Госэкзамен 2017

[1. Замкнутые классы алгебры логики. Теорема Поста о замкнутых классах алгебры логики. Структура доказательства теоремы Поста. Конечная порождаемость замкнутых классов. Решетка замкнутых классов.](#_vbifyrt92ypd)

[2. Полиномиальные формы. Представление функций алгебры логики полиномиальными формами. Теоремы об оценках длины функций алгебры логики в классах поляризованных полиномиальных форм (ППФ) и полиномиальных нормальных форм (ПНФ).](#_4f4b6r2widxo)

[3. Распознавание свойств функций алгебры логики. Алгоритмическая сложность распознавания монотонности и самодвойственности функций алгебры логики, заданных в виде ДНФ и в виде полиномов Жегалкина](#_jbm79o9clj5g)

[4. Раскраски графов, хроматическое число графа. Критерий двуцветности графа. Теоремы о верхней оценке хроматического числа графа.](#_s29tyso7e7mf)

[5. Числа Рамсея. Теоремы о верхней и нижней оценках чисел Рамсея.](#_59eaqifff3e0)

[6. Алгоритмы на графах, поиск в глубину и в ширину. Алгоритм поиска двусвязной компоненты графа. Алгоритм поиска фундаментального множества циклов графа.](#_d3po1hglgdnl)

[7. Основные методы построения новых кодов из известных: выкалывание, расширение, укорачивание, прямые суммы, (u|u+v)-конструкция.](#_wtolpnufypwi)

[8. Коды Хэмминга, Голея, Рида-Маллера и их свойства](#_yadr42kalhuk)

[9. Вероятностные алгоритмы. Вероятностные алгоритмы проверки тождеств для матриц и для многочленов. Оценка вероятности ошибки.](#_sbkgg7i9rpnn)

[10. Квантовые вычисления. Определение квантового компьютера, квантовой схемы и квантовых вычислений. Квантовый алгоритм Гровера для задачи поиска, его сложность.](#_ny4z1whyhrnw)

[11. Нейронные сети. Представимость событий в нейронных сетях.](#_kikib8owwdaf)

[12. Модели самовоспроизведения на однородных структурах, теорема Мура.](#_1no66nue7ovd)

[13. Сети Петри. Моделирование конкурирующих параллельных процессов сетями Петри. Основные свойства сетей Петри. Покрывающее дерево сети Петри. Разрешимость проблем ограниченности, безопасности и покрытия для обыкновенных сетей Петри. Неразрешимость проблемы эквивалентности для обыкновенных сетей Петри.](#_465x69mr44zj)

[14. Стандартные схемы программ. Проблема функциональной эквивалентности для стандартных схем программ. Неразрешимость проблемы функциональной эквивалентности. Логико-термальная эквивалентность стандартных схем программ. Разрешимость проблемы логико-термальной эквивалентности стандартных схем программ.](#_lfr06kluqtl1)

[15. Поколения архитектур компьютеров и парадигмы программирования. Архитектурные особенности современных микропроцессоров. Программно-аппаратная архитектура суперкомпьютеров Ломоносов и Blue Gene/P.](#_1c7ypc9p6e1n)

[16. Последовательная и параллельная сложность алгоритмов, информационный граф и ресурс параллелизма алгоритмов.](#_vzycz4vwz6cr)

[17. Методы организация параллельных вычислений при суперкомпьютерном решении сеточных задач.](#_xefddpz3858k)

[18. Архитектурные особенности графических процессоров, направленные на массивно- параллельные вычисления.](#_vk63whedaenr)

[19. Методы эффективной организации параллельных вычислений на графических процессорах.](#_a4f3zyti26i4)

Информация

[Официальный список вопросов и список литературы](http://master.cmc.msu.ru/files/2_gosexam_MK_DiscAlgoritm.pdf)

Экзамен будет 19 мая.

Состав комиссии: хз .

Соответствие наших вопросов и 618/2: 15=1,16=2,18=3 (про суперкомпы), 1=22 (теорема Поста),13=23,14=24 (Захаров)

[Ответы 618/2](https://docs.google.com/document/d/1pWe2uBlvQRrBZPpVjxNnuRDbOialjxcKRXvKx7xztjQ/edit)

[Ответы 621 группы](https://docs.google.com/document/d/17pSgxP5tPuKvQomThWMwQtIf7aPpThi7zxxtT2m8-2o/) (будут 16,18,19)

## 1. Замкнутые классы алгебры логики. Теорема Поста о замкнутых классах алгебры логики. Структура доказательства теоремы Поста. Конечная порождаемость замкнутых классов. Решетка замкнутых классов.

Функция ¬f(¬x1, … ,¬xn) называется *двойственной* к функции f(x1, … , xn) и обозначаентся f\*(x1, … , xn). Если f\* = f, то функция f называется *самодвойственной.*

Лемма о несамодвойственной функции: Из несамодвойственной функции путём подстановки вместо всех переменных функций x , ¬x, можно получить константу.

Лемма о нелинейной функции: Из нелинейной функции путём подстановки вместо всех переменных функций 0, x, y, можно получить нелинейную функцию от переменных x,y.

**Теорема Поста 1**: система функций из P2 функционально полна ⇔ система содержит:

1) Функцию, не сохраняющую константу 0.

2) Функцию, не сохраняющую константу 1.

3) НЕсамодвойственную функцию

4) НЕмонотонную функцию

5) НЕлинейную функцию

Доказательство ⇒ (прямое): Пусть система функций F из P2 полна (в P2), допустим, что в системе F нет одной из указанных функций, например немонотонной. Тогда все функции в F монотонны, т.к. класс монотонных функций замкнут относительно суперпозиции, то мы не можем получить ни одной немонотонной функции. Потому система F полной в P2 не является. Противоречие с полнотой F. Следовательно, система F содержит все 5 указанных функций.

Доказательство ⇐ (обратное): Пусть F содержит все 5 функций: f1(x1, x2,…,xn) – не сохраняет константу 0, f2(x1, x2,…,xn) – не сохраняет константу 1, f3(x1, x2,…,xn) – несамодвойственная функция, f4(x1, x2,…,xn) – немонотонная функция, f5(x1, x2,…,xn) – нелинейная функция.

Покажем, что суперпозицией функций системы F можно получить полную систему G{xy,¬x}

1. Пусть g(x) = f1(x1, x2,…,xn), тогда g(0) = f(0,…,0) = 1 далее возможны два случая:

А) g(1) = 1. ⇒ g(x) ≡1. Функция h(x) = f2(g1(x),g2(x),…,gn(x)) = f2(1,..,1) = 0. Т.е. h(x) ≡ 0 получили константы 0 и 1.

Б) g(1) =0 ⇒ g(x) = ¬x. По лемме о несамодвойственной функции суперпозицией над {f3,¬x} можно получить одну из констант, например 0. f1(0,…,0) =1 есть другая константа. В обоих случаях получили обе константы.

2. По лемме о немонотонной функции суперпозицией над {f4,0,1} можно получить отрицание х.

3. По лемме о нелинейной функции суперпозицией над {f5,1,¬x} можно получить конъюнкцию.

Функции системы G найдены {x&y, ¬x} суперпозицией функций над системой F. Следовательно, F – есть полная система в P2. чтд.

Предполные классы

**Определение**: Замкнутый класс функций К из Р2 предполон, если класс K не является полным, но для любой функции f не из K система k∨{ f } полна.

**Теорема**: Следующие замкнутые классы функций предполны:

1) Класс T0, не сохраняющий константу 0.

2) Класс T1, не сохраняющий константу 1.

3) Класс M монотонных функций.

4) Класс L линейных функций.

5) Класс S самодвойственных функций.

**Теорема** (перефразировка теоремы Поста в терминах предполных классов.): Система функций F полна ⇔ когда F не содержится целиком ни в одном из пяти предполных классов Т0, Т1, М, L, S.

**Замечание**: Пост описал все замкнутые классы алгебры логики и все их базисы, которые оказались конечными. Число замкнутых классов счетно и множество замкнутых классов образует решетку относительно включения множеств. Эта решетка имеет наибольший класс P2 (он же максимум) и три минимальных элемента 01 = [{x}], 02 = [{1}], 03 = [{ 1 }]. Наименьшего элемента решетка не имеет.

Обозначим через O∞ множество всех булевых функций, которые обладают следующим свойством: все наборы, на которых функция принимает значение 0, имеют общую компоненту. Двойственным образом определяется множество I∞.

**Теорема Поста о замкнутых классах**: Совокупность всех замкнутых классов булевых функций счётна и состоит из следующих классов.

1. Классы, содержащие константы 0 и 1 (P2, L, M, D, K, U, MU, C)
2. Классы, содержащие 0 и не содержащие 1 (L0, M0, T0, D0, K0, U0, C0, Im, MIm (m=2,3,...))
3. Классы, содержащие 1 и не содержащие 0 (L1, M1, T1, D1, K1, U1, C1, Om, MOm (m=2,3,...))
4. Классы, не содержащие 0 и 1 (L01, M01, S01, T01, D01, K01, U01, S, SL, SM, SU, I1m, MI1m, O0m, MO0m (m=2,3,...))

Каждый из перечисленных замкнутых классов порождается конечной системой своих функций.

Структура доказательства:

Из теоремы 1, всякий замкнутый класс булевых функций, отличный от класса P2, целиком содержится в одном из классов S, M, L, T0, T1. 7 теорем, которых здесь нет, описывают все замкнутые классы, которые целиком лежат в классах T0, T1, S, L, D, K.

Остаётся исследовать замкнутые классы F ⊆ M, которые не содержатся в указанных классах. Однако из F ⊆ M и F ⊈ T0 следует, что в класс F входит константа 1. Аналогично из отношений F ⊆ M и F ⊈ T1 следует, что 0 ∈ F. Кроме того, лемма о нелинейной функции позволяет утверждать, что в классе F имеется монотонная нелинейная функция от переменных x, y т.е. x ∨ y или xy.

Пусть, например, (x ∨ y) ∈ F. Поскольку F ⊈ D, в классе F есть функция f не дизъюнкция. Тогда классу F принадлежит функция g = y ∨ f, входящая в множество O∞ \ D. Согласно какой-то там теореме класс [g] может совпадать лишь с одним из классов MO∞, MO0∞. В обоих случаях (x ∨ yz) ∈ [g]. Подстановкой константы 0 получаем из функции x v yz конъюнкцию yz. Таким образом, {0, 1 , x ∨ y, xy} ⊂ F. Однако [0, 1 , x ∨ y, xy] = M. Следовательно, F = M.

Двойственным образом рассматривается случай, когда xy ∈ F.

Теорема доказана.

## 2. Полиномиальные формы. Представление функций алгебры логики полиномиальными формами. Теоремы об оценках длины функций алгебры логики в классах поляризованных полиномиальных форм (ППФ) и полиномиальных нормальных форм (ПНФ).

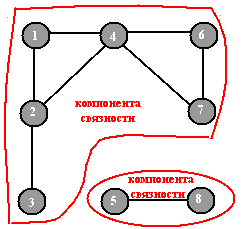
## 3. Распознавание свойств функций алгебры логики. Алгоритмическая сложность распознавания монотонности и самодвойственности функций алгебры логики, заданных в виде ДНФ и в виде полиномов Жегалкина

## 4. Раскраски графов, хроматическое число графа. Критерий двухцветности графа. Теоремы о верхней оценке хроматического числа графа.

## 5. Числа Рамсея. Теоремы о верхней и нижней оценках чисел Рамсея.

## 6. Алгоритмы на графах, поиск в глубину и в ширину. Алгоритм поиска двусвязной компоненты графа. Алгоритм поиска фундаментального множества циклов графа.

*Компонентой связности* неориентированного графа будем называть любой максимальный связный подграф этого графа. Это определение можно переформулировать так: компонента связности — это такой подграф, что для любой вершины u из этого подграфа все вершины, в которые в исходном графе существует путь из u, принадлежат этому же подграфу. На рисунке изображен граф, имеющий две компоненты связности.



Будем называть некоторую вершину неориентированного графа *точкой сочленения*, если при удалении ее и всех инцидентных ей ребер в графе увеличивается количество компонент связности. Эквивалентным определением является следующее: вершина u является точкой сочленения тогда и только тогда, когда в графе существуют две вершины v и w, отличные от u и принадлежащие одной компоненте связности, такие, что любой путь из v в w проходит через u.

Будем называть граф *двусвязным*, если он не содержит точек сочленения. Всякий максимальный двусвязный подграф графа будем называть *двусвязной компонентой*. Другими словами, двусвязная компонента графа — это любой его подграф, в котором удаление произвольной вершины и инцидентных ей ребер не влечет потерю связности этого подграфа, и к этому подграфу нельзя добавить ни одной вершины, сохранив это свойство. На рисунке в графе выделены точки сочленения (вершины 2 и 4) и указаны двусвязные компоненты ({1, 2, 4}, {4, 6, 7}, {2, 3}, {5, 8}).

Заметим, что любые две различные двусвязные компоненты либо не имеют общих вершин, либо имеют одну общую вершину, которая является точкой сочленения.

Сформулируем без доказательства следующую теорему.

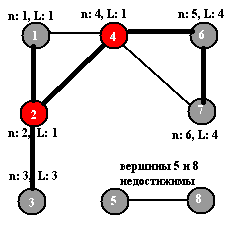
**Теорема.**

Пусть D — [остовное дерево](http://hci.fenster.name/304y/practice/lab7/#spanning) (каркас) неориентированного связного графа G, построенный методом [поиска в глубину](http://hci.fenster.name/304y/practice/lab6/#search) от вершины r. Вершина v является точкой сочленения графа G тогда и только тогда, когда:

* либо v = r и r имеет по крайней мере двух сыновей в дереве D,
* либо v не равно r и у вершины v существует в дереве D такой сын w, что ни w, ни какой-либо из его потомков в дереве D не связаны ребром ни с одним предком вершины v в D.

Изменим процедуру обхода графа в глубину, добавив в нее вычисление двух массивов: int number[N] и int L[N], где N — количество вершин графа. Для каждой вершины v в number[v] будем хранить номер этой вершины в порядке обхода графа в глубину, а в L[v] запишем число number[w], где w — вершина, которая связана обратным ребром с v или каким-либо из потомков v в остовном дереве и имеет минимальный возможный номер number, либо number[v], если такой вершины w найти не удалось. Напомним, что обратными ребрами мы называем ребра, не вошедшие в каркас графа, построенный обходом в глубину.

Разберем подробнее смысл значений L[v]. На рисунке построено остовное дерево графа с корнем в вершине r = 1 и выписаны значения массивов number[N] и L[N]. Для корня остовного дерева L[r] = used[r] = 1. Для вершины 2 значение L[2] равно 1, т.к. из вершины 4 (потомка вершины 2 в остовном дереве) можно вернуться по обратному ребру в вершину 1, для которой number[1] = 1. Аналогично, значения L[6] и L[7] равны 4, потому что из вершины 7 (потомка вершины 6) можно вернуться по обратному ребру в вершину, номер number которой равен четырем.



Зная значения L и number, найти точки сочленения не составляет труда. Согласно приведенной выше теореме, вершина, не являющаяся корнем остовного дерева, будет являться точкой сочленения тогда и только тогда, когда для какого-либо из ее потомков в остовном дереве значение L будет больше либо равно значению number самой вершины. Это означает, что из этого потомка невозможно найти обратное ребро, соединенное с предком рассматриваемой вершины, и по теореме вершина будет являться точкой сочленения. Случай с корнем рассматривается отдельно: корень является точкой сочленения тогда и только тогда, когда у него существует больше одного сына в остовном дереве.

Сформулируем алгоритм поиска точек сочленения более строго. Реализуем рекурсивную функцию void go(int curr, int prev), где curr — текущая вершина, а prev — вершина, из которой мы попали в текущую. При первом вызове curr = r, prev = –1. В теле функции будут выполняться следующие шаги:

1. Запишем в number[curr] номер вершины curr в порядке обхода в глубину.
2. Запишем в L[curr] значение number[curr].
3. Переберем в цикле все вершины, в которые есть ребро из curr. Для каждой такой вершины i выполним следующие действия:
   1. Если вершина i еще не посещена, вызовем рекурсивно функцию go с параметрами i, curr. Если после этого значение L[i] стало меньше, чем L[curr], присвоим L[curr] = L[i].
   2. Если вершина i уже была посещена и ее номер number[i] < number[curr], и при этом i не равно prev (т.е. ребро (i, prev) обратное и возвращается в вершину с меньшим номером), то если L[curr] > number[i], присвоим L[curr] = number[i].

Данный алгоритм заполнит массивы L[N] и number[N] требуемым образом. Проверять, является ли вершина точкой сочленения, можно на шаге 3a. Также, если реализовать стек для хранения ребер, можно реализовать вывод самих двусвязных компонент: ребро нужно добавлять в стек на шагах 3a (перед рекурсивным вызовом) и 3b и выталкивать из стека в поток вывода все ребра вплоть до текущего (curr, i), если на шаге 3а выяснилось, что для текущей вершины curr выполняется условие теоремы.

## 7. Основные методы построения новых кодов из известных: выкалывание, расширение, укорачивание, прямые суммы, (u|u+v)-конструкция.

## 8. Коды Хэмминга, Голея, Рида-Маллера и их свойства

## 9. Вероятностные алгоритмы. Вероятностные алгоритмы проверки тождеств для матриц и для многочленов. Оценка вероятности ошибки.

## 10. Квантовые вычисления. Определение квантового компьютера, квантовой схемы и квантовых вычислений. Квантовый алгоритм Гровера для задачи поиска, его сложность.

## 11. Нейронные сети. Представимость событий в нейронных сетях.

## 12. Модели самовоспроизведения на однородных структурах, теорема Мура.

## 13. Сети Петри. Моделирование конкурирующих параллельных процессов сетями Петри. Основные свойства сетей Петри. Покрывающее дерево сети Петри. Разрешимость проблем ограниченности, безопасности и покрытия для обыкновенных сетей Петри. Неразрешимость проблемы эквивалентности для обыкновенных сетей Петри.

Позиция p в сети Петри π = (P,T, F, W , M) называется **безопасной** , если для любой разметки M0, M0 ∈ R(π) , верно равенство M0 (p) ≤ 1 .

**Живость** перехода t означает, что любое конечное вычисление можно продолжить таким образом, чтобы на некотором шаге сработал переход t .

**Устойчивость** перехода t означает, что если этот переход может сработать, то никакой другой переход не может, сработав, лишить его этой возможности.

**Проблема ограниченности** состоит в том, чтобы для произвольной заданной сети Петри проверить, является ли она ограниченной.

Очевидно, что сеть Петри ограничена тогда и только тогда, когда множество ее достижимых разметок R(π) конечно.

Чтобы убедиться в этом, достаточно построить граф достижимых разметок G(π) и убедиться, что в этом графе конечное число вершин. Однако непросто убедиться в том, что граф G(π) имеет бесконечно много вершин.

**Критерий неограниченности сети Петри**

Сеть Петри π = (P,T, F, W , M') является неограниченной тогда и только тогда, когда существует хотя бы одно такое вычисление M'τ' −→∗ M' τ''−→∗ M'', в котором для пары конфигураций M', M'' выполняется соотношение M' <= M'' , причем хотя бы для одной позиции p справедливо строгое неравенство M'(p) < M''(p) .

Критерий неограниченности сети Петри позволяет построить алгоритм решения проблемы ограниченности сети. Этот алгоритм осуществляет проверку ограниченности сети Петри путем построения т.н. **дерева покрытия разметок** . Вершинами этого дерева служат наборы, состоящие из натуральных чисел, а также специального символа ∞ . Наборы, в которых содержится символ ∞ , будем называть предельными .

**Теорема о дереве покрытия разметок**

1. Для любой сети Петри π дерево покрытия разметок Γ(π) является конечным.

2. Сеть Петри π является ограниченной тогда и только тогда, когда в дереве покрытия разметок Γ(π) отсутствуют предельные вершины.

**Следствие**

1. Проблема ограниченности обыкновенных сетей Петри алгоритмически разрешима.

2. Проблема безопасности ординарных сетей Петри алгоритмически разрешима.

**Проблема R-эквивалентности** состоит в том, чтобы для заданной пары сетей Петри π1 и π2 с одним и тем же множеством позиций P проверить равенство R(π1) = R(π2) .

**Теорема о неразрешимости проблемы R-эквивалентности для сетей Петри.**

Проблема R-эквивалентности для сетей Петри неразрешима.

## 14. Стандартные схемы программ. Проблема функциональной эквивалентности для стандартных схем программ. Неразрешимость проблемы функциональной эквивалентности. Логико-термальная эквивалентность стандартных схем программ. Разрешимость проблемы логико-термальной эквивалентности стандартных схем программ.

**Стандартная схема программ** это конечный размеченный ориентированный граф, вершины которого разделены на пять классов:

1. Начальная вершина . В схеме имеется в точности одна начальная вершина. Она помечена начальным оператором. Из нее исходит одна дуга. Она не имеет входящих дуг.

2. Финальная вершина . В схеме может быть несколько финальных вершин, и каждая из них помечена финальным оператором и не имеет ни одной исходящей дуги.

3. Вершина-преобразователь . Одна помечена оператором присваивания, и из нее исходит в точности одна дуга.

4. Вершина-распознаватель . Она помечена тестом; из нее исходят две дуги, одна из которых помечана символом 0 (0-дуга), а другая символом 1 (1-дуга).

5. Вершина-петля . Она помечена оператором петли и не имеет исходящих дуг.

Стандартные схемы программ π1 и π2 называются **функционально эквивалентными** (обозначается π1 ∼ π2 ), если для любой интерпретации I и для любого начального состояния памяти ξ верно одно из двух:

I оба вычисления comp(π1, I, ξ) и comp(π2, I, ξ) бесконечны (зацикливаются);

I оба вычисления comp(π1, I, ξ) и comp(π2, I, ξ) завершаются, и результаты этих вычислений одинаковы val(π1, I, ξ) = val(π2, I, ξ) .

Проблема функциональной эквивалентности для стандартных схем программ состоит в том, чтобы для произвольной заданной пары стандартных схем программ π1 и π2 выяснить, являются ли эти схемы программ функционально эквивалентными π1 ∼ π2 ?

Стандартная схема программ π называется **тотальной** , если для любой интерпретации I и для любого начального состояния памяти ξ вычисление comp(π, I, ξ) завершается. Проблема тотальности для стандартных схем программ состоит в том, чтобы для произвольной стандартной схемы программ π выяснить, является ли эта схема программ тотальной?

**Утверждение 1.**

**Проблема тотальности стандартных схем программ сводима к проблеме функциональной эквивалентности.**

<http://mk.cs.msu.ru/images/6/62/Lecture_PM_5.pdf> стр 40

**Теорема 1.**

Каковы бы ни были схемы программ π1 и π2 , они **функционально** **эквивалентны** тогда и только тогда, когда функционально эквивалентны соответствующие им размеченные системы переходов LTS(π1) и LTS(π2) .

<http://mk.cs.msu.ru/images/f/f8/Lecture_PM_6.pdf> стр 14

Известно, что проблема пустоты для многоленточных автоматов алгоритмически неразрешима.

Неразрешимость этой проблемы можно доказать построением процедуры, транслирующей каждую машину Тьюринга M в двухголовочный автомат DM , который достигает финального состояния в том и только том случае, если входное слово представляет собой конечную последовательность конфигураций машины Тьюринга M , образующую завершающееся вычисление этой машины на пустой ленте.

Поскольку проблема останова машин Тьюринга на пустой ленте неразрешима, также неразрешима и проблема пустоты для двухголовочных автоматов.

А теперь покажем, как проблема пустоты для двухголовочных автоматов сводима к аналогичной проблеме для стандартных схем программ, работающих в эрбрановских интерпретациях.

**Теорема 3.**

Для любого бинарного двухголовочного конечного автомата D = (S1, S2,s0, F,T) существует такая схема программ πD , которая имеет хотя бы одно конечное завершающееся вычисление в том и только том случае, когда автомат D допускает хотя бы одно входное слово.

**Логико-термальной историей пути** α будем называть чередующуюся последовательность lth(α) = B(v1)θ0, δ1, B(v2)θ1θ0, δ2, . . . , δn, B(vn+1)θn · · · θ2θ1θ0 состоящую из

I примеров B(vi)θi−1 · · · θ1θ0 тех атомов, которые приписаны вершинам пути vi , 1 ≤ i ≤ n + 1 , и специализированы композицией подстановок θi−1 · · · θ1θ0 , помечающих дуги пути α ,

I битов (булевых значений) δi , которые также помечают дуги пути α .

<http://mk.cs.msu.ru/images/6/6d/Lecture_PM_7.pdf>

**Логико-термальной характеристикой** (детерминантом ) стандартной схемы π назовем множество всех ее логико-термальных историй Det(π) = {lth(α) : α терминальный путь в системе переходов для схемы π}

Стандартные схемы программ π1 и π2 назовем **логико-термально эквивалентными** (л-т эквивалентными), обозначая это отношение записью π1 lt∼ π2 , если выполняется равенство Det(π1) = Det(π2) .

Из утверждения 1 и определения л-т эквивалентности следует

Теорема 1.

Для любых схем программ π1 и π2 верно соотношение π1 lt∼ π2 ⇒ π1 ∼ π2.

*Таким образом, проблема проверки л-т эквивалентности схем программ это задача синтаксического анализа программ.*

Пути в системах переходов LTS(π1) и LTS(π2)будем называть **совместными** , если для каждого i, 1 ≤ i ≤ n, справедливо равенство (условие перехода одинаковы).

Чтобы проверить л-т эквивалентность схем программ π1 и π2 ,нужно сравнить л-т истории всех пар совместных путей в системах переходов для этих схем. Поэтому все парысовместных путей целесообразно собрать в одной конечнойструктуре графе совместных вычислений.

**Граф совместных вычислений**

<http://mk.cs.msu.ru/images/6/6d/Lecture_PM_7.pdf> стр 30

Теорема 2.

**Схемы программ π1 и π2 л-т эквивалентны тогда и только**

<http://mk.cs.msu.ru/images/6/6d/Lecture_PM_7.pdf> стр 44

**Алгоритм проверки логико-термальной эквивалентности стандартных схем программ**

<http://mk.cs.msu.ru/images/6/6d/Lecture_PM_7.pdf> стр 78

Таким образом, для проверки л.-т. эквивалентности программ π0 lt∼ π00 достаточно

1. построить граф Γ[π0, π00] совместных вычислений схем программ π0 и π00,

2. применить описанный алгоритм разметки вершин w графа подстановками ηw ,

3. проверить для каждой вершины w = (v0, v00) в графеΓ[π0, π00] выполнимость равенства B0[v0]ηw = B00[v00]ηw ,где ηw это подстановка, вычисленная алгоритмом разметки для вершины w .

## 15. Поколения архитектур компьютеров и парадигмы программирования. Архитектурные особенности современных микропроцессоров. Программно-аппаратная архитектура суперкомпьютеров Ломоносов и Blue Gene/P.

### **Поколения архитектур компьютеров и парадигмы программирования**

### **Векторно-конвейерные компьютеры**

Середина 70-х годов.

Суперкомпьютер Cray-1

**Особенности архитектуры:**

векторные функциональные устройства, зацепление функциональных устройств, векторные команды в системе команд, векторные регистры.

**Программирование:**

векторизация самых внутренних циклов.

### **Векторно-параллельные компьютеры**

Начало 80-х годов.

Суперкомпьютер Cray X-MP, Суперкомпьютер Cray Y-MP

**Особенности архитектуры:**

векторные функциональные устройства, зацепление функциональных устройств, векторные команды в системе команд, векторные регистры. Небольшое число процессоров объединяются над общей памятью.

**Программирование:**

векторизация самых внутренних циклов и распараллеливание на внешнем уровне, единое адресное пространство, локальные и глобальные переменные.

### **Массивно-параллельные компьютеры**

Начало 90-х годов.

Суперкомпьютер Cray T3D, Суперкомпьютер Intel Paragon XPS140

**Особенности архитектуры:**

тысячи процессоров объединяются с помощью коммуникационной сети по некоторой топологии, распределенная память.

**Программирование:**

обмен сообщениями, отсутствие единого адресного пространства, PVM, Message Passing Interface. Необходимость выделения массового параллелизма, явного распределения данных и согласования параллелизма с распределением

### **Параллельные компьютеры с общей памятью**

Середина 90-х годов.

DEC AlphaServer, Суперкомпьютер Sun StarFire

**Особенности архитектуры**:

сотни процессоров объединяются над общей памятью.

**Программирование:**

единое адресное пространство, локальные и глобальные переменные, Linda, OpenMP.

### **Кластеры из узлов с общей памятью**

Начало 2000-х.

Суперкомпьютер МГУ “Чебышев”, “K” суперкомпьютер

**Особенности архитектуры:**

большое число многопроцессорных узлов объединяются вместе с помощью коммуникационной сети по некоторой топологии, распределенная память; в рамках каждого узла несколько (многоядерных) процессоров объединяются над общей памятью.

**Программирование:**

неоднородная схема MPI+OpenMP; необходимость выделения массового параллелизма, явное распределение данных, обмен сообщениями на внешнем уровне; распараллеливание в едином адресном пространстве, локальные и глобальные переменные на уровне узла с общей памятью.

### **Кластеры из узлов с общей памятью с ускорителям**и

Середина 2000-х.

Суперкомпьютер МГУ “Ломоносов”, Суперкомпьютер Tianhe-2

**Особенности архитектуры:**

большое число многопроцессорных узлов объединяются вместе с помощью коммуникационной сети по некоторой топологии, распределенная память; в рамках каждого узла несколько (многоядерных) процессоров объединяются над общей памятью; на каждом узле несколько ускорителей (GPU, Phi).

**Программирование:**

MPI+OpenMP+OpenCL/CUDA;

С 1976 года до наших дней:

* 70-е – Векторизация циклов
* 80-е – Распараллеливание циклов (внешних) + Векторизация (внутренних)
* 90-е - MPI
* середина 90-х - OpenMP
* середина 2000-х - MPI+OpenMP
* 2010-е - CUDA, OpenCL, MPI+OpenMP+ускорители (GPU, Xeon Phi) …

### **Архитектурные особенности современных микропроцессоров**

* Многозадачность

Возможность работы в одном из двух режимов: реальном (real) и защищенном (protected). В *реальном* режиме возможно выполнение только одной программы. Адресация оперативной памяти без специальных драйверов ограничивается 1Мб. В *защищенном* (protected) режиме обеспечивается выполнение сразу нескольких программ за счет переключения между задачами («переключение контекста процессора»). Адресация основной памяти расширена до 4 ГБ (в последних МП – до 100 ГБ).

* Поддержка системы виртуальных машин

Дальнейшее развитие принципа многозадачности, возможность моделирования в одном МП работу нескольких компьютеров, управляемых *разными* ОС.

* Конвейерная обработка команд

Одновременное выполнение разных тактов последовательных команд в разных частях МП с непосредственной передачей результатов выполнения из одной части МП в другую. Позволяло достигнуть пятикратного увеличения производительности МП.

* Кэширование данных

Использование высокоскоростного буфера для обмена данными между микропроцессорной памятью (регистрами МП) и основной памятью ЭВМ. В кэш-память заранее копируются те участки памяти, с которыми собирается работать МП. Управление процессом кэширования осуществляется кэш-контроллером и производится *параллельно* с работой центрального процессора. Современные ЭВМ имеют иерархически организованную кэш-память (до 3 уровней).

* Суперскалярная архитектура

Наличие в микропроцессоре более 1 конвейера для выполнения команд (параллелизм на уровне инструкций)

* Суперскалярная архитектура с поддержкой внеочередного исполнения команд («динамическое исполнение»)

Наличие в микропроцессоре более 1 конвейера для выполнения команд, а также специальных схем, позволяющих изменить изначальную последовательность выполнения команд (не нарушая смысла алгоритма) с целью параллельной загрузки всех конвейеров.

* Гарвардская архитектура процессора

В кэш-памяти 1 уровня предусмотрено разделение команд и данных, которые хранятся отдельно друг от друга для повышения эффективности обработки

* Расширенный набор инструкций

Новые команды, расширяющие базовый набор инструкций МП, для работы с мультимедийной информацией и одновременной однотипной обработки множественных данных.

* Гибридизация RISC и CISC архитектуры

Преобразование стандартных x86-инструкций в RISC-подобные команды фиксированной длины. Еще не выполненные команды записываются в кэш инструкций в том порядке, в котором они будут подаваться на исполняющие устройства (конвейеры) МП. В кэш-памяти может храниться до 12000 микрокоманд. Перевод инструкций формата x86 в микрокоманды ядра процессора происходит асинхронно с работой основных исполняющих устройств.

* Технология одновременной многопоточности

Эмуляция двух логических исполняющих устройств на одном физическом с целью более эффективно исполнять параллельно запущенные потоки команд (параллелизм на уровне потоков)

* Многоядерные процессоры

Объединение двух или более исполняющих устройств на одной ИС, действующих как единое устройство. Обычно имеют общий кэш и интерфейсную систему для связи с другими устройствами ЭВМ

* Технология автоматического увеличения тактовой частоты процессора

Для обеспечения дополнительной производительности и при условии соблюдения ограничений по мощности, температуре и току, процессор может автоматически «разгоняться», то есть увеличивать рабочую тактовую частоту всех своих ядер.

### **Программно-аппаратная архитектура суперкомпьютеров Ломоносов и Blue Gene/P.**

[Суперкомпьютер "ЛОМОНОСОВ"](http://parallel.ru/cluster/lomonosov.html)

Ломоносов-1 производительность - 1.7 PFLOPS

Ломоносов-2 - 2,962 Пфлопс

Суперкомпьютер «Ломоносов» — первый гибридный суперкомпьютер такого масштаба в России и Восточной Европе. В нём используется 3 вида вычислительных узлов и процессоры с различной архитектурой. Предполагается использовать суперкомпьютер для решения ресурсоёмких вычислительных задач в рамках фундаментальных научных исследований, а также для проведения научной работы в области разработки алгоритмов и программного обеспечения для мощных вычислительных систем.

Суперкомпьютер «Ломоносов», установленный в Московском университете в 2009 году, относится к уникальным системам высшего диапазона производительности. В настоящее время он содержит 6654 вычислительных узла, более 94000 процессорных ядер, обладает пиковой производительностью 1,37 Пфлоп/с. Реальная производительность системы на тесте Linpack равна 674 Тфлоп/с, что позволило ему занять в июне 2011 года 13–ое место в списке Top500 самых мощных компьютеров мира.

Также на узлах суперкомпьютера имеются графические ускорители NVidia Tesla K40M.

Файловая система

На суперкомпьютерах НИВЦ МГУ используются сетевые файловые системы, поэтому на вычислительные узлы не требуется ничего копировать. На суперкомпьютерах “Ломоносов” и “Ломоносов-2” домашние каталоги находятся на отдельной файловой системе и не доступны на вычислительных узлах. На вычислительных узлах вместо домашнего каталога пользователя видится содержимое каталога ~/\_scratch. Поэтому перед запуском программы скопируйте её саму и нужные файлы в каталог ~/\_scratch или в подкаталог. Внимание! Содержимое каталогов ~/\_scratch периодически очищается, поэтому не храните там данные долговременно и копируйте их обратно в домашний каталог.

- На суперкомпьютере “Ломоносов” установлена система управления задачами SLURM.

- может работать как с OpenMP, так и с MPI (Message Passing Interface)

- Параллельная файловая система Lustre,

- Файловая система NFS,

- архитектура сети использует сетевую топологию на основе коммутаторов (Switched fabric). FDR InfiniBand

Описание вычислительного комплекса IBM Blue Gene/P

[Описание вычислительного комплекса IBM Blue Gene/P](http://hpc.cs.msu.su/bgp)

IBM Blue Gene/P — массивно-параллельная вычислительная система, которая состоит из двух стоек, включающих 8192 процессорных ядер (2 x 1024 четырехъядерных вычислительных узлов), с пиковой производительностью 27,9 терафлопс (27,8528 триллионов операций с плавающей точкой в секунду).

Коммуникационные сети:

-трехмерный тор (three-dimensional torus)

-сеть общего назначения, объединяющие все вычислительные узлы; предназначена для операций типа «точка-точка»

-вычислительный узел имеет двунаправленные связи с шестью соседями

-глобальные коллективные операции (global collective)

-коммуникации типа «один-ко-многим» (broadcast-операции и редукция)

-загрузка, мониторинг, диагностика, отладка, доступ к счетчикам производительности гигабитная Ethernet-сеть (4 соединения на стойку)

Чтобы разгрузить процессорное ядро от операций, связанных с передачей сообщений по сети трехмерного тора, используется устройство прямого доступа к памяти (direct memory access, DMA). Кроме уменьшения нагрузки на ядро, этот механизм уменьшает вероятность взаимной блокировки процессов, обменивающихся сообщениями, которая может возникнуть вследствие ошибок программиста.

Окружение Blue Gene/P включает

-фронтэнд (front end node) — система, открытая для доступа по протоколу SSH; служит для доступа пользователей на вычислительный комплекс; вся связь с комплексом осуществляется только через эту машину; предназначена для разработки пользователями программ, компилирования проектов и постановки задач в очередь; работа с ней осуществляется в интерактивном режиме

-сервисный узел (service node) — обеспечивает контроль над системой Blue Gene/P; к этой машине доступа по SSH нет

систему управления высокопроизводительной параллельной файловой системой IBM General Parallel File System (GPFS)

## 16. Последовательная и параллельная сложность алгоритмов, информационный граф и ресурс параллелизма алгоритмов.

[Link](https://algowiki-project.org/ru/%D0%93%D0%BB%D0%BE%D1%81%D1%81%D0%B0%D1%80%D0%B8%D0%B9)

**Последовательная сложность** (serial complexity) алгоритма - число операций, которые нужно выполнить при его последовательном исполнении.

**Параллельная сложность** (parallel complexity) алгоритма - число шагов, за которое можно выполнить данный алгоритм в предположении доступности неограниченного числа необходимых процессоров (функциональных устройств, вычислительных узлов, ядер и т.п.). Параллельная сложность алгоритма понимается как высота канонической [ярусно-параллельной формы](https://algowiki-project.org/ru/%D0%93%D0%BB%D0%BE%D1%81%D1%81%D0%B0%D1%80%D0%B8%D0%B9#.D0.AF.D1.80.D1.83.D1.81.D0.BD.D0.BE-.D0.BF.D0.B0.D1.80.D0.B0.D0.BB.D0.BB.D0.B5.D0.BB.D1.8C.D0.BD.D0.B0.D1.8F_.D1.84.D0.BE.D1.80.D0.BC.D0.B0_.D0.B3.D1.80.D0.B0.D1.84.D0.B0_.D0.B0.D0.BB.D0.B3.D0.BE.D1.80.D0.B8.D1.82.D0.BC.D0.B0).

**Ярусно-параллельная форма (ЯПФ)** (parallel form) - это представление графа алгоритма, в котором:

* все вершины разбиты на перенумерованные подмножества ярусов;
* начальная вершина каждой дуги расположена на ярусе с номером меньшим, чем номер яруса конечной вершины;
* между вершинами, расположенными на одном ярусе, не может быть дуг.

*Высота* ЯПФ - это число ярусов. Ширина яруса - число вершин, расположенных на ярусе. *Ширина* ЯПФ - это максимальная ширина ярусов в ЯПФ.

**Канонической ярусно-параллельной формой** называется ЯПФ, высота которой на единицу больше длины критического пути, а все входные вершины расположены на первом ярусе. Для заданного графа его каноническая ЯПФ единственна. Высота канонической ЯПФ соответствует [параллельной сложности](https://algowiki-project.org/ru/%D0%93%D0%BB%D0%BE%D1%81%D1%81%D0%B0%D1%80%D0%B8%D0%B9#.D0.9F.D0.B0.D1.80.D0.B0.D0.BB.D0.BB.D0.B5.D0.BB.D1.8C.D0.BD.D0.B0.D1.8F_.D1.81.D0.BB.D0.BE.D0.B6.D0.BD.D0.BE.D1.81.D1.82.D1.8C) алгоритма.

**Граф алгоритма** (algorithm graph) - это ориентированный ациклический мультиграф, вершины которого соответствуют операциям алгоритма, а дуги - передаче данных между ними. Вершины графа алгоритма могут соединяться несколькими дугами, в частности когда в качестве разных аргументов одной и той же операции используется одна и та же величина. Граф алгоритма почти всегда является параметризованным графом. В частности, его вид часто зависит от входных данных.

Граф алгоритма используется как удобное представление алгоритма при исследовании его структуры, ***ресурса параллелизма***, а также других свойств. Его можно рассматривать как параметризованную информационную историю. Он сохраняет её информативность, при это обладая компактностью за счёт параметризации. Разработана методика построения графа алгоритма по исходному тексту программ.

*См. также:* [*Граф алгоритма*](https://ru.wikipedia.org/wiki/%D0%93%D1%80%D0%B0%D1%84_%D0%B0%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC%D0%B0)

## 17. Методы организация параллельных вычислений при суперкомпьютерном решении сеточных задач.

## 18. Архитектурные особенности графических процессоров, направленные на массивно- параллельные вычисления.

***GPGPU*** - General-Purpose computing on GPU, вычисления общего вида на GPU

***CUDA - Compute Unified Device Architecture*** - программно-аппаратная архитектура от Nvidia, позволяющая производить вычисления с использованием графических процессоров

*Преимущества CUDA выражаются в соотношениях:*

* Цена - производительность
* Производительность - энергопотребление

Возможности устройства определяются его ***Compute Capability*** - <номер поколения>.<номер модификации> (поколения Tesla -> Fermi -> Kepler -> Maxwell)

(На суперкомпе Ломоносов стоят Fermi)

***Архитектурные особенности графических процессоров, направленные на массивно-параллельные вычисления***:

Сравнение CPU и GPU:

* CPU:
  + мало ядер (2, 4, 8, 16, 24)
  + высокая тактовая 2,66 - 3,6 ГГц
  + физическое ядро это 2 логических ядра (Hyper-Threading)
  + ядра работают независимо, 3 уровня кешей (L1 разделяется на кеш данных и инструкций по 32KB) (L2 - 256 KB) (L3 до 20 MB)
  + обращения в память обрабатываются отдельно для каждого ядра
* GPU:
  + физическое устройство 1-й GPU (для карты Fermi у других числа отличаются):
    - 14 мультипроцессоров (SM), каждый мультипроцессор имеет по:
      * 32 скалярных ядра по 1,5 ГГц
      * 16 SFU (специальные ядра) - интерполяция и трансцендентная математика
      * 128 KB регистров на все ядра, но есть ограничение, сколько доступно регистров одной нити
      * 3 вида кеша (L1): текстурный, глобальный, константный
      * L1 кеш (глобальный) может работать в режиме 16 KB или 48 KB, при этом во втором режиме будет работать “разделяемая память” (доступна только между нитями одного мультипроцессора)
      * 16 x Load/Store - зачем это???
      * графическая часть (в этом вопросе она нам побоку): PolyMorphEngine (графический конвейер), текстурные юниты
      * 2 warp sheduller’a
    - L2 кеш (768 KB) - общий на все мультипроцессоры
    - L3 - отсутствует
    - Все ядра в мультипроцессоре работают синхронно, выполняя один код (код тоже называется ядром) - SIMD - single instruction multiple data, а точнее SIMT - single instruction multiple thread
    - Оперативная память 6 GB имеет *высокую пропускную способность* и *высокую латентность* (задержку)
    - Быстрое переключение контекста для групп нитей (поддержка миллионов виртуальных нитей).
    - Когда нити варпа обращаются к памяти, лучше им это делать так, чтобы они попали в одну кеш линию, а то попадут в разные блоки памяти и тогда им придётся ждать не 1 операцию чтения, а 2 операции чтения для загрузки данных на все ядра варпа.  
      Кеш линия - это выравнивание по 32 байта для варпа.  
      Чтобы попадать в кеш-линии нужно выравнивать данные по 32.  
      В CUDA есть особые функции для выделения и копирования памяти с паддингом.  
      Если к матрице идёт доступ по столбцам а не построчно - то это полное не попадание в кеш-линии и всё будет работать адски медленно, нужно транспонировать матрицу
    - Банки памяти - 32 банка выдают 4 последовательных байта за 2 такта  
      чтение из банков поддерживает операцию broadcast  
      конфликт банков памяти снижает производительность доступа к памяти
  + логическое устройство
    - грид -> блок -> нить - 3-х мерные координаты  
      ориентация нити в гриде осуществляется посредством встроенных переменных
    - нити никогда не мигрируют между блоками или ядрами SM
    - нити блока могут иметь общую память (shared memory), блок всегда располагается на одном мультипроцессоре.  
      все блоки грида загружаются на мультипроцессоры GPU по мере возможности (блоков может быть существенно больше, чем места на мультипроцессорах)
    - есть существенное ограничение на максимальное количество нитей в блоке, есть ограничения и на максимальный размер блока и грида по разным координатам - они большие
    - есть возможность синхронизации нитей внутри одного блока
    - все нити блока разбиваются на варпы - кусочки по 32 нити, внутри варпа все нити выполняют одну и ту же инструкцию, если в программном ядре стоит условие if, то какие-то нити будут просто простаивать.  
      На SM могут одновременно выполняться врапы разных блоков (но не больше 8 разных блоков), но не больше 48 варпов => 1536 нитей на каждый SM. (варпы выступают как контексты для ядер (в SM 32 ядра - это один контекст), контексты могут быстро переключаться, но допускается не больше 48 контекстов, т.е. SM не способен обрабатывать более 1536 нитей)  
      Для хорошей загрузки SM - нужно, иметь правильный размер блока! (например, 512)
    - Виртуально:
      * все нити выполняются параллельно
      * нету аппаратной синхронизации нитей между блоками
      * существует общая память в рамках блока

При программировании на CUDA, код состоит из CPU кода, который:

* настройка GPU (GPU - периферийное устройство, управляемое CPU)
* управление памятью GPU (выделение, освобождение, копирование между CPU и GPU и между GPU1 и GPU2)
* Точечный запуск кода GPU by CPU
* Всякий последовательный код CPU

Хорошо распараллеливаются на GPU задачи, которые:

* Имеют параллелизм по данным
* Одна и та же последовательность вычислений, применяемая к разным данным
* Могут быть разбиты на подзадачи одинаковой сложности (подзадача будет решаться блоком нитей)
* Каждая подзадача может быть выполнена независимо от всех остальных, нет потребности в глобальной синхронизации
* Число арифметических операций велико по сравнению с операциями доступа в память для покрытия латентности памяти вычислениями

Технологии, позволяющие работать быстрее:

* асинхронные операции с GPU (запуск ядра, копирование данных)  
  потоки выполнения (streams) - используются для организации последовательностей из асинхронных операций (следующая операция из потока дождётся завершения предыдущей) (по умолчанию все асинхронные операции попадают в поток “0” - Default Stream)  
  потоки можно синхронизировать (зависать пока определённые потоки не завершатся полностью)
* операции могут закончиться ошибками - обязательно нужно их явно проверять после вызова операции над GPU
* к потокам можно приписывать события (они будут вноситься в интересующий поток выполнения)
* Для компиляции используется отдельный компилятор от nvidia - nvcc
* Память может выделяться статическая (только на чтения для ядер GPU)
* Адреса памяти на GPU свои, но если есть поддержка unifiedAddressing, то адресное пространство GPU встраивается в общее адресное пространство.
* L1 кеш при желании можно отключить (на этапе компиляции) и работать мимо него (полезно при разреженном доступе к памяти)
* Общая память - это по сути кеш управляемый пользователем.
* nvidia предоставляет широкие возможности по профлированию
* можно выделять pinned оперативную память - т.е. память с отключённой виртуализацией, тогда можно будет копировать данные в память GPU напрямую из ОЗУ минуя CPU  
  Также появится возможность обращаться к памяти на ОЗУ напрямую из нитей (только нужно будет сконвертировать указатель на память в ОЗУ, т.к. указатели для CPU и GPU будут не совпадать, если только не включен UVA - unified virtual address - unified Addressing)  
  Большое количество pinned памяти деградирует систему в целом, т.к. частично отключает виртуализацию.

## 19. Методы эффективной организации параллельных вычислений на графических процессорах.

По умолчанию все cuda задачи кладутся в поток выполнения “0”, программные ядра всегда выполняются асинхронно.

memcpyAsynс в случае копирования host <-> device

* для pinned memory - управление сразу возвращается в хост
* для pageable (не pinned) - управление возвращается в хост после копирования в буфер

Возможные случаи параллелизма:

* Параллельные копирование и выполнение ядра
* Параллельные выполнение ядер
  + При запуске двух ядер в разных потоках они могут выполняться параллельно только на границах, когда последняя волна блоков (хвост) первого из запущенных ядер не полностью загружает устройство  
    Вывод: При разбиении задачи на подзадачи выигрыш можем получить только за счет:
    - Параллельного выполнения ядер и копирований
    - Параллельного выполнения копирований
* Параллельные копирования с хоста на устройство и с устройства на хост

Если отсутствует возможность параллельного копирования памяти в обе стороны, то неправильный порядок отправки команд может привести к простою.

В случае единственной аппаратной очереди (Compute Capability <= 3.0) следующая операция может начаться только после того, как завершится предыдущая - поэтому, если загружать команды как: KERN - MemCpy - KERN, то второй KERN будет ждать пока завершится MemCpy и нельзя будет выиграть на параллельном выполнении KERN, а если переставить на KERN - KERN - MemCpy - то MemCpy от первого ядра запустится лишь после второго KERN, хотя могла бы выполняться параллельно со вторым KERN - так что так тоже плохо, какой случай выбрать зависит от того, какой размер KERN и какой размер памяти для копирования

MPI + OpenMP + CUDA - хорошо гармонируют, так как MPI - ускорение экстенсивным путём, между процессорами, OpenMP - простое ускорение экстенсивным путём внутри процессора, а CUDA - ускорение интенсивным путём однотиповых вычислений.

Один CPU поток может управлять несколькими GPU, для этого нужно лишь переключать контексты.

<MultiGPU и OpenMP> или <MultiGPU и MPI> неплохо сочетаются, если каждый поток OpenMP будет сочетаться с своим контекстом GPU (контекст - это отдельная GPU)

Единственный минус от совмещения технологий - тяжелее писать код и забодаешься это компилировать.

Если есть UVA, то peer-to-peer делаются неявно при вызове cudaMemcpy, если не поддерживается, то нужно указывать специальную функцию cudaMemcpyPeer с явным указанием устройств

cudaMemcpyPeer - дождётся завершения работы устройств и начнёнт копирование (блокирующее для обоих устройств)

cudaMemcpyPeerAsync - дождётся завершения работы устройств и начнёт не блокирующее копирование для обоих устройств

cudaMemcpyPeer\* - оба вызова не блокируют CPU

Используется ли процессор в копировании - или идёт копирование через контроллер на PCI шине мимо процессора - это отдельная фича железа, которую нужно включать если есть - cudaDeviceEnablePeerAccess - даёт ускорение в несколько раз.