УДК 004.272

А.А. Новокрещенов, В.П. Хранилов

МЕТОД АВТОМАТИЧЕСКОГО РАСПАРАЛЛЕЛИВАНИЯ ЦИКЛОВ И ГЕНЕРАЦИИ ПАРАЛЛЕЛЬНЫХ ПРОГРАММ ДЛЯ АРХИТЕКТУРЫ CUDA

Нижегородский государственный технический университет им. Р.Е. Алексеева

Сформулирована задача распараллеливающей трансляции последовательных вложенностей циклов. В рамках поставленной задачи предложены методы: распараллеливания циклов; учета особенностей организации процесса параллельных вычислений; генерации кода, если в качестве целевой архитектуры выступает nVidia CUDA.

Ключевые слова: автоматическое распараллеливание, вложенность циклов, генерация кода, CUDA.

Задача автоматической распараллеливающей трансляции последовательной программы представляет собой последовательность преобразований, которая приводит к построению параллельной формы исходной программы. Параллельная программа – это такая, результаты выполнения которой идентичны результатам выполнения исходной программы, эффективно использующая возможности целевой параллельной архитектуры и записанная на языке входной программы (возможно, с некоторыми языковыми расширениями, характерными для компилятора целевой архитектуры).

Такая постановка задачи трансляции актуальна. Это связанно с тем, что поиск параллельности, скрытой в последовательной программе, и ее последующая адаптация для целевой параллельной архитектуры крайне затруднительны вручную. Использование одного языка как на входе, так и на выходе транслятора, избавляет от необходимости написания полноценного компилятора – для получения параллельного исполняемого кода можно воспользоваться уже существующим компилятором конкретной архитектуры. Схема предлагаемой процедуры трансляции представлена на рис. 1.

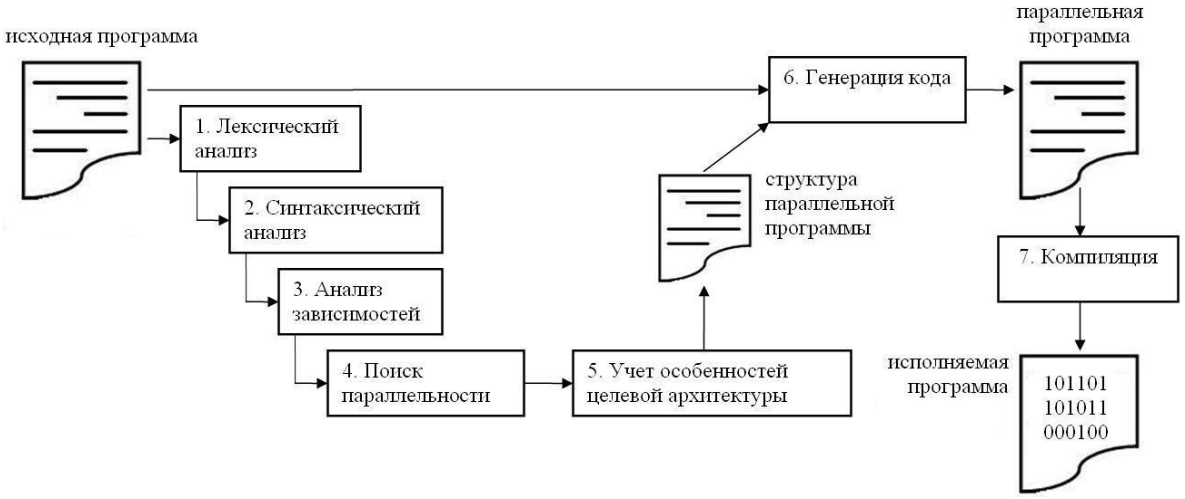

Рис. 1. Процедура получения параллельного кода

Непосредственное распараллеливание исходной программы происходит в рамках четвертой, пятой и шестой фаз трансляции (рис. 1), поэтому именно здесь должна быть учтена организация процесса параллельных вычислений на целевой архитектуре.

Целью данной работы является построение метода, выполняющего функциональность четвертой, пятой и шестой фаз трансляции для параллельной архитектуры CUDA. Другими

словами, разрабатываемый метод должен обеспечивать поиск параллельности, опираясь на информацию о зависимостях в исходной программе (фаза 4), адаптировать результаты фазы 4 для параллельной архитектуры CUDA (фаза 5) и генерировать параллельную программу (фаза 6). Для получения исполняемой программы, результат фазы 6 компилируется с использованием CUDA-компилятора. Следует отметить, что наиболее критичной частью разрабатываемого метода является фаза поиска параллельности, так как именно эта часть в основном определяет структуру параллельной программы.

Учитывая постановку задачи, данная статья содержит три части, каждая из которых, определяет устройство отдельной фазы трансляции.

Каждый алгоритм поиска параллельности предназначен для анализа определенного типа программных конструкций – существуют методы для поиска параллельности на уровне функций, отдельных циклов и т.д. Разрабатываемый в рамках данной работы метод должен осуществлять распараллеливание циклов, поэтому фаза поиска параллельности должна быть основана на алгоритме поиска параллельности в циклах. Особое внимание циклам уделяется по вполне определенным причинам. Циклические конструкции заключают в себе большой объем вычислений (особенно в программах, обрабатывающих большие массивы данных) и являются хорошим источником параллелизма. Более того, параллелизм, заключенный в циклах, относительно просто идентифицируется и хорошо масштабируется в пространстве процессоров, по сравнению с параллелизмом, скрытым в других типах программных конструкций [1].

На сегодняшний день существует ряд мощных алгоритмов, предназначенных для автоматического распараллеливания вложенностей циклов [1, 2]. Их можно разделить на две основные группы – алгоритмы поиска временных планов и алгоритмы поиска разбиений пространства процессоров. Алгоритмы первого типа выполняют конвейеризацию циклов, определяя то, какие экземпляры инструкций исходной программы могут быть запущены одновременно. Алгоритмы второго типа позволяют распределить независимые потоки экземпляров инструкций между процессорами.

Параллельная архитектура CUDA относится к классу SIMD-архитектур. Характерной особенностью архитектур этого класса является наличие одного потока инструкций и множества потоков данных. Другими словами, все процессоры, задействованные в вычислениях, в один момент времени выполняют одну и ту же инструкцию, но применительно к разным элементам данных. Чаще всего, это достигается за счет параметризации SIMD-программы номером процессора. Аналогичный подход используется и при написании программ для CUDA. Поэтому в рамках поставленной задачи поиск параллельности предпочтительно выполнять с помощью алгоритмов нахождения разбиений пространства процессоров.

При написании программ для архитектуры CUDA на передний план выходит понятие потока. Поток – это последовательность инструкций, выполняемая на отдельном процессоре. Программа для CUDA в ходе выполнения, представляет собой множество параллельно исполняемых потоков. Каждый поток обладает своим уникальным идентификатором. Понятие потока является основным при разработке CUDA-программ потому, что за привязку потоков к процессорам отвечает среда выполнения CUDA и программист не способен вмешаться в данный процесс. Однако то, какие инструкции будут выполнять различные потоки, полностью зависит от программиста. Поэтому задачу поиска параллельности предлагается сформулировать следующим образом: из множества всех экземпляров инструкций исходной программы необходимо сформировать максимально возможное число независимых потоков. Независимыми потоками считаются такие, в которых зависимости по данным между экземплярами инструкций не выходят за пределы потока. Другими словами, все зависимые по данным экземпляры инструкций исходной программы должны выполняться в рамках одного потока. Зависимость по данным между экземплярами инструкций существует в том случае, если они обращаются к одному и тому же элементу данных, при этом хотя бы одно из обращений является обращением на запись [1].

Архитектура CUDA не допускает существование потоков с отрицательными номерами.

Сформируем список требований к алгоритму поиска параллельности, на котором будет основана четвертая фаза трансляции:

1. Результатом работы алгоритма должен являться набор аффинных функций, отображающих экземпляры инструкций исходной программы в пространство потоков.
2. В качестве входа алгоритм должен принимать информацию о зависимостях по данным, присутствующим в распараллеливаемом участке кода в форме многогранного сокращенного графа зависимостей (МСГЗ) [2].
3. Для каждой инструкции распараллеливаемого участка кода исходной программы должна быть составлена своя функция.
4. Функции должны быть составлены таким образом, чтобы число потоков, в рамках которых будет выполняться распараллеливаемый участок кода, было максимально возможным для конкретной программы.
5. Функции должны быть составлены таким образом, что бы все потоки выполнения были независимы.
6. Каждый отдельный поток должен идентифицироваться целым неотрицательным числом.

Функции разбиения должны быть пригодны для генерации параллельного кода уже существующими методами генерации. МСГЗ используется потому, что является одной из распространенных форм представления зависимостей [2].

Ни один из проанализированных в рамках данной работы алгоритмов поиска аффинных функций разбиения не удовлетворяет требованию 6, характерному для архитектуры CUDA. В данной работе предлагается алгоритм поиска, отвечающий всем шести требованиям.

Для представления алгоритма введем следующие обозначения. Инструкции исходной программы будем обозначать в соответствии с порядком их следования в исходном коде – 1, 2, 3 и т.д.; экземпляр инструкции i - i^*; $\vec{x}_i$ - вектор индексных переменных циклов, охватывающих инструкцию i, конкретное значение вектора $\vec{x}_i$, определяющее i^* будем обозначать $\vec{x}_i^*$; Π_i - функция отображения инструкции i; $\Delta_{i,j} = \Pi_j(\vec{x}_j) - \Pi_i(\vec{x}_i)$ - разность функций отображения инструкций i и j, представляющая собой номер потока; D_i - пространство итераций инструкции i; $D_i^{\wedge}$ - многогранник независимых экземпляров инструкции i; $D_{i,j}$ - многогранник зависимостей, определяющий зависимость между инструкциями i и j, при этом инструкция j является зависимой. Следует отметить, что пространство итераций D_i представляет собой параметрический многогранник от $\vec{x}_i$, вектора структурных параметров циклов и константы. Многогранник зависимостей $D_{i,j}$ представляет собой параметрический многогранник от $\vec{x}_i$, $\vec{x}_j$ векторов структурных параметров циклов, охватывающих инструкции i, j и константы.

Первый шаг предлагаемого метода распараллеливания заключается в поиске размерности пространства потоков. Размерность пространства потоков прямым образом влияет на размерность искомых функций отображения.

Приведем следующие интуитивно понятные рассуждения. Независимые экземпляры инструкций исходной программы могут выполняться в разных потоках одновременно. Только в этом случае степень параллельности, скрытая в исходной программе, не будет снижена при распараллеливании. Размерность должна быть выбрана таким образом, чтобы все независимые экземпляры инструкций могли быть взаимнооднозначно отображены в пространство потоков.

Для определения размерности пространства, в котором данные потоки будут расположены, для каждой инструкции i входной программы необходимо выполнить следующий алгоритм.

1. Вход: D_i - пространство итераций инструкции *i*, все многогранники зависимостей, в которых участвует инструкция *i* - $D_{j,i}$, $D_{k,i}$, и т.д.;

2. К каждому многограннику зависимостей применить алгоритм исключений Фурье-Моцкина для исключения из них всех индексных переменных, входящих в многогранники пространств итераций инструкций, от которых зависит инструкция *i*. Результатом данного шага являются многогранники $\overline{D}_{j,i}$, $\overline{D}_{k,i}$ и т.д. соответственно. На понятийном уровне данные многогранники представляют собой множества экземпляров инструкций *i*, зависимых от экземпляров инструкций *j, k,* и т.д. соответственно;

3. Определить множество независимых экземпляров инструкции *i* - $D_i^{\wedge}$ как разницу многогранников: $D_i^{\wedge} = D_i - \overline{D}_{i,j} - \overline{D}_{k,j} - \ldots$ Как видно, $D_i^{\wedge}$ так же является многогранником, который представляет собой какую-то часть D_i;

4. Определить, от какого количества индексных переменных зависит $D_i^{\wedge}$;

5. Очевидно, наиболее удобно, когда размерность пространства потоков для инструкции *i* будет совпадать с размерностью пространства, в котором расположен $D_i^{\wedge}$, т.е. количеством индексных переменных, определенных на предыдущем шаге.

Размерность пространства потоков, в котором будет выполняться параллельная программа, предлагается определять как максимальную размерность многогранника $D_i^{\wedge}$ среди размерностей, полученных в ходе применения означенного алгоритма ко всем инструкциям.

Вторым шагом предлагаемого метода распараллеливания является составление прообразов аффинных функций отображения для каждой вершины входного МСГЗ (каждой инструкции исходной программы). Данная процедура основана на аффинной форме леммы Фаркаша [3]. Лемма утверждает, что функция $f(\vec{x})$ будет неотрицательна на всем многограннике $D \equiv A^* \vec{x} + \vec{b} \geq \vec{0}$ тогда и только тогда, когда она представляет собой положительную аффинную комбинацию вида

$$f(\vec{x}) = \lambda_0 + \vec{\lambda} * (A^* \vec{x} + \vec{b}), \lambda_0 \geq 0, \vec{\lambda} \geq \vec{0}. \quad (1)$$

Множители λ_i называют множителями Фаркаша. Учитывая условие 6 списка требований к алгоритму, функция отображения каждой инструкции должна быть неотрицательна на всем пространстве итераций соответствующей инструкции:

$$\Pi_i(\vec{x}_i) = \lambda_0 + \vec{\lambda} * D_i, \lambda_0, \vec{\lambda} \geq \vec{0}. \quad (2)$$

Таким образом, для каждой инструкции *i* исходной программы должен быть составлен прообраз функции отображения $\Pi_i(\vec{x}_i)$ в форме (2), размерность которой должна совпадать с $D_i^{\wedge}$.

Процедура распараллеливания сводится к поиску множителей Фаркаша, которые приводят к построению функций отображения, удовлетворяющих требованиям 4, 5, 6 вышеозначенного списка требований к алгоритму.

Третьим шагом предлагаемого метода является учет зависимостей, а более точно – учет условия 5. Для этого необходимо, чтобы для каждой дуги входного МСГЗ выполнялось следующее условие. Все экземпляры инструкций *i, j,* принадлежащих многограннику $D_{i,j}$, помечающему дугу МСГЗ, должны выполняться в рамках потока с одним номером:

$$\Delta_{i,j} = \Pi_j(\vec{x}_j^*) - \Pi_i(\vec{x}_i^*) = \vec{0}, \forall \{\vec{x}_j^*, \vec{x}_i^*\} \in D_{i,j}. \quad (3)$$

Учитывая то, что многогранники зависимостей можно сократить, применив к ним метод исключений Фурье-Моцкина, выражение (3) можно сократить, исключив эти же переменные:

$$\overline{\Delta}_{i,j} = \Pi(\vec{x}_j^*) - \Pi(h(\vec{x}_i^*)) = \vec{0}, \forall \vec{x}_j^* \in \overline{D}_{i,j}, \quad (4)$$

где функция *h* – аффинная функция, определяющая соответствие между зависимыми экзем-

плярами инструкций 1 и 2. Информация соответствия – это ни что иное, как уравнения, содержащиеся в соответствующих многогранниках зависимостей.

Результатом третьего шага является набор систем уравнений относительно искомых множителей Фаркаша, которые содержат в себе все ограничения, вытекающие из зависимостей данных. Способ получения систем уравнений из выражений вида (4) будет детально изложено в примере. Следует отметить, что данные системы всегда представляют собой системы линейных уравнений.

Четвертым шагом предлагаемого метода является учет условия 4. Чтобы количество параллельных потоков выполнения было максимальным, необходимо выполнение каждого независимого экземпляра каждой инструкции в отдельном потоке. Из геометрических соображений очевидно, что последнее будет справедливо, если для каждой инструкции i исходной программы все точки многогранника $D_i^{\wedge}$ будут взаимнооднозначно отображаться в пространство потоков. Учитывая, что пространства итераций и пространство потоков являются целочисленными, то взаимнооднозначное отображение возможно лишь в том случае, если матрица, составленная из коэффициентов $\Pi_i(\vec{x}_i)$ при индексных переменных оставшихся в $D_i^{\wedge}$, является унимодулярной [4]. В конечном счете, требование об унимодулярности должно быть выражено в виде набора линейных ограничений, которые необходимо будет добавить к соответствующей системе линейных уравнений, полученной на предыдущем шаге.

Всякая элементарная матрица унимодулярна [5]. Основываясь на данном утверждении можно получить линейные уравнения, учитывающие условие 4. Данные уравнения следует добавить к соответствующим системам, полученным на третьем шаге. Также к этим системам добавляются неравенства, требующие положительности множителей Фаркаша.

Пятым шагом предлагаемого метода является отыскание решения на полученных системах уравнений/неравенств. В общем случае это задача целочисленного линейного программирования. В данной работе предлагается находить решение следующим способом. Если с каждым неизвестным множителем Фаркаша связать одномерное пространство, то система превращается в многогранник в Z^d, где d – это количество множителей Фаркаша. Конечное решение предлагается определять как точку лексикографического минимума данного многогранника. Для определения лексикографического минимума на многограннике следует ввести отношение порядка, т.е. упорядочить координатные оси пространства, в котором определен многогранник. Упорядочить их следует таким образом, чтобы коэффициенты при индексных переменных оказались минимальными. Результатом данного шага является вектор искомых множителей Фаркаша.

Следующей фазой трансляции является учет особенностей целевой архитектуры. Функции разбиения практически полностью определяют структуру искомой параллельной программы. Однако они не учитывают особенностей организации процесса параллельных вычислений на целевой архитектуре. Поэтому необходимо определить факторы, наиболее сильно влияющие на производительность. Для дальнейших рассуждений сделаем краткое пояснение организации вычислений на архитектуре CUDA.

Вычислительное устройство CUDA представляет собой массив мультипроцессоров, выполняющих одну задачу в рамках большого числа потоков. Другими словами, данное вычислительное устройство можно отнести к классу SIMD. Код, который должен выполняться на устройстве, необходимо оформить в виде специальной функции – ядра (kernel). Потоки, выполняющие ядро, организованы в виде решетки потоков (grid), структура которой определяется программистом и должна быть известна на этапе компиляции. Программист должен знать заранее, в каком количестве потоков будет выполняться ядро, и организовать эти потоки в решетку с наиболее подходящей структурой.

Решетка состоит из блоков потоков (thread block) фиксированного размера, каждый из которых обладает уникальным идентификатором. Каждый поток «знает» идентификатор блока, к которому он принадлежит, и собственный идентификатор в рамках данного потока.

Как показывает практика, структура решетки потоков сильно влияет на производительность. Таким образом, для получения эффективной параллельной программы мало знать, в каком количестве потоков будет выполняться программа, а также, какие экземпляры инструкций будет выполнять каждый поток (на этот вопрос отвечают функции разбиения), еще необходимо организовать потоки в решетку с такой структурой, которая приведет к высокой эффективности выполнения.

Эксперименты, провидимые в рамках данной работы, показали, что наибольшая эффективность выполнения достигается, когда блок содержит в себе 512 потоков. Это совпадает с данными технической документации. При этом неправильный выбор величины блока может привести к тому, что эффективность выполнения того же самого ядра снизится в разы.

Фаза учета особенностей целевой архитектуры опирается на результаты фазы поиска параллельности (набор функций разбиения), и ее целью является построение структуры решетки потоков, блоки которой содержат 512 потоков.

Если общее количество потоков не кратно 512, то в данной работе предлагается увеличить его до ближайшего кратного 512 сверху. Добавленные потоки представляют собой «фиктивные» потоки, которые не выполняют никаких действий – запускаются и тут же завершаются. Такой подход позволит подобрать подходящую структуру решетки потоков, тем самым увеличив быстродействие. Поскольку добавленные потоки не выполняют действий, они не являются источником накладных расходов.

Последняя фаза трансляции представляет собой процедуру генерации кода. Для генерации кода в данной работе используется метод, предлагаемый в [1]. Данный метод предназначен для генерации SIMD-программ по аффинным функциям отображения, что и требуется. Результатом фазы генерации кода является код, полученный с помощью указанного метода, со следующими тривиальными модификациями. Код, получаемый в результате использования метода из [1], представляет собой код, вложенный в циклы с итерациями в пространстве потоков. Для написания CUDA-программ данные циклы не требуются. Поэтому первая модификация – это устранение внешних циклов с итерациями в пространстве потоков. Чтобы расширение пространства потоков не привело к искажению результатов выполнения, необходимо ввести проверку в виде оператора ветвления в результирующий код, запрещающую фиктивным потокам выполнение каких-либо действий. Составление подобных ветвлений, проверяющих значение идентификатора потока, представляет собой вторую модификацию кода.

Рассмотрим применение предложенного метода трансляции на примере. На рис. 1 представлена последовательная программа, выполняющая перемножение полиномов степени N.

```
for (i = 0; i <= N; i++)
{
    for (k = 0; k <= N; k++)
    {
        if (i == 0 || k == 0)
            C[i - k + N] = A[i] * B[-k + N];                    /* 1 */

        if( i !=0 && k != 0)
            C[i - k + N] = C[i - k + N] + A[i] * B[-k + N];     /* 2 */
    }
}
```

Рис. 2. Программа-пример

Как видно, данная программа содержит две инструкции – 1 и 2. Следует отметить, что инструкции «окружены» операторами ветвлений, которые определяют срабатывание отдельной инструкции. Другими словами, данные операторы «вырезают» области пространства итераций, определяемого парой циклов от i и j.

Учитывая границы циклов и соответствующие операторы ветвления, получаем следующие пространства итераций:

$$D_1 \equiv \left(\begin{array}{cccc} 1 & 0 & 0 & 0 \\ \hline 0 & 1 & 0 & 0 \\ 0 & -1 & 1 & 0 \end{array}\right) \times \begin{pmatrix} i_1 \\ k_1 \\ N \\ 1 \end{pmatrix} \frac{=0}{\geq \vec{0}} \cup \left(\begin{array}{cccc} 0 & 1 & 0 & 0 \\ \hline 1 & 0 & 0 & 0 \\ -1 & 0 & 1 & 0 \end{array}\right) \times \begin{pmatrix} i_1 \\ k_1 \\ N \\ 1 \end{pmatrix} \frac{=0}{\geq \vec{0}},$$

$$D_2 \equiv \begin{pmatrix} 1 & 0 & 0 & -1 \\ -1 & 0 & 1 & 0 \\ 0 & 1 & 0 & -1 \\ 0 & -1 & 1 & 0 \end{pmatrix} \times \begin{pmatrix} i_2 \\ k_2 \\ N \\ 1 \end{pmatrix} \geq \vec{0}$$

Рис. 3. Пространства итераций инструкций 1 и 2

Пространство итераций первой инструкции представляет собой объединение двух многогранников. Такое объединение возникло из-за оператора «ИЛИ» в первом ветвлении.

Результатом фазы анализа зависимостей является МСГЗ, представленный на рис. 4.

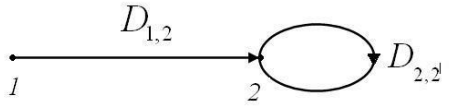

Рис. 4. МСГЗ программы-примера

Данный граф содержит две вершины, соответствующие инструкциям 1 и 2 исходной программы, две дуги, соответствующие двум типам зависимостей. Многогранники зависимостей представлены ниже.

$$D_{1,2} \equiv \left(\begin{array}{cccccc} 1 & 0 & -1 & 0 & 0 & 1 \\ 0 & 1 & 0 & -1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & -1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -1 & -1 & 0 \end{array}\right) \times \begin{pmatrix} i_1 \\ k_1 \\ i_2 \\ k_2 \\ N \\ 1 \end{pmatrix} \frac{=0}{\geq \vec{0}} \cup \left(\begin{array}{cccccc} 1 & 0 & -1 & 0 & 0 & 1 \\ 0 & 1 & 0 & -1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ \hline 1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -1 & 1 & 0 \end{array}\right) \times \begin{pmatrix} i_1 \\ k_1 \\ i_2 \\ k_2 \\ N \\ 1 \end{pmatrix} \frac{=0}{\geq \vec{0}}$$

Рис. 5. Многогранник зависимостей $D_{1,2}$

В силу того, что пространство итераций инструкции 1 представляет собой объединение двух многогранников, то многогранник $D_{1,2}$, также представляет собой объединение двух многогранников.

На рис. 6 представлен многогранник зависимостей $D_{2,2'}$.

Как видно из представленных МСГЗ, в программе присутствует два типа зависимостей. Экземпляры инструкции 2 зависят от экземпляров инструкции 1, если для индексных переменных выполняется одно из условий:

$$i_1 = i_2 - 1 \wedge j_1 = j_2 - 1 \wedge i_1 = 0 \wedge 0 \leq j_1 \leq N \wedge 0 \leq i_2 \leq N \wedge 0 \leq j_2 \leq N$$

или $i_1 = i_2 - 1 \wedge j_1 = j_2 - 1 \wedge j_1 = 0 \wedge 0 \le i_1 \le N \wedge 0 \le i_2 \le N \wedge 0 \le j_2 \le N$. Экземпляры инструкции 2 зависят от других экземпляров инструкции 2, если для индексных переменных выполняется условие: $i_2 = i_2^| - 1 \wedge j_2 = j_2^| - 1 \wedge 1 \le i_2 \le N \wedge 1 \le j_2 \le N \wedge 1 \le i_2^| \le N \wedge 1 \le j_2^| \le N$.

$$D_{2,2^|} \equiv \left(\begin{array}{cccccc} 1 & 0 & -1 & 0 & 0 & 1 \\ 0 & 1 & 0 & -1 & 0 & 1 \\ \hline 1 & 0 & 0 & 0 & 0 & -1 \\ -1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & -1 \\ 0 & -1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & -2 \\ 0 & 0 & -1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & -2 \\ 0 & 0 & 0 & -1 & 1 & 0 \end{array}\right) \times \begin{pmatrix} i_2 \\ k_2 \\ i_2^| \\ k_2^| \\ N \\ 1 \end{pmatrix} \underset{\ge \vec{0}}{=0}$$

Рис. 6. Многогранник зависимостей $D_{2,2^|}$

В соответствии с предложенным методом поиска функций отображения, первое, что необходимо сделать, это определить размерность пространства потоков. В соответствии с предложенным алгоритмом определения размерности пространства потоков применяем метод исключений Фурье-Моцкина к многогранникам зависимостей. Из $D_{1,2}$ исключаем переменные, входящие в D_1, а из $D_{2,2^|}$ исключаем переменные, входящие в D_2, получая $\overline{D}_{1,2}$ и $\overline{D}_{2,2^|}$ соответственно.

Учитывая то, что $D_{1,2}$ представляет собой объединение двух многогранников, то метод исключения Фурье-Моцкина применяем к каждому многограннику в отдельности.

$$\overline{D}_{1,2} \equiv \left(\begin{array}{cccc} 1 & 0 & 0 & -1 \\ \hline 0 & 1 & 0 & -1 \\ 0 & -1 & 1 & 0 \end{array}\right) \times \begin{pmatrix} i_2 \\ k_2 \\ N \\ 1 \end{pmatrix} \underset{\ge \vec{0}}{=0} \cup \left(\begin{array}{cccc} 0 & 1 & 0 & -1 \\ \hline 1 & 0 & 0 & -1 \\ -1 & 0 & 1 & 0 \end{array}\right) \times \begin{pmatrix} i_2 \\ k_2 \\ N \\ 1 \end{pmatrix} \underset{\ge \vec{0}}{=0}$$

$$\overline{D}_{2,2^|} \equiv \begin{pmatrix} 1 & 0 & 0 & -2 \\ -1 & 0 & 1 & 0 \\ 0 & 1 & 0 & -2 \\ 0 & -1 & 1 & 0 \end{pmatrix} \times \begin{pmatrix} i_2^| \\ k_2^| \\ N \\ 1 \end{pmatrix} \ge \vec{0}$$

Рис. 7. Многогранники зависимых инструкций

Далее определяем множества независимых инструкций. Глядя на МСГЗ, очевидно, что все экземпляры инструкции 1 независимы, так как в вершину 1 не входит ни одна дуга. Таким образом, многогранник независимых инструкций для инструкции 1 будет совпадать с многогранником пространства итераций данной инструкции. Многогранник независимых инструкций для инструкции 2 будет определяться следующим образом: $D_2^\wedge = D_2 - \overline{D}_{1,2} - \overline{D}_{2,2^|}$.

Данное выражение объясняется тем, что некоторые экземпляры инструкции 2 зависят от экземпляров инструкции 1, при этом множество зависимых экземпляров содержится в $\overline{D}_{1,2}$. С другой стороны, некоторые экземпляры 2 зависят от других экземпляров 2, такие экземпляры содержатся в $\overline{D}_{2,2^|}$. Формально, многогранники $\overline{D}_{1,2}$ и $\overline{D}_{2,2^|}$ зависят от разных индексных переменных, однако участвовать в операции нахождения разности они все равно могут, так как они представляют собой различные части одного и того же пространства итераций. Представим многогранники независимых экземпляров инструкций.

$$D_1^{\wedge} \equiv \left(\begin{array}{cccc} 1 & 0 & 0 & 0 \\ \hline 0 & 1 & 0 & 0 \\ 0 & -1 & 1 & 0 \end{array}\right) \times \left(\begin{array}{c} i_1 \\ k_1 \\ N \\ 1 \end{array}\right) \begin{array}{l} =0 \\ \hline \geq \vec{0} \end{array} \bigcup \left(\begin{array}{cccc} 0 & 1 & 0 & 0 \\ \hline 1 & 0 & 0 & 0 \\ -1 & 0 & 1 & 0 \end{array}\right) \times \left(\begin{array}{c} i_1 \\ k_1 \\ N \\ 1 \end{array}\right) \begin{array}{l} =0 \\ \hline \geq \vec{0} \end{array},$$

$$D_2^{\wedge} \equiv \varnothing$$

Рис. 8. Многогранники независимых инструкций

Видно, все экземпляры инструкции 2 являются зависимыми. Учитывая то, что размерность каждого из многогранников, образующих $D_1^{\wedge}$, равна единице, то размерность пространства потоков также равна единице. Для операций над многогранниками в данной работе используется библиотека Polylib.

Следующим шагом, в соответствии с предложенным методом распараллеливания, является составление прообразов аффинных функций процессорного отображения для каждой инструкции *i* исходной программы, как аффинной комбинации на многограннике D_i в форме (2). Учитывая одномерность пространства потоков, прообразы будут скалярными функциями. Следует отметить, что многогранник D_1 является частью многогранника пространства итераций, образуемого циклами по *i* и *j*, в силу того, что первый оператор ветвления «вырезает» часть последнего многогранника, определяя D_i. Поэтому, при составлении прообраза функции отображения для инструкции 1 можно потребовать положительности данной функции на многограннике, образуемом циклами. Таким образом, прообразы аффинных функций отображения для инструкции 1 и 2 будут соответственно представлять собой

$$\begin{aligned} \Pi_1(\vec{x}_1) &= \lambda_0 + (\lambda_1 - \lambda_2)i_1 + (\lambda_3 - \lambda_4)k_1 + (\lambda_2 + \lambda_4)N, \\ \Pi_2(\vec{x}_2) &= \lambda_5 + (\lambda_6 - \lambda_7)i_1 + (\lambda_8 - \lambda_9)k_1 + (\lambda_7 + \lambda_9)N - \lambda_6 - \lambda_8. \end{aligned} \quad (5)$$

Для гарантии неотрицательности номеров потоков множители Фаркаша должны быть неотрицательны $\lambda_k \geq 0, 0 \leq k \leq 9$.

Следующим шагом, в соответствии с предложенным методом распараллеливания, является учет зависимостей. Для каждой дуги МСГЗ должно выполняться условие (3). Конкретно для рассматриваемого примера:

$$\Delta_{1,2} = \Pi_2(\vec{x}_2^*) - \Pi(\vec{x}_1^*) = 0, \forall \vec{x}_2^*, \vec{x}_1^* \in D_{1,2},$$

$$\Delta_{2,2^|} = \Pi(\vec{x}_2^{*|}) - \Pi(\vec{x}_2^*) = 0, \forall \vec{x}_2^{*|}, \vec{x}_2^* \in D_{2,2^|}$$

Учитывая то, что $i_1 = i_2 - 1 \wedge k_1 = k_2 - 1 \wedge i_1 = 0$ или $i_1 = i_2 - 1 \wedge k_1 = k_2 - 1 \wedge k_1 = 0$ для дуги МСГЗ, помеченной $D_{1,2}$, и $i_2 = i_2^| - 1 \wedge k_2 = k_2^| - 1$ для дуги, помеченной $D_{2,2^|}$, переходим к форме (4):

$$\overline{\Delta}_{1,2} = (-\lambda_3 + \lambda_4 + \lambda_8 - \lambda_9)\, j_2 + (-\lambda_2 - \lambda_4 + \lambda_7 + \lambda_9)N$$
$$-\lambda_0 + \lambda_3 - \lambda_4 + \lambda_5 - \lambda_7 + \lambda_8 = 0, \forall\{i_2 = 1, k_2 \geq 1\} \in \overline{D}_{1,2},$$

$$\overline{\Delta}_{1,2} = (-\lambda_1 + \lambda_2 + \lambda_6 - \lambda_7) i_2 + (-\lambda_2 - \lambda_4 + \lambda_7 + \lambda_9)N$$
$$-\lambda_0 + \lambda_1 - \lambda_2 + \lambda_5 - \lambda_6 + \lambda_9 = 0, \forall\{i_2 \geq 1, k_2 = 1\} \in \overline{D}_{1,2},$$

$$\overline{\Delta}_{2,2^|} = \lambda_6 - \lambda_7 + \lambda_8 - \lambda_9 = 0, \forall\{i_2^|, k_2^|\} \in \overline{D}_{2,2^|}$$

Тривиальное решение, когда данное условие будет справедливо, представляет собой следующую систему уравнений:

$$\begin{cases} -\lambda_3 + \lambda_4 + \lambda_8 - \lambda_9 = 0, \\ -\lambda_2 - \lambda_4 + \lambda_7 + \lambda_9 = 0, \\ -\lambda_0 + \lambda_3 - \lambda_4 + \lambda_5 - \lambda_7 + \lambda = 0, \\ -\lambda_1 + \lambda_2 + \lambda_6 - \lambda_7 = 0, \\ -\lambda_0 + \lambda_1 - \lambda_2 + \lambda_5 - \lambda_6 + \lambda_9 = 0, \\ \lambda_6 - \lambda_7 + \lambda_8 - \lambda_9 = 0. \end{cases} \quad (6)$$

Система (6) содержит в себе все ограничения, вытекающие из зависимостей данных. Любые функции (5), удовлетворяющие данным ограничениям, будут приводить к тому, что все пары зависимых экземпляров инструкций исходной программы будут отображаться в потоки с одинаковым номером. Другими словами, система (6) гарантирует независимость потоков параллельной программы.

Следующим шагом, в соответствии с предложенным методом распараллеливания, является составление условий, приводящих к построению функций отображения, максимизирующих количество параллельных потоков. Для этого необходимо, чтобы каждый независимый экземпляр каждой инструкции выполнялся в отдельном потоке. Выше было показано, что формально для этого необходимо, чтобы матрицы, составленные из коэффициентов функций отображения (5) при индексных переменных, не обратившихся в константы в $D_1^{\wedge}$ и в $D_2^{\wedge}$ соответственно, были унимодулярными. Учитывая то, что составленные матрицы состоят из одного элемента, требование их элементарности излишне. Так как в рассматриваемом примере многогранник $D_2^{\wedge}$ представляет собой пустое множество, получаем четыре – условия:

$$\begin{cases} \lambda_1 - \lambda_2 = 1, \\ \lambda_3 - \lambda_4 = 1 \end{cases} \quad \begin{cases} \lambda_1 - \lambda_2 = 1, \\ \lambda_3 - \lambda_4 = -1 \end{cases} \quad \begin{cases} \lambda_1 - \lambda_2 = -1, \\ \lambda_3 - \lambda_4 = 1 \end{cases} \quad \begin{cases} \lambda_1 - \lambda_2 = -1, \\ \lambda_3 - \lambda_4 = -1 \end{cases} \quad (7)$$

Дополняя систему (6) поочередно одним из условий унимодулярности, а также требованием о неотрицательности множителей Фаркаша, получаем четыре результирующих системы уравнений/неравенств.

В соответствии последним шагом предлагаемого метода, решение определяем как лексикографический минимум многогранника. Для этого вводим отношение порядка – упорядочиваем координатные оси, в которых располагаются многогранники решений:

$$\lambda_5, \lambda_0, \lambda_9, \lambda_8, \lambda_7, \lambda_6, \lambda_4, \lambda_3, \lambda_2, \lambda_1.$$

Оказывается, что лексикографическим минимумом обладают только два результирующих многогранника решений, полученные дополнением системы (6) вторым и третьим условиями (7) соответственно. Для отыскания лексикографического минимума на многограннике в данной работе используется библиотека Piplib.

Представим функции отображения, полученные в результате отыскания лексикографического минимума данных многогранников:

$$\Pi_1(\vec{x}_1) = i_1 - k_1 + N, \qquad \Pi_1(\vec{x}_1) = -i_1 + k_1 + N,$$
$$\Pi_2(\vec{x}_2) = i_2 - k_2 + N \qquad \Pi_2(\vec{x}_2) = -i_2 + k_2 + N$$

Для генерации кода можно использовать любое из этих решений. Для простоты будем использовать решение, представленное слева:

$$\begin{aligned} \Pi_1(\vec{x}_1) &= i_1 - k_1 + N, \\ \Pi_2(\vec{x}_2) &= i_2 - k_2 + N. \end{aligned} \tag{8}$$

Подставляя в (4) максимальные и минимальные значения индексной переменной i и j, получаем, что пространство потоков представляет собой отрезок от нуля до $2N$. Другими словами, рассматриваемая программа-пример допускает выполнение в рамках $2N+1$ параллельно запущенных потоков.

Метод генерации кода, используемый в данной работе, подробно объясняется в [1]. Здесь же приведем его конечный результат. Назначение данного метода – уплотнение циклов и устранение избыточных проверок в параллельной программе.

```
for (ID = 0; ID <= 2*N; ID++)
  for (i = max (0, ID - N)); i <= min (N, ID); i++)
      for (k = max (0, i - ID + N); k <= min (N, i - ID + N); k ++)
      {
          if (i == 0 || k == 0)
              C[i - k + N] = A[i] * B[-k + N];                    /* 1 */

          if (i != 0 && k != 0)
              C[i - k + N] = C[i - k + N] + A[i] * B[-k + N];   /* 2 */
      }
```

Рис. 9. Код с плотными границами и без избыточных проверок

Данный код представляет собой эффективный SIMD-код, однако он не учитывает особенностей организации процесса вычислений на архитектуре CUDA. Следует напомнить, что для выполнения параллельного кода в рамках CUDA, необходимо сформировать структуру решетки потоков, которая приведет к эффективному использованию ресурсов GPU. Учитывая тесты производительности, приведенные в первой части работы, необходимо сформировать такую решетку потоков, чтобы размер блока составлял 512 потоков.

Пространство потоков для рассматриваемого примера одномерно, соответственно решетка потоков будет одномерной. С другой стороны, программа – пример допускает параллельное выполнение в N потоках. Учитывая данные обстоятельства, количество блоков в решетке будет составлять $ceil(2N+1/512)$, где $ceil$ – ближайшее целое сверху. Число фиктивных потоков будет рассчитываться по формуле - $ceil(2N+1/512)512 - N$.

Для того, чтобы расширение пространства потоков не привело к искажению результатов выполнения, вводим дополнительную проверку в результирующий код, запрещающую фиктивным потокам выполнение каких-либо действий. Так как пространство потоков расположено в пределах от нуля до $2N$ плюс определенное количество фиктивных потоков, то фиктивные потоки будут представлять собой потоки с номером, большим $2N$. Результат фазы генерации кода представлен на рис. 10.

Для проведения компьютерного эксперимента было реализовано две программы – последовательная (для выполнения перемножения полиномов с использованием CPU), и параллельная (для выполнения аналогичной операции с использованием GPU). Параллельная программа написана на основе кода с рис. 10. Код последовательной программы идентичен коду с рис. 2. Для запуска последовательных программ используется следующая конфигурация оборудования: CPU – Pentium Dual-Core 2,6 GHz, RAM – 2048 Мб. Для запуска параллель-

ных программ использовалось устройство: NVIDIA GeForce 9600 GT (64 скалярных процессора, 512 MHz каждый), 512 Мб встроенной памяти.

```
if (ID <= 2*N)
{
    for (i = max (0, ID - N)); i <= min (N, ID); i++)
        for (k = max (0, i - ID + N); k <= min (N, i - ID + N); k ++)
        {
            if (i == 0 || k == 0)
                C[i - k + N] = A[i] * B[-k + N];                    /* 1 */

            if (i != 0 && k != 0)
                C[i - k + N] = C[i - k + N] + A[i] * B[-k + N];   /* 2 */
        }
}
```

Рис. 10. Результат фазы генерации кода

Представим результаты эксперимента – время выполнения последовательной и параллельной программ при различных N.

Таблица 1

Время выполнения последовательной и параллельной программ

N	CPU, миллисекунды	GPU, миллисекунды	Фиктивных потоков
1000	17.8	2.4	47
2000	71.3	8.3	95
3000	91.4	18.3	143
5000	278.4	51.2	239
7000	455.6	94.4	335
10000	820	190.2	479

Графическая интерпретация полученных экспериментальных результатов представлена на рис. 11.

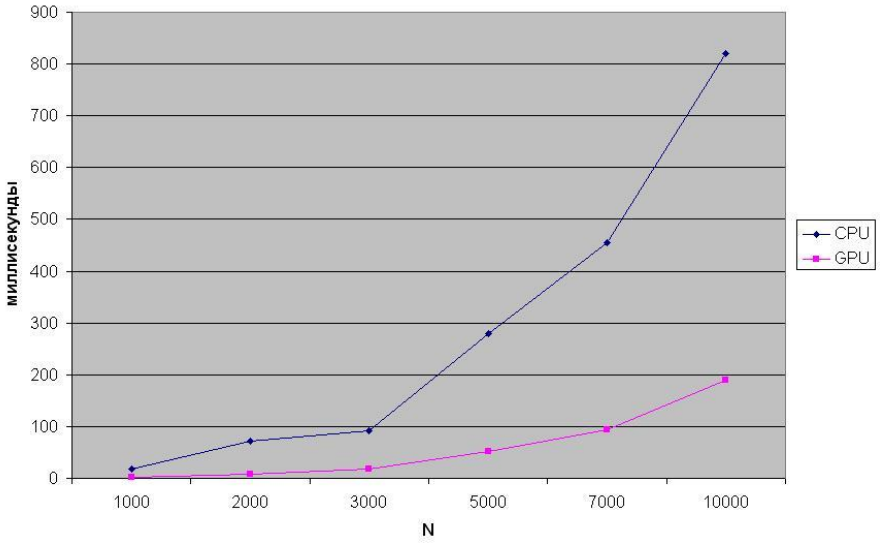

Рис. 11. Результаты эксперимента

Можно видеть, что производительность GPU практически не зависит от количества фиктивных потоков. Следует отметить, что параллельная программа оказалась заметно эф-

фективнее последовательного аналога при всех значениях N. Более того, чем больше N, тем больше разница во времени выполнения последовательной и параллельной программ, что подтверждает полезность предложенного алгоритма распараллеливания.

Библиографический список

1. Компиляторы: принципы, технологии и инструментарий / А. Ахо [и др.]. – М.: Вильямс, 2008. – 1184 с.
2. **Касьянов, В.Н.**, Реструктурирующие преобразования: алгоритмы распараллеливания циклов / В.Н. Касьянов, И.Л. Мирзуитова // Программные средства и математические основы информатики. – Новосибирск, 2004.
3. **Feautrier, P.** Some efficient solutions to the affine scheduling problem. Part 1: One dimensional time / P. Feautrier, International journal of parallel programming, 1992. V. 21
4. **Lengauer, C.** Loop parallesation in polytope model / C. Lengauer // 4th International conference of concurrency theory, 1993.
5. **Курош, А.Г.** Курс высшей алгебры / А.Г. Курош. – СПб.: Лань, 2004. – 432 с.

Дата поступления
в редакцию 12.07.2011

A.A. Novokreshchenov, V.P. Hranilov

METHOD OF AUTOMATIC LOOP PARALLELISATION AND GENERATING PARALLEL PROGRAMS FOR THE CUDA ARCHITECTURE

This article defines a problem of sequential loops parallelizing translation. The following methods are studied within the context of the problem: loops parallelizing, parallel computing process structure recording, code generation in case NVIDIA CUDA serves as target architecture.

Key words: automatic parallelization, loop nest, code generating, CUDA.