ float sharedSrc[480], в который дублируются данные из исходного массива. В каждом блоке параллельно в разных потоках запускаются вычисления для различных j, и в каждом потоке последовательно перебираются все k для каждого j. Промежуточные данные из разных потоков записываются в массив __shared__ float cache[512]. Суммирование промежуточных данных осуществляется по принципу редукции.	0,004306
Kernel 4	Использование константной памяти для хранения исходного массива. В каждом блоке параллельно в разных потоках запускаются вычисления для различных j, и в каждом потоке последовательно перебираются все k для каждого j. Промежуточные данные из разных потоков записываются в массив __shared__ float cache[512]. Суммирование промежуточных данных осуществляется по принципу редукции.	0,004851
Kernel 5	выделение в глобальной памяти графической карты места для массива разностей всех пар элементов: массив M[]. При вычислении произведений разностей обращение к массиву M[]. Редукция для суммирования промежуточных значений.	0,005184
Kernel 6	Использование двумерных гридов. Размерность грида по X отображает перебор элементов исходного массива, по Y – перебор индексов k, количество потоков в блоке – перебор по индексу j.	0,005220

Ядро	Особенности	Время выполнения кода ядра при SIZE=480, N=100 (ms)
	В каждом блоке исходный массив дублируется в shared-памяти в массиве sharedSrc[]. Shared-массив cache[] для каждого блока заполняется результатами вычисления суммы произведений относительно индекса k. Редукцией суммируется этот массив, записывая сумму в нулевую ячейку. Далее эти суммы заносятся в массив M[], память cache[] теперь используется для суммирования с редукцией этих сумм по индексу j	

Проведенные тестирования различных вариантов использования возможностей CUDA для оптимизации заданного алгоритма позволили сделать ряд выводов.

Исходный набор данных целесообразно дублировать в shared-память блоков, если предполагается очень большое количество обращений к этим данным, относительно их объема, так как само копирование может стоить некоторого времени. Такой подход должен ускорять доступ к данным, а также препятствовать высокой конкуренции потоков при обращении к входным данным, если они располагаются в общей, глобальной памяти.

Использование константной памяти окажется менее эффективным, чем использование дублирования исходных данных в Shared-памяти, если исходный массив достаточно мал в масштабах потенциальных возможностей использования константной памяти. Копирование исходного массива в shared-память блоков не займет много времени, но даст ощутимый прирост производительности.

Сокращение количества повторяющихся арифметических операций за счет создания массива для хранения промежуточных результатов вычисления в глобальной памяти, может быть оправдано в том случае, если промежуточные данные в массиве будут впоследствии использоваться большое количество раз, а на вычисление их каждый раз могло бы тратиться очень много времени за счет сложности расчетов.

При желании существенно увеличить число запускаемых параллельно потоков за счет соответственно значительного увеличения числа блоков следует избегать злоупотребления дублированием исходных данных в shared-память, и созданием крупных буферных массивов в глобальной памяти, так как это может ухудшить производительность.

Имея в оптимизируемом алгоритме операции сложения или умножения большого ряда чисел, следует отдать должное применению редукции.

Наконец, на вопрос о том, в какой ситуации обоснована замена линейного выполнения альтернативным вариантом с распараллеливанием на видеокарте, можно сказать следующее. Сложность операций по обработке данных должна соответствовать цене перемещения данных на устройство и обратно. Код, который переносит данные для кратковременного использования небольшим числом потоков, не позволит получить выигрыш производительности, или этот

выигрыш будет небольшим. В идеальном случае большое число потоков должно произвести большой объем обработки.

© 2017, Савченко Н.В.

*Исследование возможностей повышения
производительности алгоритма с применением
технологии CUDA*

© 2017, Savchenko N.V.

*Possibility of increasing the algorithm performance
using CUDA technology*