УДК 681.3

А.Е. Дорошенко, К.А. Жереб

АЛГЕБРО-ДИНАМИЧЕСКИЕ МОДЕЛИ ДЛЯ РАСПАРАЛЛЕЛИВАНИЯ ПРОГРАММ

Предложены алгебро-динамические модели и метод проверки корректности оптимизирующих преобразований для многопоточных программ и программ для графических ускорителей. Описано использование этих моделей с помощью техники переписывающих правил для доказательства корректности преобразований и повышения эффективности распараллеливания вычислений.

Введение

В настоящее время активно развиваются новые аппаратные платформы для параллельных вычислений, такие как многоядерные процессоры [1] и графические ускорители [2]. Эти устройства являются доступными по цене и установлены в большинстве современных компьютеров. При этом эти платформы позволяют существенно повысить производительность приложений, за счет их распараллеливания. Однако современные средства разработки параллельных программ для таких платформ являются достаточно сложными и требуют от разработчика использования новых моделей программирования и знания деталей аппаратной и программной реализации.

Таким образом, актуальной является задача автоматизации разработки параллельных программ. В работах [3–5] авторами предложен подход, основанный на автоматизации преобразований программ с использованием техники переписывающих правил [6–8]. Однако при использовании этого подхода возникает вопрос о корректности применяемых преобразований: будет ли преобразованная программа эквивалентна исходной, или в процессе преобразования были внесены ошибки?

В данной работе предложен метод проверки корректности преобразований для многопоточных программ и программ для графических ускорителей. Для этих платформ построены модели исполнения программ с использованием алгебро-динамического подхода [9]. В терминах построенных моделей сформулированы свойства, определяющие корректность преобразований, и приведены условия, позволяющие доказывать такие свойства. Предложены методы построения модели и проверки условий с использованием техники переписывающих правил. Приведены примеры доказательства корректности преобразований.

Вопрос доказательства корректности оптимизирующих преобразований достаточно хорошо исследован в литературе. Так, в работе [10] предложен алгоритм для доказательства корректности оптимизирующих преобразований, направленных на изменение структуры циклов. Работы [11–12] описывают системы Cobalt и Rhodium для задания преобразований программ и автоматического доказательства их корректности. Авторы [13] используют аппарат темпоральной логики для ручного доказательства корректности некоторых классических оптимизирующих преобразований. В работе [14] описан разработанный автором компилятор для подмножества языка C и доказана его корректность. Однако все эти работы рассматривают преобразования только в контексте работы оптимизирующего компилятора, тогда как в данной работе рассмотрены преобразования на уровне исходного кода.

Кроме того, в работах [10–13] используются способы записи преобразований, по сути эквивалентные переписывающим правилам, но в каждом случае эта функциональность реализована «с нуля»; в результате используются непривычные пользователю обозначения и отсутствуют

некоторые средства, стандартные для систем переписывающих правил. В данной работе используется система Termware [6–8], предоставляющая унифицированное представление преобразований различных видов. В этом смысле данная работа близка к [15], где используется система Maude [16] для автоматического доказательства определенных свойств многопоточных программ. Однако в работе [15] сделана попытка полностью автоматического доказательства свойств программ путем перебора всех вариантов исполнения, что приводит к невозможности использования такого подхода для многих практических задач в силу большой вычислительной сложности. В данной работе использован более практический подход, основанный на комбинации статического анализа и автоматической генерации тестов, поэтому рассмотренный подход применим к более широкому кругу задач.

Материал данной работы организован следующим образом. В разделе 1 построена модель исполнения для многопоточных программ, а также рассмотрен метод доказательства корректности преобразований на основе этой модели. Раздел 2 содержит аналогичную модель для программ, исполняемых на графических ускорителях. Раздел 3 описывает автоматизацию использования моделей с помощью системы переписывающих правил Termware. Работу завершают выводы и направления дальнейшей работы.

1. Модель многопоточных программ

1.1. Общее описание модели. Данный раздел описывает модель для многопоточных программ. Однако общая схема подхода, описанного здесь, будет использована также для построения модели для графических ускорителей (раздел 2).

Вначале опишем модель собственно программы (синтаксическую модель), используя алгебру алгоритмов Глушкова (АГ) [9]. Далее построим модель исполнения программы (семантическую модель). Для описания процесса исполнения используется алгебро-динамический подход [9]. На основе описанных моделей определим важнейшие свойства программ. Использование этих свойств позволяет доказывать корректность преобразований программ; рассмотрим примеры подобных доказательств для конкретных преобразований. Проверка описанных свойств проводится в автоматизированном режиме, с использованием системы Termware – этот процесс описан в разделе 3.

1.2. Модель программы. Будем использовать следующую упрощенную модель программы на современных языках программирования. Программа состоит из множества компонентов, соответствующих отдельным методам (функциям, процедурам) языка программирования:

$$P = \{P_1, ..., P_k\}.$$

Будем считать, что компонент описывается идентификатором (именем, уникальным в пределах программы), а также моделью кода:

$$P_i = (name_i, code_i).$$

Для моделирования кода будем использовать алгебру алгоритмов Глушкова. Процедурный код будем представлять в виде выражения этой алгебры. Алгебра Глушкова (АГ) является двухосновной алгеброй $A(Y,U)$, содержащей множества операторов Y и условий (предикатов) U. В алгебре определены стандартные операции: логические операции конъюнкции AND, дизъюнкции OR и отрицания NOT, последовательная композиция $THEN$, ветвление IF, итерация (цикл) $WHILE$.

На множествах операторов и условий вводятся базовые элементы, после чего можно строить различные выражения алгебры, которые будут описывать сложные операторы и условия. Базовые операторы и условия обычно зависят от предметной области. Выделим один базовый оператор, общий для всех предметных областей: вызов метода $call(P_i)$.

1.3. Модель исполнения последовательной программы. Для описания исполнения программ будем использовать общие понятия теории дискретных динамических систем [9]. Дискретная динамическая система (ДДС) определяется как тройка (S_0, S, d), где S – множество, на-

зываемое пространством состояний; $S_0 \subseteq S$ – множество начальных состояний; $d \subset S \times S$ – бинарное отношение переходов в пространстве состояний. Система может перейти из состояния s_i в состояние s_j, если $(s_i, s_j) \in d$.

Для построения модели исполнения необходимо определить интерпретацию выражений алгебры Глушкова, из которых состоит программа. Пусть V – множество переменных программы. Для простоты будем считать, что переменные не имеют типа и принимают значения в некотором универсальном множестве D. Частичные отображения $b: V \to D$ переменных на их значения будем называть состояниями памяти. Информационной средой является множество состояний памяти:

$$B = \{b : V \to D\}.$$

Тогда операторы алгебры алгоритмов интерпретируются как функции на множестве B, а условия – как предикаты на том же множестве. Интерпретация базовых операторов и условий определяется в рамках соответствующей предметной области.

Теперь можно определить ДДС, которая описывает исполнение последовательной программы. Состояния имеют вид $s = (b, R, F)$, где $b \in B$ – текущее состояние памяти, $R \in Y$ – текущее состояние управления, представленное в виде остаточной программы, $F \in (P \to Y)^*$ – стек вызовов функций, т.е. последовательность идентификаторов функций и операторов АГ, описывающих состояние управления данной функции. Начальное состояние для заданной программы имеет вид $s_0 = (b_0, Y(P_m), (P_m \to \varnothing))$, где P_m – основная функция (точка входа в программу). Отношение перехода задается следующими правилами перехода:

1) $(b, yR, F) \to (y(b), R, F)$, где $y \in Y$ – базовый оператор;

2) $(b, if\,(u, P, Q)R, F) \to$
$$\to \begin{cases} (b, PR, F), u(b) = 1, \\ (b, QR, F), u(b) = 0; \end{cases}$$

3) $(b, while(u, P)R, F) \to$
$$\to \begin{cases} (b, Pwhile(u, P)R, F), u(b) = 1, \\ (b, R, F), u(b) = 0; \end{cases}$$

4) $(b, call(P_j)R, (..., P_i \to \varnothing)) \to$
$$\to (b, Y(P_j), (..., P_i \to R, P_j \to \varnothing));$$

5) $(b, \varepsilon, (..., P_i \to R, P_j \to \varnothing)) \to$
$$\to (b, R, (..., P_i \to \varnothing)).$$

Правило 1) определяет исполнение базовых операторов. Правила 2) и 3) описывают ветвление и циклическую конструкцию. Работа с функциями описана правилами 4) (вызов функции) и 5) (возврат из функции).

Финальные состояния (из которых невозможен переход ни в какое другое состояние) имеют вид $s_f = (b, \varepsilon, \varnothing)$.

Определенную таким образом ДДС будем обозначать S^{ser}. Она моделирует исполнение заданной последовательной программы. Отметим, что система S^{ser} – детерминированная, поскольку для каждого состояния переход определен однозначно.

1.4. Модель исполнения многопоточной программы. Основываясь на описанной в предыдущем пункте модели исполнения последовательной программы S^{ser}, построим аналогичную модель для параллельной многопоточной программы.

Будем строить модель путем расширения уже построенной модели S^{ser}. В частности, модель программы остается прежней. Будем предполагать, что код всех потоков объединен в одной программе, как это реализовано во всех современных языках, поддерживающих многопоточность (таких как C/C++, C#, Java).

Для моделирования поведения потоков потребуется добавление новых операторов: запуска потока, критической секции, синхронизации. Оператор $call\_thread(P_i, T_j)$ описывает запуск функции на новом потоке. Этот оператор использует два параметра: P_i – идентификатор функции для запуска и T_j – идентификатор потока, на котором будет запущена

функция. Операторы критической секции $lock(cs_i)$ и $unlock(cs_i)$ позволяют определять участки кода, которые могут исполняться только на одном потоке. Операторы синхронизации $wait(ev_i)$, $signal(ev_i)$ и $reset(ev_i)$ используются для взаимодействия потоков.

Модель исполнения, как и в последовательном случае, строится в виде ДДС. В качестве состояний модели используются частичные отображения $s = \{T_i \to s_i\}$ множества T всех потоков на состояния отдельных потоков. Множество всех потоков T содержит специальный стартовый поток T_0, а также все потоки T_j, упоминаемые в операторах $call\_thread(P_i, T_j)$. Состояния отдельных потоков $s_i = (b, R_i, F_i) \in S^{ser}$ аналогичны использованным в последовательной модели, т.е. содержат состояние памяти b, состояние управления R_i и стек вызовов F_i. Отметим, что в модели с общей памятью состояние памяти b является общим для всех потоков. Это значит, что состояние многопоточной программы можно было бы более компактно переписать в виде пары $s' = (b, \{T_i \to (R_i, F_i)\})$, т.е. вынести общее состояние памяти из состояний отдельных потоков. Однако далее будем использовать более длинную запись с дублированием b, чтобы подчеркнуть аналогию с последовательной моделью (и облегчить задание отношения переходов).

Начальное состояние имеет вид $s_0 = \{T_0 \to s_0^{ser}\} = \{T_0 \to (b_0, Y(P_m), (P_m \to \varnothing))\}$. Исполняется только один поток, и его состояние аналогично начальному состоянию последовательной программы. Все остальные потоки явно задаются в программе. Такая модель используется в большинстве платформ для многопоточного программирования.

Отношение переходов использует уже определенное отношение d^{ser} для последовательной программы. К правилам 1)–5), определенным в п. 1.3, добавим правила для новых операторов:

6) $(b, call\_thread(P_i, T_j)R, F) \to$
$\to (b, R, F); T_j \to (b, Y(P_i), (P_i \to \varnothing))$;

7) $\{T_i \to (b, lock(cs)R, F)\} \to$
$\to \{T_i \to (b \cup \{cs \to T_i\}, R, F)\}$ при условии $b(cs) = \varnothing$ или $b(cs) = T_i$;

8) $\{T_i \to (b \cup \{cs \to T_i\}, unlock(cs)R, F)\} \to$
$\to \{T_i \to (b \cup \{cs \to \varnothing\}, R, F)\}$;

9) $(b, wait(ev)R, F) \to (b, R, F)$ при условии $b(ev) = 1$;

10) $(b, signal(ev)R, F) \to$
$\to (b \cup \{ev \to 1\}, R, F)$;

11) $(b, reset(ev)R, F) \to$
$\to (b \cup \{ev \to 0\}, R, F)$.

Правило 6) определяет создание нового потока при срабатывании оператора $call\_thread(P_i, T_j)$. Правила 7), 8) описывают поведение операторов критической секции, а правила 9)–11) – поведение операторов синхронизации.

Отношение переходов для всей многопоточной программы объединяет переходы отдельных потоков. Эта процедура производится следующим образом: выбирается некоторое подмножество активных потоков, для каждого из выбранных потоков производится переход в соответствии с отношением d^{ser+}, и новое состояние программы получается объединением состояний отдельных потоков. Новые состояния отдельных потоков используют состояние управление, определенное отношением переходов для S^{ser}. Для определения состояния общей памяти используется функция $merge: B \times B^* \to B$. Эта функция меняет состояние каждой переменной, которая была изменена хотя бы в одном из потоков. Формально,

$$merge(b_0, b_1, ..., b_k) = \{v_i \to \begin{cases} !\{b_j(v_i) \mid b_j(v_i) \neq b_0(v_i),\ j = \overline{1,k}\} \\ b_0(v_i) \end{cases}\}$$

(здесь использовано обозначение $!A$ для произвольного элемента множества A).

Построенное отношение переходов осталось дополнить еще одним правилом, описывающим завершение работы потока.

$$12)\ \{T_1 \to s_1, ..., T_{k-1} \to s_{k-1}, T_k \to (b, \varepsilon, \varnothing), T_{k+1} \to s_{k+1}, ..., T_n \to s_n\} \to \to \{T_1 \to s_1, ..., T_{k-1} \to s_{k-1}, T_{k+1} \to s_{k+1}, ..., T_n \to s_n\}.$$

Таким образом, поток, который достиг финального состояния в смысле ДДС для последовательной программы, завершает работу и исключается из списка активных потоков.

Финальные состояния для многопоточной системы в целом могут быть двух видов. В случае успешного завершения работы всех потоков ДДС переходит в состояние $s'_f = (b, \{\})$. Будем называть такое состояние *финальным*, и отождествлять его с финальным состоянием последнего потока $s_f = (b, \varepsilon, \varnothing)$. Также возможна ситуация, когда все еще исполняются один или несколько потоков, но ни один из них не может совершить переход (потому что в каждом из потоков очередным оператором является *lock* или *wait*, и в правилах 7) или 9) не выполняется условие перехода). Такие состояния будем называть *тупиковыми*.

Построенную ДДС для моделирования многопоточных программ будем обозначать S^{mt}. В отличие от системы S^{ser}, система S^{mt} не является детерминированной, поскольку при каждом переходе возможно выполнение произвольных активных потоков.

1.5. Свойства программ. Используя построенные модели, можно формально описать некоторые свойства программ. При этом возможно автоматическое вычисление таких свойств.

В данной работе рассматривается применение моделей для доказательства корректности преобразований программ. *Корректность* преобразования подразумевает тот факт, что исходная и преобразованная программы дают одинаковый результат. Формально это условие можно сформулировать следующим образом. Пусть выделено подмножество $V_R \subset V$ результирующих переменных. Тогда две программы P^1 и P^2 будем называть *эквивалентными по результату*, если для одинаковых начальных данных b_0, программы одновременно приходят или не приходят в финальные состояния $s_f^1 = (b^1, \varepsilon, \varnothing)$ и $s_f^2 = (b^2, \varepsilon, \varnothing)$, при этом финальные состояния памяти совпадают по результирующим переменным: $b^1_{V_R} = b^2_{V_R}$.

Эквивалентность по результату является наиболее общим свойством, но непосредственная проверка этого свойства практически нереализуема: требуется рассмотрение всех возможных путей исполнения программы, в зависимости как от входных данных, так и вариантов выполнения различных потоков. Поэтому необходимо введение более частных свойств, описывающих корректность преобразований, которые могут быть проверены на практике.

Сначала определим некоторые свойства операторов АГ. Для каждого оператора $y \in Y$ определены множества $R(y) \subset V$ переменных, от которых зависит результат применения оператора, и $W(y) \subset V$ переменных, которые изменяются в результате применения оператора. Два оператора $y_1, y_2 \in Y$ являются *зависимыми* (по данным), если $R(y_1) \cap W(y_2) \neq \varnothing$, или $W(y_1) \cap R(y_2) \neq \varnothing$, или $W(y_1) \cap W(y_2) \neq \varnothing$. Два оператора $y_1, y_2 \in Y$ являются *перестановочными*, (или *коммутативными*) относительно операции композиции, если $y_1 y_2 = y_2 y_1$. Очевидно, что независимые операторы будут перестановочными; обратное неверно (например, две копии одного оператора $y \in Y$ всегда перестановочны, но они зависимы, если $W(y) \neq \varnothing$).

Теперь определим следующие свойства программ: беступиковость, бесконфликтность и эквивалентность по операторам. Программу P будем называть *беступиковой*, если при ее исполнении не возни-

кают тупиковые состояния. Все последовательные программы (в модели S^{ser}), очевидно, являются беступиковыми.

В модели S^{mt} будем называть *конфликтным переходом* ситуацию, когда среди операторов, выполняющихся одновременно на разных потоках, есть зависимые. Программу P будем называть *бесконфликтной*, если при ее исполнении не возникают конфликтные переходы. Для бесконфликтных программ нет необходимости использовать функцию *merge*; вместо объединения результатов одновременного воздействия нескольких операторов можно применить эти операторы последовательно, в любом порядке.

Если известен порядок исполнения программы (последовательность переходов ДДС), можно определить историю использования базовых операторов. Для этого добавим к состоянию ДДС еще одну компоненту, $h \in Y$. В начальном состоянии положим $h = \varepsilon$. Также модифицируем правило перехода 1):

$$(b, yR, F, h) \to (y(b), R, F, hy).$$

При одновременном срабатывании нескольких правил 1) на разных потоках, будем добавлять соответствующие операторы в произвольном порядке. Оператор h в финальном состоянии будем называть историей использования операторов.

Две истории h_1, h_2 будем считать эквивалентными, если h_2 можно получить из h_1 последовательными перестановками пар операторов, которые являются коммутативными. Две программы P^1 и P^2 будем называть *эквивалентными по операторам*, если для одинаковых начальных данных b_0 они производят эквивалентные истории использования операторов.

Необходимость введения свойств беступиковости, бесконфликтности и эквивалентности по операторам обосновывается следующим утверждением.

Лемма 1. Если две программы P^1 и P^2 – беступиковые, бесконфликтные, а также эквивалентные по операторам, то они являются эквивалентными по результату.

Кроме того, как будет показано в дальнейшем, проверка этих свойств для многих задач практически реализуема.

1.6. Проверка условий. Свойства эквивалентности по операторам, беступиковости и бесконфликтности, определенные в предыдущем пункте, в общем случае являются сложными для проверки. Однако для определенных частных случаев можно указать условия, достаточные для обеспечения этих свойств, и при этом легко проверяемые. Приведем некоторые такие условия.

Условие эквивалентности по операторам фактически означает, что в программе переставлены некоторые операторы, причем эти операторы должны быть коммутативными. Возможна перестановка операторов в пределах последовательной композиции, в этом случае требование коммутативности будем обозначать как условие *CommuteThen*. Более частым случаем, однако, является перестановка операторов в пределах цикла: в этом случае отдельные итерации цикла исполняются в другом порядке, возможно в нескольких циклах. Для обеспечения эквивалентности по операторам необходимо проверить как коммутативность отдельных итераций (условие *CommuteWhile*), так и тот факт, что существует взаимно однозначное соответствие между итерациями в двух программах (условие *SameIterations*).

Заметим, что проверка условий коммутативности является достаточно сложной, поскольку требует анализа свойств всех операторов (включая вызовы функций), а также возможных вариантов исполнения программы. Возможна ситуация, когда для конкретной предметной области известны факты коммутативности некоторых операторов. Также коммутативность может быть установлена на основании независимости операторов. В разделе 3 будут рассмотрены некоторые способы проверки коммутативности с использованием системы переписывающих правил TermWare.

Для свойства бесконфликтности можно сформулировать следующее условие: любые два оператора, которые являются зависимыми, и которые могут вы-

полняться на различных потоках, должны находиться внутри критической секции с одинаковым идентификатором (будем обозначать это условие *AllNeededCS*). Непосредственная проверка этого условия требует полного анализа зависимостей, что является сложной задачей [10]. Однако в некоторых частных случаях проверка упрощается: например, независимыми будут операторы, которые меняют только локальные переменные для потока.

Для свойства беступиковости начнем со случая, когда в программе не используются операторы *wait* (т.е. потенциальные тупики могут возникнуть только из-за операторов *lock*). В этом случае можно построить следующий ориентированный граф критических секций. В качестве вершин используются все используемые в программе критические секции cs_i; дуга из cs_i в cs_j добавляется в том случае, если некоторый поток T_k может выполнить оператор $lock(cs_j)$, при этом $b(cs_i) = T_k$, т.е. тот же поток уже занял cs_i. В этом случае программа будет беступиковой, если полученный граф будет ациклическим (условие *NoCyclesCS*).

Частным случаем этого условия является ситуация, когда программа вообще не содержит вложенных критических секций (условие *NoNestedCS*). В этом случае граф критических секций не будет содержать ребер, и автоматически будет ациклическим. Данное условие достаточно простое для проверки. Кроме того, если программа ему удовлетворяет, можно ожидать, что ее синхронизационные конструкции проще для понимания разработчиками.

В случае, когда в программе используются операторы *wait*, проверка беступиковости усложняется. В этом случае для каждого оператора $wait(ev_i)$ должен найтись соответствующий оператор $signal(ev_i)$, при этом $wait(ev_i)$ должен сработать до очередного вызова $reset(ev_i)$. Проверка такого условия для всех возможных вариантов исполнения программы достаточно сложна.

Поэтому вместо общих условий отсутствия тупиков при использовании операторов *wait*, в данной работе рассматриваются определенные шаблоны (паттерны) их использования, которые гарантируют корректность. Паттерны – это комбинированные операторы, содержащие базовый оператор *wait*, и которые являются единственным способом его использования.

Рассмотрим два паттерна использования оператора *wait*. Первый, $\_Join(T_i)$, определяет ожидание окончания работы потока T_i. Данный паттерн состоит из двух частей. После последнего оператора, исполняемого на потоке T_i, вставляется оператор $signal(ev\_T_i)$. Кроме того, используется комбинированный оператор (в данном случае эквивалентный базовому) $\_Join(T_i) = wait(ev\_T_i)$. Кроме этих двух операторов, точка синхронизации ev_T_i больше нигде не используется.

Второй паттерн, $\_Barrier(ts_i)$, определяет в программе барьер, на котором каждый поток ждет все остальные потоки. Паттерн состоит из одного комбинированного оператора:

$\_Barrier(bar_i) = lock(cs\_b_i); inc(count\_b_i);$
$if\,(count\_b_i < k, unlock(cs\_b_i); wait(ev\_b_i),$
$assign(count\_b_i, 0); signal(ev\_b_i);$
$unlock(cs\_b_i))$.

Этот оператор использует дополнительную переменную $count\_b_i$, критическую секцию cs_b_i и точку синхронизации ev_b_i. В переменной $count\_b_i$ накапливается количество потоков, которые ожидают на барьере. Если это количество меньше общего количества потоков, текущий поток также ожидает; иначе все потоки прекращают ожидание, и количество потоков устанавливается равным 0.

Барьер не будет вызывать тупиков, если общее количество потоков, где он используется, равно параметру k, а также в каждом потоке срабатывает одинаковое количество операторов $\_Barrier$ (условие *CorrectBarrier*). Паттерн $\_Join(T_i)$ не будет вызывать тупиков, если беступиковой

является программа, полученная удалением оператора $\_Join(T_i)$ и всех последующих операторов, исполняющихся на том же потоке (условие *CorrectJoin*).

Отметим, что во многих платформах многопоточного программирования присутствует непосредственная реализация операторов $\_Join$ и $\_Barrier$, поэтому нет необходимости в реализации этих паттернов с использованием операторов *wait* и *signal*. Однако рассмотренные условия остаются справедливыми и в этом случае.

1.7. Пример доказательства корректности преобразования. Рассмотрим пример использования построенной модели для доказательства корректности распараллеливающего преобразования. Пусть исходная последовательная программа имеет следующий вид:

$$Ser = for(i,0,m,\ for(j,0,n,\ body(i,j))).$$

Здесь использован оператор цикла со счетчиком *for*(*var*,*min*,*max*,*body*), который выражается через общий оператор цикла *while*. Рассмотрим два случая распараллеливания. Первый случай реализуется, когда итерации внешнего цикла являются коммутативными. В этом случае возможно распараллеливание на уровне этого цикла. Рассмотрим следующее преобразование: данный участок программы переходит в параллельный эквивалент

$$Par1 = for(k,0,threads,\ call\_thread(pbody1,T_k));\ for(k,0,threads,\ \_Join(T_k)).$$

При этом вычисления переносятся в функцию *pbody*1:

$$pbody1 = for(i,min(k),max(k),\ for(j,0,n,\ \_AddLocks(body(i,j)))).$$

Эта функция повторяет структуру исходного цикла, однако границы внешнего цикла зависят от номера потока и определяют ту часть итераций, которые выполняются на данном потоке: $min(k)=k*bs$, $max(k)=min(m,(k+1)*bs)$, $bs=m/threads$.

Также в тело внутреннего цикла добавляются необходимые критические секции (операция *_AddLocks*). Эта операция заключает все зависимые операторы в критические секции, обеспечивая выполнение условия *AllNeededCS*.

Теперь можно доказать следующее утверждение.

Теорема 1. Программы *Ser* и *Par*1 эквивалентны по результату, если итерации внешнего цикла в *Ser* коммутативны.

Доказательство. В силу леммы 1 достаточно проверить беступиковость и бесконфликтность программы *Par*1 (для последовательной программы эти свойства, очевидно, выполнены) и эквивалентность по операторам программ *Ser* и *Par*1.

Беступиковость обеспечивается свойством *CorrectJoin*: при удалении операторов *_Join* остается программа, не содержащая *wait*. К этой программе применимо условие *NoNestedCS*, поскольку операция *_AddLocks* исключает добавление вложенных критических секций. Беступиковость доказана.

Бесконфликтность следует из выполнения условия *AllNeededCS*, которое также обеспечивается операцией *_AddLocks*.

Для доказательства эквивалентности по операторам необходимо проверить условия *CommuteWhile* и *SameIterations*. Первое из них выполнено по условию; второе следует из построения циклов в программе *Par*1. Действительно, можно установить взаимно однозначное соответствие между итерацией с номером *i* в программе *Ser* и итерацией с номером (*i mod bs*) в потоке *i/bs* программы *Par*1.

Таким образом, все условия выполнены и теорема доказана.

Во втором случае распараллеливания итерации внешнего цикла не являются коммутативными, но этим свойством обладают итерации внутреннего цикла. В этом случае можно было бы применить аналогичное преобразование к внутреннему циклу, однако в этом случае во внешнем цикле происходило бы многократное

создание потоков, что привело бы к значительным накладными расходам. Поэтому используем преобразование, которое создает потоки только один раз, но при этом гарантирует правильный порядок выполнения итераций внешнего цикла.

Для этого преобразования используется следующая функция, исполняемая на потоках:

$$pbody2 = for(i,0,m, \; for(j, min(k), max(k), \; \_AddLocks(body(i,j))); \; \_Barrier(bar)).$$

При этом распараллеливается внутренний цикл и используется оператор $\_Barrier$ для обеспечения последовательного исполнения итераций внешнего цикла.

Для этого преобразования можно сформулировать аналогичное утверждение о корректности:

Теорема 2. Программы *Ser* и *Par*2 эквивалентны по результату, если итерации внутреннего цикла в *Ser* коммутативны.

Доказательство. Аналогично теореме 1. Дополнительно необходимо проверить условие *CorrectBarrier*. Это условие выполнено: операторы $\_Barrier$ исполняются во всех потоках, при этом в каждом потоке количество операторов одинаково и равно m.

Таким образом, построенная модель позволяет доказывать корректность преобразований программ. Преобразования определенного вида будут корректными, если выполнены определенные условия, накладываемые на программу. В разделе 3 будут рассмотрены некоторые способы проверки таких условий.

2. Модель программ для графических ускорителей

2.1. Модель программы. В предыдущем разделе построена алгебродинамическая модель исполнения многопоточной программы. В данном разделе построим аналогичную модель для другой программно-аппаратной платформы, а именно для программ, которые исполняются на графических ускорителях.

Программирование графических ускорителей для решения задач, не связанных непосредственно с обработкой графики, активно развивается в настоящее время. Однако на данный момент не существует единой общепризнанной платформы для таких вычислений. Кроме того, различные платформы существенно отличаются по используемым подходам и парадигмам организации вычислений. Поэтому, в отличие от многопоточных программ, необходимо выбрать конкретную платформу, для которой и будет построена модель вычислений. Как и в работе [5], будем использовать платформу CUDA корпорации NVidia [17].

Особенностью программ для графических ускорителей является их разделение на две части: код, который исполняется на общецелевом процессоре (CPU) и специализированный код для графического ускорителя (GPU). Эти части могут быть реализованы на разных языках. Например, в [5] рассматривались программы, в которых CPU-код был реализован на C#, тогда как GPU-код – на C for CUDA, специальном расширении языка C.

В модели программы эту особенность будем учитывать следующим образом. По-прежнему будем рассматривать программу как набор компонент; однако теперь каждая компонента относится либо к CPU-, либо к GPU-коду. При этом для CPU и GPU наборы базовых операторов могут различаться.

2.2. CPU-программа. При построении модели исполнения программ для графических ускорителей будем отдельно рассматривать исполнение на CPU и на GPU. Для каждого режима построим соответствующую модель в виде ДДС. Модель исполнения программы в целом, S^{gc}, строится как объединение этих двух ДДС.

Код, относящийся к CPU, исполняется так же, как и любая обычная программа. Поэтому для моделирования этой части можно использовать одну из ранее разработанных моделей S^{ser} или S^{mt}. В данной работе будем использовать модель

S^{ser}. Это значит, что CPU-часть программы выполняется последовательно, и не использует средства многопоточности.

В модель S^{ser} потребуется добавить средства взаимодействия с GPU. В данной модели рассмотрим три новых оператора: $copy\_gpu(m_G, m_C)$ – копирование данных в видеопамять, $copy\_back(m_C, m_G)$ – копирование результатов из видеопамяти, и $call\_gpu(P_i, block, grid)$ – запуск кода (ядра CUDA) на GPU. Соответственно потребуется добавить правила переходов, которые будут менять состояние ДДС S^{gc} в целом, а не только один из ее компонентов. Формальное описание этих правил приведено в п. 2.5, после рассмотрения GPU-компонента.

2.3. GPU-программа: уровень блоков. Модель S^{gpu} исполнения кода на GPU строится по тому же общему принципу, что и многопоточная модель S^{mt}. Строится многоуровневая ДДС, на низшем уровне которой моделируется исполнение отдельных потоков. Для каждого потока используется модифицированная ДДС S^{ser}, и переходы ДДС отдельных потоков объединяются в переход ДДС более высокого уровня, например, с использованием функции *merge*.

Первым важным отличием модели для графических ускорителей от многопоточной модели S^{mt} заключается в большей сложности, которая требует использования большего количества уровней. Модель S^{mt} содержит два уровня: отдельные потоки и программу в целом. Модель S^{gpu} содержит один дополнительный уровень, а именно уровень блоков. Отдельные потоки (первый уровень) объединяются в блоки (второй уровень), которые в свою очередь формируют GPU-программу в целом (третий уровень).

Вторым отличием является наличие иерархии памяти. В CUDA используется 5 видов памяти: регистры, разделяемая (shared) память, кэш констант, кэш текстур и глобальная память. В рамках модели ограничимся рассмотрением двух наиболее часто используемых видов памяти, а именно shared память и глобальная память.

Различные виды памяти в модели проявляются в виде дополнительных компонентов состояния. Для низшего уровня (потоков) состояние имеет вид $s = (b_g, b_s, R, F)$, где $b_g \in B_g$ – состояние глобальной памяти, $b_s \in B_s$ – состояние shared памяти. Однако для упрощения модели будем объединять состояния различных видов памяти в общее состояние $b = b_g \cup b_s$, $b \in B = B_g \times B_s$. Будем считать, что все операторы и предикаты АГ также действуют на объединенном множестве B.

На уровне потоков действуют правила перехода 1) – 5) (правило 4) имеет дополнительное ограничение – вызываемая функция должна работать на GPU, что описывается модификатором `__device__` в C for CUDA).

На уровне блоков, аналогично системе S^{mt}, состоянием является множество потоков, каждый со своим состоянием $s^2 = \{T_i \to s_i^1\}$. В отличие от S^{mt}, количество потоков всегда одинаково и определяется параметрами блока. Поэтому нет необходимости в правиле 12) (завершение потока).

Переходы на уровне блоков объединяются из переходов на уровне потоков, как и в S^{mt}. Функция *merge* применяется к обоим видам памяти b_g, b_s (или к объединенной памяти b). Отличие от многопоточной модели заключается в дополнительном ограничении на переходы: все одновременно срабатывающие потоки должны исполнять одинаковые операторы (поскольку в архитектуре GPU на каждый блок выделяется только одно управляющее устройство).

Средства работы с потоками, содержащиеся в S^{mt}, в случае GPU неприменимы. Вместо них используется единственный оператор *_Barrier*, поведение которого аналогично соответствующему патерну для многопоточной модели. В C for CUDA этот оператор реализован примитивом `__syncthreads()`.

2.4. GPU-программа: уровень программы. Уровень программы в целом строится из уровня блоков таким же обра-

зом, как уровень блоков строился из уровня потоков.

Состояниями программы являются отображения из множества блоков в множество состояний уровня блоков: $s^3 = \{B_j \to s_j^2\}$. Как и на втором уровне, количество блоков постоянно и определяется параметрами запуска ядра.

В отличие от уровня блоков, shared память является уникальной для каждого блока, а не общей, как глобальная память (или общая память в модели S^{mt}). Поэтому объединение результатов функцией *merge* на данном уровне используется только для глобальной памяти b_g.

Еще одной особенностью объединенных переходов на уровне всей программы является следующее ограничение: не может начаться исполнение нового блока (который находится в начальном состоянии уровня блоков), если существует хотя бы один блок, который уже начал исполняться, но не задействован в данном переходе. Иными словами, не происходит переключение между разными одновременно исполняемыми блоками; каждый блок, начавший исполняться, исполняется до конца.

На уровне программы существуют свои средства синхронизации (например, атомарные операции). Однако они доступны не во всех устройствах, сильно замедляют исполнение программы, к тому же противоречат идеологии CUDA, согласно которой блоки должны исполняться независимо и в произвольном порядке. Поэтому не будем включать эти средства в модель.

2.5. Взаимодействие CPU и GPU. Совместная работа двух частей программы координируется из CPU-кода, с использованием операторов *copy_gpu*, *copy_back* и *call_gpu* . Первые два оператора копируют данные между памятью, доступной CPU, и глобальной памятью GPU. Формально это сводится к тому, что некоторое множество переменных (обычно массив) в состоянии одного устройства принимают те же значения, что соответствующие переменные в состоянии другого.

Оператор *call_gpu* собственно запускает код для исполнения на графическом ускорителе. Его работа описывается двумя правилами:

13) $(s^c, (b_g, \varnothing)) \to (s^c, s_0^g(P_i, block, grid))$;

14) $(s^c, s_f^g) \to ((b, R, F), (b_g', \varnothing))$

$s^c = (b, call\_gpu(P_i, block, grid)R, F)$.

Правило 13) описывает создание начального состояния ДДС S^{gpu} при исполнении оператора *call_gpu*. Параметры этого состояния – количество блоков, потоков, исполняемый модуль – извлекается из параметров оператора. Отметим, что оператор не удаляется из состояния управления: с точки зрения CPU, этот оператор выполняется все время, пока идут вычисления на GPU. Правило 14) срабатывает, когда вычисления на GPU окончены. Оно очищает состояние управления GPU, и завершает выполнение оператора *call_gpu*. Состояние памяти CPU в процессе вычислений на GPU не меняется: требуется явное копирование результатов вычислений оператором *copy_back*.

2.6. Свойства GPU-программ. К программам для графических ускорителей применимы те же свойства, что и для многопоточных программ. Однако некоторые из этих свойств имеют определенные особенности для GPU-программ.

Свойство эквивалентности по операторам требует уточнения, поскольку в CPU- и GPU-программах используются различные наборы базовых операторов, которые действуют на различных множествах переменных. Поэтому для определения эквивалентности будем отождествлять те переменные из памяти CPU и глобальной памяти GPU, между которыми производилось копирование с помощью операторов *copy_gpu* или *copy_back*. Также будем отождествлять одинаковые по смыслу операторы, которые действуют на этих множествах.

Определение бесконфликтности остается прежним. Особенность GPU-программ заключается в отсутствии эффективных средств обеспечения бесконфликтности, поэтому это свойство должно быть обеспечено в самой программе.

Из-за ограниченного набора средств синхронизации, для проверки беступиковости GPU-программ применимо единственное условие *CorrectBarrier*. При этом оно упрощается, поскольку используется всегда правильное количество потоков. Таким образом, это условие сводится к тому, что в каждом потоке оператор *_Barrier* используется одинаковое количество раз.

2.7. Преобразования GPU-программ. В качестве примера использования построенной модели для графических ускорителей, рассмотрим преобразования той же последовательной программы, которая рассматривалась в п. 1.7. Построим преобразования последовательной программы в программу для GPU, для обоих случаев (когда возможно распараллеливание внешнего и внутреннего цикла).

В первом случае, преобразованная программа имеет следующий вид:

*Gpu*1=*copy_gpu*;
call_gpu(*gbody*1,*block*2*d*,*grid*2*d*);
copy_back.

Производится копирование необходимых данных в видеопамять, затем вызов ядра *gbody*1, затем копирование результатов. Ядро запускается в двумерном режиме, измерения которого соответствуют двум исходным циклам. Ядро *gbody*1 имеет вид

*gbody*1 =*assign*(*i*,*_GetCoor*(*x*));
assign(*j*,*_GetCoor*(*y*));
_CpuToGpu(*body*(*i*,*j*)).

Сначала вычисляются номера исходных итераций *i* и *j*, для этого используются параметры текущего потока. После этого исполняется тело внутреннего цикла для этих значений. При этом используется функция *_CpuToGpu* для преобразования между операторами CPU- и GPU-программ. Отметим, что в данном случае выделяется по одному потоку на каждую итерацию, в отличие от многопоточной программы, где использовалось ограниченное количество потоков, на каждом из которых выполнялось большое количество итераций цикла.

Для второго случая, т.е. распараллеливание внутреннего цикла, программа имеет следующий вид:

*Gpu*2=*copy_gpu*;
for(*i*,0,*m*,
call_gpu(*gbody*2,*block*1*d*,*grid*1*d*));
copy_back.

Здесь вызов ядра *gbody*2 происходит в цикле. При этом копирование данных между обычной памятью и видеопамятью происходит только один раз. Ядро *gbody*2 соответствует только внутреннему циклу:

*gbody*2 = *assign*(*j*,*_GetCoor*(*x*));
_CpuToGpu(*body*(*i*,*j*)).

В данном случае нельзя было использовать тот же метод, что в многопоточной программе (однократный вызов ядра и синхронизация между итерациями внешнего цикла). Это объясняется невозможностью синхронизации потоков, принадлежащих к разным блокам.

Для GPU-программ также можно сформулировать утверждения о корректности преобразований.

Теорема 3. Программы *Ser* и *Gpu*1 эквивалентны по результату, если итерации внешнего цикла в *Ser* независимы.

Теорема 4. Программы *Ser* и *Gpu*2 эквивалентны по результату, если итерации внутреннего цикла в *Ser* независимы.

Доказательство проводится аналогично теореме 1. Заметим, что в случае GPU необходимо потребовать более сильное ограничение, а именно независимость итераций цикла. Это ограничение позволит обеспечить бесконфликтность даже при отсутствии критических секций или аналогичных средств.

3. Использование переписывающих правил

3.1. Система Termware. Описанные в разделах 1 и 2 модели предоставляют теоретические средства для проверки корректности преобразований программ. Однако для их эффективного использования необходимы также программные сред-

ства автоматизации. В частности, уже упоминалась сложность проверки условий, которые определяют корректность преобразований. В данной работе для автоматизации проверки условий используется система переписывающих правил Termware, которая также используется для автоматизации самих преобразований [3–5].

Termware предназначена для описания преобразования над термами, т.е. выражениями вида $f(t_1, \ldots, t_n)$. Для задания преобразований используются правила Termware, т.е. конструкции вида `source [condition] -> destination [action].`

Здесь `source` – исходный терм (образец для поиска), `condition` – условие применения правила, `destination` – преобразованный терм, `action` – дополнительное действие при срабатывании правила. Каждый из 4 компонентов правила может содержать переменные (которые записываются в виде `$var`), что обеспечивает общность правил. Компоненты `condition` и `action` являются необязательными. Они могут исполнять произвольный процедурный код, в частности использовать дополнительные данные о программе.

3.2. Переход между представлениями программы. Одним из средств, предоставляемых системой Termware, является переход от исходного кода программы, например, на языке C#, к представлению программы в виде терма (соответствующему дереву синтаксического разбора). Такое представление, по сути, является моделью программы, хотя и низкоуровневой (поскольку каждая синтаксическая конструкция языка программирования явно представлена в модели). Подобные модели использовались для описания преобразований программ в [5].

Однако в данной работе, как и в [3–4], используется более высокоуровневое представление программы, в виде операторов алгебры Глушкова. Такое представление является более компактным, к тому же оно удобнее для теоретического анализа. С другой стороны, оно требует специальных средств для получения модели и преобразования ее в код программы. Таким средством может быть, например, инструментарий ИПС [3].

В данной работе рассмотрим другой способ преобразования, с использованием средств Termware. Для некоторой предметной области можно сформулировать список операторов и предикатов АГ, и соответствие между ними и реализующим их кодом. Это соответствие можно сформулировать таким образом, что появится возможность использовать переписывающие правила Termware для автоматического перехода между двумя уровнями представления программы.

Для этого будем использовать паттерны Termware. В общем случае, паттерн определяется двумя системами правил: R_p – система правил для выделения паттерна из произвольного терма, R_g – система правил для расшифровки паттерна. В более частном случае паттерн задается парой термов t_p – обозначение паттерна (элемент модели высокого уровня) и t_g – образец, задающий паттерн (элемент модели низкого уровня). В этом случае $R_p = \{t_g \to t_p\}$ и $R_g = \{t_p \to t_g\}$.

Примером такого задания паттернов могут быть *_Join* и *_Barrier*, определенные в п. 1.6. В этом случае в качестве t_p выступают сами операторы *_Join* или *_Barrier*, а в качестве t_g – их реализация в виде комбинированного оператора.

Если для некоторой предметной области заданы операторы и предикаты высокоуровневой модели в виде паттернов из операторов низкоуровневой модели, становится возможна разработка программы на любом из уровней: высокоуровневая модель (алгебра алгоритмов), низкоуровневая модель (дерево синтаксического разбора), исходный код. Например, для уже существующего приложения можно применить парсер Termware для перехода к дереву синтаксического разбора. После этого используются паттерны и соответствующие правила R_p для перехода к алгебро-алгоритмическому представлению. В этом представлении возможно осуществление некоторых преобразований, в т.ч. проверка корректности с использованием описанных в разделах 1 и 2 моделей.

Далее используются правила R_g для перехода от высокоуровневой к низкоуровневой модели, и затем генератор Termware для воссоздания исходного кода. При этом возможно использование дополнительных преобразований на уровне дерева синтаксического разбора, а также непосредственное внесение изменений в исходный код, для тех случаев, когда не реализованы соответствующие преобразования. Это позволяет повысить удобство разработки, поскольку для реализации каждого преобразования можно использовать наиболее подходящие средства.

3.3. Проверка условий по модели программы. В п. 1.6 приведены примеры условий, которые используются при доказательстве корректности преобразований. Для проверки некоторых из этих условий можно использовать переписывающие правила. В этом случае Termware используется не для преобразования одной программы (модели программы) в другую, а для вычисления некоторых свойств программы.

В качестве примера рассмотрим систему правил для проверки условия *NoNestedCS:*

1. *Method*(*$head*,*$body*) -> *Method* (*$head*,[*m*0*:$body*],*_Mark*).
2. [*m*0*:lock*(*$cs*)*:$x*] -> [*lock(*$cs*): m*1(*$cs*)*:$x*].
3. [*m*1(*$cs*)*:unlock*(*$cs*)*:$x*] -> -> [*unlock*(*$cs*)*: m*0*:$x*].
4. [*m*1(*$cs*)*:lock*(*$cs*1)*:$x*] -> ->*NIL* [*error*()].
5. [*m0:$x:$y*] -> [*$x: m0:$y*].
6. [*m*1(*$cs*)*:$x:$y*] -> -> [*$x: m*1(*$cs*)*:$y*].

Здесь правило 1 добавляет в начало тела каждого метода специальный термаркер *m*0, который используется для обхода всех операторов метода. Правило 2 запоминает идентификатор критической секции, заменяя маркер *m*0 на *m*1. Правило 3 соответствует правильному варианту использования, когда внутри критической секции нет других критических секций. Правило 4 срабатывает при невыполнении условия *NoNestedCS;* в этом случае система правил сообщает об ошибке. Правила 5 и 6 носят служебный характер и обеспечивают продвижение маркеров *m*0 и *m*1 по операторам.

Следует отметить, что невыполнение условия *NoNestedCS* еще не свидетельствует об ошибке: в этом случае необходимо проверять более общие условия, например, *NoCyclesCS.*

3.4. Проверка условий с помощью тестов. Для многих условий их проверка по модели программы является очень сложной. Примером может служить *CommuteWhile* – условие, позволяющее выполнять итерации цикла в произвольном порядке, что необходимо для распараллеливания.

Для проверки этого условия, вместо статического анализа кода программы (или соответствующей модели), можно использовать результаты исполнения кода. При этом генерируются специальные тестовые программы, которые запускают необходимый код, получают результаты его исполнения и сравнивают эти результаты с ожидаемыми. Ожидаемые результаты можно получить путем исполнения исходной программы (которая предполагается правильной).

В простейшем случае в качестве тестов можно использовать преобразованную программу (полностью или частично – только те методы, которые были изменены). При этом к программе необходимо добавить проверку результатов, что реализуется простыми правилами Termware.

Преимуществом такого подхода является то, что проверяется исполнение именно той программы, которая является результатом преобразования. Недостаток может заключаться в том, что некоторые ошибки в параллельных программах могут проявляться только при специальных условиях. Поэтому вполне возможно, что при исполнении тестов результаты совпадут с ожидаемыми, тогда как при реальном использовании программы будут возникать ошибки.

Чтобы уменьшить вероятность подобной ситуации, можно использовать до-

полнительные тесты, которые используют специальным образом преобразованный код. Например, для проверки условия *CommuteWhile* можно сгенерировать последовательную программу, содержащую тот же цикл, но с обращенным порядком итераций. Если в исходной программе итерации были перестановочны, то такая модифицированная программа должна давать тот же результат. В противном случае результат может как отличаться, так и совпадать (если при конкретном способе перестановки операторов случайно получается такой же результат). Можно использовать и более сложные перестановки, например, развернуть цикл на 2 итерации и перемешать операторы из этих итераций.

Особенность тестов такого вида заключается в том, что если тест не проходит, это свидетельствует о невыполнении условия. Если же тест проходит, это не может гарантировать выполнение свойства. Поэтому необходимо использовать несколько различных тестов. Преимущество Termware заключается в том, что переписывающие правила позволяют автоматизировать создание большого количества тестов для каждой конкретной программы.

3.5. Пример использования. Рассмотрим пример использования описанного подхода на конкретной задаче – программе для реализации игры «Жизнь» [18]. Задача была изначально реализована в виде последовательной программы на языке C#. Далее к ней были применены преобразования для перехода к параллельным реализациям: многопоточной и для графического ускорителя.

Исходная последовательная программа соответствует общей схеме, рассмотренной в п. 1.7. Внешний цикл соответствует последовательному проведению итераций игры, тогда как внутренний цикл является обходом всех ячеек в пределах одной итерации. В этом случае реализуется второй из рассмотренных случаев: возможно распараллеливание по внутреннему циклу, тогда как итерации внешнего цикла могут выполняться только последовательно. Для проверки данного условия были сгенерированы тесты с использованием перестановки итераций, которые были описаны в п. 3.4.

Для случая многопоточной программы было применено преобразование *Par*2, описанное в п. 1.7. Преобразование применялось на уровне алгебро-алгоритмического представления программы, поэтому использовались переписывающие правила для перехода между низкоуровневым и высокоуровневым представлением программы. После преобразования программы в многопоточную к ней были применены некоторые оптимизирующие преобразования, описанные в работе [4] (в частности, использование локальных копий общих данных).

Была измерена производительность преобразованных программ, для оценки эффективности преобразований. Измерялось время выполнения 500 итераций, для поля 512x512. Результаты измерений приведены в таблице.

Как видно из результатов, исходное распараллеливание оказалось неэффективным, и полученная программа оказалась даже менее производительной, чем последовательная версия. Это объясняется неэффективным использованием средств синхронизации (более подробно данный вопрос рассмотрен в [4]). Однако применение оптимизирующих преобразований позволило достичь вполне заметного ускорения.

Таблица. Результаты измерений

Вариант	Время исполнения, с
Последовательный	9,8
Многопоточный	27,8
Оптимизированный	3,4

Реализация преобразования той же последовательной программы в программу для графического ускорителя описана в работе [19]. Распараллеливающие и оптимизирующие преобразования позволили достичь ускорения в 25 раз. При этом использование высокоуровневой модели позволило достаточно кратко описать все использованные преобразования, а также скрыть от разработчика различие между конкретными языками реализации последовательной программы (C#) и GPU-части параллельной программы (C for CUDA).

Выводы

В работе предложен метод доказательства корректности распараллеливающих преобразований, основанный на применении алгебро-динамических моделей и системы переписывающих правил Termware. Построены модели исполнения многопоточных программ и программ для графических ускорителей. Для этих моделей сформулированы свойства программ, и предложен метод проверки этих свойств с использованием переписывающих правил. Предложенный метод позволяет автоматизировать проверку корректности для многих оптимизирующих преобразований.

Дальнейшие исследования в данном направлении предполагают формулирование дополнительных условий, обеспечивающих корректность преобразований, а также реализацию переписывающих правил для проверки данных условий. Также возможна интеграция с системами для автоматического доказательства теорем, с целью более полной автоматизации процесса доказательства корректности.

1. *Akhter S., Roberts J.* Multi-Core Programming. Increasing Performance through Software Multi-threading. – Intel Press, 2006. – 336 p.
2. *General-Purpose* Computation Using Graphics Hardware. http://www.gpgpu. org .
3. *Дорошенко А.Е., Жереб К.А., Яценко Е.А.* Формализованное проектирование эффективных многопоточных программ // Проблеми програмування. – 2007. – № 1. – С. 17–30.
4. *Дорошенко А.Е., Жереб К.А., Яценко Е.А.* Об оценке сложности и координации вычислений в многопоточных программах // Проблеми програмування. – 2007. – № 2. – С. 41–55.
5. *Дорошенко А.Е., Жереб К.А.* Разработка высокопараллельных приложений для графических ускорителей с использованием переписывающих правил // Проблеми програмування. – 2009. – № 3. – С. 3–18.
6. *Doroshenko A., Shevchenko R.* A Rewriting Framework for Rule-Based Programming Dynamic Applications, Fundamenta Informaticae. – 2006. – Vol. 72, N 1–3. –P. 95–108.
7. *TermWare*. – http://www.gradsoft.com.ua/products/termware_rus.html .
8. *Дорошенко А.Е., Шевченко Р.С.* Система символьных вычислений для программирования динамических приложений // Проблеми програмування. – 2005. – № 4. – С. 718–727.
9. *Андон Ф.И., Дорошенко А.Е., Цейтлин Г.Е., Яценко Е.А.* Алгеброалгоритмические модели и методы параллельного программирования. – Киев: Академпериодика, 2007. – 631 с.
10. *Kundu S., Tatlock Z., and Lerner S.* Proving optimizations correct using parameterized program equivalence. In Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation (Dublin, Ireland, June 15–21, 2009). PLDI '09. – P. 327–337.
11. *Lerner S., Millstein T., and Chambers C.* Automatically proving the correctness of compiler optimizations. In Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (San Diego, California, USA, June 09–11, 2003). PLDI '03. – P. 220–231.
12. *Lerner S., Millstein T., Rice E., and Chambers C.* Automated soundness proofs for dataflow analyses and transformations via local rules. SIGPLAN Not. – 2005. – Vol. 40, N 1. – P. 364–377.
13. *Lacey D., Jones N. D., Van Wyk E., and Frederiksen C. C.* Proving correctness of compiler optimizations by temporal logic. In Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (Portland, Oregon, January 16–18, 2002). POPL '02. – P. 283–294.
14. *Leroy X.* Formal certification of a compiler back-end or programming a compiler with a proof assistant. In Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (Charleston, South Carolina, USA, January 11–13, 2006). POPL '06. – P. 42–54.
15. *Farzan A., Chen F., Meseguer J., and Rosu G.* Formal Analysis of Java Programs in JavaFAN. In Int. Conf. on Computer Aided Verification, Boston, Mass., 2004.
16. *Timothy Winkler*, "Programming in OBJ and Maude", in Functional Programming, Concurrency, Simulation and Automated Reasoning, International Lecture Series 1991–1992, Springer-Verlag, McMaster University, Hamilton, Ontario, Canada, 1993. – P. 229–277.
17. *NVidia* CUDA technology. http://www.nvidia.com/cuda .

18. *Conway's* Game of Life http://en.wikipedia.org/wiki/Conway%27s_Game_of_Life.
19. *Дорошенко А.Е., Жереб К.А.* Автоматизация разработки приложений для графических ускорителей с использованием переписывающих правил // Труды 5 Восточно-европейской научно-практической конференции по программной инженерии CEE-SECR 2009. – Москва, 28–29 октября 2009.

Получено 17.12.2009

Об авторах:

Дорошенко Анатолий Ефимович,
доктор физико-математических наук, профессор, заведующий отделом теории компьютерных вычислений Института программных систем НАН Украины,

Жереб Константин Анатольевич,
младший научный сотрудник.

Место работы авторов:

Институт программных систем
НАН Украины,
03680, Киев-187,
Проспект Академика Глушкова, 40.
Тел.: (044) 526 1538.
e-mail: dor@isofts.kiev.ua,
zhereb@gmail.com