Министерство науки и высшего образования РФ

Федеральное государственное автономное образовательное учреждение высшего образования «Сибирский федеральный университет»

Кафедра высокопроизводительных вычислений

РЕФЕРАТ

по теме: «История развития языков и моделей программирования для высокопроизводительных вычислений»

Подготовил Горбацевич Андрей Анатольевич

Направление подготовки: 2.3 Информационные технологии и телекоммуникации

Программа: 2.3.1 Системный анализ, управление и обработка информации, статистика

Проверил_____ /_____
(ФИО преподавателя) *(зачтено/не зачтено)*

Отдел аспирантуры
(печать)

Красноярск 2025

СОДЕРЖАНИЕ

Введение

Актуальность темы: развитие технологий тесно связано с ростом объёмов обрабатываемых данных, особенно в сфере искусственного интеллекта. В связи с этим история высокопроизводительных вычислений становится как никогда актуальной. Рост массивов данных, развитие ИИ и необходимость ускорения вычислений делают историю языков и моделей программирования для HPC особенно важной. Эволюция подходов позволяет оптимизировать параллельные вычисления на суперкомпьютерах и ускорителях вроде GPU. Ограничения аппаратного обеспечения и растая сложность задач, требующих распределённой обработки, ведут к появлению новых вычислительных моделей.

Первым высокоуровневым языком для HPC стал Fortran, разработанный IBM в 1954 году под руководством Джона Бэкуса, который обеспечил эффективность на быстро эволюционирующем оборудовании и остаётся стандартом для бенчмарков TOP500[6]. В 1960–1970-х годах последовательное программирование уступило место параллельным подходам из-за роста многоядерных систем[7].

В контексте развития высокопроизводительных вычислений особую роль сыграли параллельные модели и языки программирования. Уже в 1990-е годы сформировались ключевые подходы – **message passing** и **shared memory**[8]. Стандарт MPI (1994 [8]) стал основой для распределённых систем и фактически закрепился как де-факто стандарт параллельных вычислений на кластерах. Параллельно развивался OpenMP (1997 – для Fortran и 1998 – для C/C++), ориентированный на многопоточное программирование для многоядерных CPU.

Следующий значимый этап связан с появлением GPU-программирования. В 2007 году NVIDIA представила CUDA[9], впервые предложив удобную модель для унифицированных вычислений на GPU. Её развитие было дополнено открытым стандартом OpenCL[10] (2008), получившим поддержку NVIDIA и AMD. Параллельно появился Chapel – язык, создававшийся Cray в рамках программы DARPA HPCS (2002–2012) и ориентированный на высокопроизводительную параллельность в масштабах суперкомпьютеров[18]. Эти модели существенно

расширили возможности обработки данных, обеспечив переносимость и адаптивность к гетерогенным вычислительным системам.

В итоге эволюция – от классических решений вроде Fortran до современных GPU-ориентированных моделей CUDA и OpenCL – значительно упростила разработку высокопроизводительных алгоритмов. Смещение парадигм в сторону **параллелизма данных** и **гетерогенных вычислений** определяет современное состояние HPC и формирует требования к моделям программирования будущего.

1 Ранние языки для научных расчётов

1.1 Fortran – первый язык для научных вычислений

Fortran (аббревиатура от **FOR**-mula **TRAN**-slation, «транслятор формул») разработан командой IBM под руководством Джона Бэкуса в 1954–1957 годах. Первая версия (Fortran I) выпущена в 1957 для компьютера IBM 704[11]. Проект начался из-за того, что написание программ в машинных кодах было крайне трудоёмким, как характеризовал этот процесс сам Бэкус: «Рукопашный бой с машиной[11].»

Ключевой проблемой был разрыв между мышлением учёных и инженеров – формулами – и тем, что понимала машина – машинные коды. Fortran был сделан так, что код на этом языке был довольно близок к математической нотации (Листинг 1).

```
Y = A*X**2 + B*X + C
```

Листинг 1. Пример кода на языке Fortran

Такой подход позволил уменьшить как объёмы кода, так и снизить количество ошибок при его написании.

Компиляторы Fortran принесли новые виды оптимизации кода: развёртку циклов, распределение регистров, оптимизация повторяющихся подвыражений и другие. Что особенно важно – генерируемый код был сопоставим по эффективности с ассемблерным кодом, написанным вручную, а иногда даже быстрее.

Таким образом, Fortran стал фундаментом для высокопроизводительных вычислений:

- стандартизация научных расчётов: общий язык для численного программирования;
- библиотеки: BLAS, LINPACK/LAPACK и другие фундаментальные библиотеки написаны на Fortran;
- портируемость: код можно было переносить между разными компьютерами, для которых был доступен компилятор Fortran;

- долголетие: современные версии до сих пор используются в суперкомпьютерных центрах.

При этом разработка и улучшение Fortran не остановилась, отдельные его модули и библиотеки до сих пор используются для высокопроизводительных вычислений, а последняя версия языка была выпущена в 2023 году.

1.2 Алгол, Лисп и компиляторы

Но не Фортраном единым замыкается разработка программ.

1.2.1 Algol

В 1958 году комитетом европейских и американских учёных на съезде в институте ETH в Цюрихе был разработан язык программирования ALGOL (аббревиатура от **ALGO**-rithmic **L**-anguage, «алгоритмический язык»), точнее его вариант ALGOL 58 [12].

В ходе разработки языка были выведены его ключевые концепции:

- блочная структура программ;
- разрешение рекурсии;
- формальное описание синтаксиса в форме BNF-нотации.

Многи последующие императивные языки программирования позаимствовали синтаксис Алгола – но стоит отметить, что сам язык широкого распространения за пределами научных вычислений не получил.

1.2.2 Lisp

В 1985 году Джон Маккарти представил миру Lisp (аббревиатура от **Lis**-t **P**-rocessing) – язык программирования для символьных вычислений и ИИ [26].

Особенностями языка можно назвать:

- «функции первого класса» – их можно передавать, возвращать и проводить композицию;
- каррирование – преобразование функции множества аргументов в набор вложенных функций;

- отсутсвие циклов – их заменили рекурсией;
- универсальная структура данных – список;
- символьное дифференцирование;

Lisp позже повлияли на Python (list comprehensions), JavaScript (функции высшего порядка), языки типа Haskell, Scala и другие.

Также этот язык является прародителем функционального программирования, хотя он и является мультипарадигменным.

1.2.3 Оптимизирующая компиляция

Как уже было сказано ранее, компилторы стали применять систематические оптимизации.

Ключевые достижения:

- компилятор Fortran мог генерировать более эффективный ассемблерный код, по сравнению с человеком;
- анализ потока данных [22];
- алгоритмы оптимизации графов (например, «алгоритм Хайтина» для раскраски графов и последующего распределения регистров);

Для HPC оптимизирующие компиляторы стали критическим звеном между абстрактным кодом и эффективным использованием железа. Современные компиляторы (GCC, LLVM, Intel ICC) применяют сотни проходов оптимизации: векторизацию, предсказание переходов, межпроцедурную оптимизацию, полиэдрическую оптимизацию для вложенных циклов, однако база была заложена давно.

2 Векторные- и суперкомпьютеры

2.1 Концепция векторных вычислений

Векторные вычисления представляют собой фундаментальную ступень развития высокопроизводительных вычислительных систем, которая позволила достичь качественного скачка в производительности при решении задач, требующих обработки больших однородных массивов данных.

2.1.1 Суперкомпьютеры Cray и становление векторной обработки

Компания Cray Research, основанная Сеймуром Креем в 1972 году, стала пионером в области векторных суперкомпьютеров. Первая машина серии Cray-1, представленная в 1976 году, продемонстрировала революционный подход к организации вычислений[13]. Архитектура системы была построена на принципе конвейерной обработки векторных данных, что позволяло выполнять одну операцию над множеством элементов данных одновременно. Главная память машины состояла из 16 независимых блоков по 64 тысячи машинных слов каждый, а 12 функциональных модулей позволяли выполнять логические, скалярные и векторные операции над этими данными с высокой степенью параллелизма.

Ключевое преимущество векторной архитектуры заключалось в том, что одна векторная инструкция заменяла целый цикл скалярных операций. Это сокращало количество необходимых к исполнению команд и устраняло накладные расходы на управление циклами. Для научных и инженерных расчетов, где типичными являются операции с матрицами и массивами данных, такой подход обеспечивал многократное ускорение по сравнению со скалярными процессорами того времени.

2.1.2 Модель SIMD как основа векторизации

Концептуальной основой векторных вычислений стала модель SIMD, сформулированная в таксономии Флинна[20]. В своей ранней реализации эта модель предполагала, что один поток команд управляет обработкой множества потоков данных параллельно. Векторный процессор фактически реализовывал

данную парадигму аппаратно: единственная векторная инструкция запускала идентичную операцию над всеми элементами векторного регистра.

Важно отметить, что ранняя форма SIMD в векторных суперкомпьютерах и современные векторные расширения процессоров существенно отличаются. Векторные машины работали с векторами переменной длины и обладали специализированными конвейерами для различных типов операций, а SIMD-расширения современных процессоров оперируют фиксированными по размеру векторами для исполнения этих операций ничего кроме самого процессора не требуется.

2.1.3 Векторные расширения в языке Fortran

Параллельно с развитием аппаратных векторных систем происходила эволюция программных средств, позволяющих эффективно использовать возможности векторной обработки[19]. Язык Fortran в стандарте Fortran 90 получил расширение в виде конструкции для работы с массивами как с едиными объектами, что позволяло программистам того времени выражать векторные операции в знакомой математической нотации.

Компиляторы Fortran получили возможность автоматической векторизации циклов, распознавая паттерны кода, которые могли быть преобразованы в векторные инструкции[15]. Это означало, что существующий код мог быть оптимизирован для векторных архитектур без полного переписывания, хотя наибольшая эффективность достигалась при использовании явных векторных конструкций языка. Появление директив компилятора также позволило программистам явно управлять процессом векторизации, указывая компилятору на возможности параллелизма в коде.

Развитие векторных вычислений заложило основу для современных параллельных архитектур и продолжает оставаться актуальным в контексте современных процессоров с SIMD-расширениями и графических ускорителей, использующих сходные принципы обработки данных.

2.2 Новые компиляторные оптимизации

Параллельно с развитием векторных архитектур происходила модернизация в технологиях компиляции, направленная на максимальное использование возможностей современного оборудования. Компиляторы превращались из простых трансляторов высокоуровневого кода в машинный в сложные системы оптимизации, способные радикально трансформировать программу для достижения максимальной производительности.

Центральной задачей стала векторизация циклов. Компиляторы научились анализировать структуру циклов и определять, можно ли безопасно преобразовать последовательные итерации в единую векторную операцию. Для успешной векторизации цикла компилятору необходимо доказать, что между итерациями отсутствуют зависимости, так как в обратном случае применение векторных оперций приведёт к некорректным результатам вычисления.

Неполное разворачивание циклов стало одной из ключевых техник оптимизации – компилятор мог группировать несколько итераций цикла, что давало ему возможность переупорядочить инструкции и уже затем, возможно, заменить их векторными операциями. Цикл всё ещё присутствует, но он может быть значительно меньше оригинального варианта.

Компиляторы, такие как для TI ASC и Cray Fortran, анализировали циклы на отсутствие зависимостей данных между итерациями, чтобы преобразовать последовательный код в векторные операции[28]. Это обеспечивало безопасность параллелизма без изменения результатов программы. Методы включали решение систем диофантовых уравнений для доказательства независимости.

Программисты использовали директивы (например, Cray VECTOR directives[14]) для подсказок компилятору о безопасной векторизации при неполном статическом анализе. Межпроцедурный анализ учитывал вызовы функций для глобальной векторизации, как в Convex Application Compiler. Эти нововведения легли в основу современных компиляторов.

3 Эра массового параллелизма и распределённых систем

3.1 Переход к архитектуре MIMD

Сдвиг в сторону MIMD[20] означал отход от прежних парадигм векторной обработки и одиночных инструкций. Этот переход был обусловлен как технологической осуществимостью, так и признанием того факта, что многие реальные вычислительные задачи требовали гибкости в том, как различные элементы конвейера обработки могли работать независимо друг от друга.

В этот период начали появляться многоядерные системы, хотя по сравнению с современными реализациями они были еще в зачаточном состоянии. Ранние кластеры собирались из стандартных рабочих станций, объединяемых сетевой инфраструктурой. Оказалось, что не обязательно нужны специализированные монолитные суперкомпьютеры для получения больших вычислительных мощностей. Первые такие кластерные системы заложили основу для современных архитектур центров обработки данных и инфраструктуры облачных вычислений.

3.2 Языки программирования и параллельные вычисления

С расширением возможностей аппаратного обеспечения все более актуальной становилась задача выражения параллельных вычислений в языках программирования. Появилось несколько подходов, каждый из которых отражал различные философские представления о том, как программисты должны концептуализировать параллельное выполнение кода.

3.2.1 Occam и модель CSP

Occam, разработанный в начале 1980-х годов группой учёных из Оксфордского университета, являлся реализацией модели CSP (Communicating Sequential Processes, Теория Взаимодействующих Последовательных Процессов) – сформулированной Тони Хоаром в нескольких работах[23].

В Occam процессы считаются фундаментальными единицами, коммуницирующими друг с другом посредством «передачи сообщений», что предоставляет

сильную теоретическую основу для рассуждений о параллельном поведении программ.

Однако CSP, в разрезе параллельного программирования, повлияла не только на Occam, но и заложила концепции, применяемые в параллельном программировании по сей день: такие как синхронная передача сообщений через каналы и отсутствие общих переменных между процессами. Эти идеи активно применяются в современных языках, включая Go (горутины и каналы), Erlang, Limbo и `core.async` в Clojure.

3.2.2 High Performance Fortran (HPF) и его ограничения

HPF стал попыткой расширения языка программирования Fortran путём добавления директив, позволяющих компиляторам автоматически разделять код для последующего его исполнения на распределённых системах. И хотя в проект были вложены немалые ресурсы, он столкнулся с проблемой чистой компиляторной оптимизации.

Оказалось, что автоматически распределять векторные операции по нескольким хостам – не такая простая задача, что привело к ограниченному использованию HPF, по сравнению с другими, более «ручными» вариантами.

3.2.3 Message Passing Interface (MPI)

В 1994 году появился стандарт MPI, который вместо попыток автоматической параллелизации или создания нового языка программирования предоставлял стандартизированную библиотеку обмена сообщениями между процессами. Такой подход, в противопоставление HPF, предоставлял разработчикам полный прямой контроль над межпроцессорным взаимодействием, оставляя, однако, сложные механизмы передачи сообщений «за кадром» разработки. Такой подход также упростил переносимость кода между платформами. В общем и целом, всё вышеобозначенное привело к большой популярности и всеобщему признанию MPI.

3.2.4 OpenMP

OpenMP появился как дополнение к MPI – в этой библиотеки реализовывался параллелизм через общую память с помощью директив для компилятора, вставляемых в код программы. Такой подход OpenMP снижал сложность внедрения многопоточного выполнения в рамках одной машины. При этом различия между моделью распределённых вычислений MPI и подходом OpenMP позволяло использовать их вместе для ещё большего ускорения вычислений.

3.3 Что считать «естественной» моделью параллелизма?

Распространение подходов к параллельному программированию отражало более глубокие, неразрешённые вопросы о фундаментальной природе параллельных вычислений. Исследователи и практики вели – и продолжают вести – длительные дискуссии о том, следует ли выражать параллелизм в алгоритме или же стоит полагаться на аппаратно-ориентированные подходы.

Сторонники алгоритмического подхода утверждают, что параллельные вычисления представляют собой принципиально иную логику, требующую явного выражения в моделях программирования. Полагается, что программист должен понимать и контролировать стратегии декомпозиции, обмен данными и требования к синхронизации. Эта точка зрения проявляется в системах, подобных MPI, где параллелизм выражается явно, в коде.

Аппаратно-ориентированный подход стремится сделать параллелизм более неявным, перекладывая распараллеливание на «плечи» компилятора и среды исполнения. Эта философия лежала, например, в HPF. Сторонники такого подхода считают, что отделение алгоритмической корректности от производительности параллельного исполнения сделает параллельные вычисления более доступными для широкого круга разработчиков.

Эти философские различия были не просто академическими спорами, но действительно отражали неопределённость относительно того, как будет развиваться параллельная вычислительная парадигма. Эти вопросы – об уровнях абстракции, ответственности программиста и взаимосвязи между алгоритмами

и архитектурой, – продолжают находить отклик в современных обсуждениях проектирования параллельных и распределённых систем.

4 Появление GPGPU и ломка парадигм

4.1 Исторические предпосылки

К началу 2000-х годов графические процессоры превратились в высокоспециализированные устройства, предназначенные исключительно для обработки графики и визуализации. Их архитектура развивалась по траектории, радикально отличавшейся от центральных процессоров: вместо сложных систем предсказания ветвлений и многоуровневых кэшей GPU делали ставку на массовый параллелизм. Типичный графический процессор того времени содержал десятки – затем и сотни – простых вычислительных ядер, способных одновременно обрабатывать множество пикселей или вершин геометрии.

Ключевой концепцией стала модель SIMT (Single Instruction, Multiple Threads), которая представляла собой эволюцию классической SIMD-архитектуры. Ключевым различием между SIMD и SIMT является «строгое исполнение» (lockstep execution) – что означает выполнение одного и того же выражения (инструкции) всеми процессорами одновременно. Эта модель идеально подходила для графических задач, где миллионы пикселей требовали одинаковой обработки с разными входными данными.

Исследовательское сообщество увидело потенциал использования GPU для неграфических вычислений еще в начале 2000-х. Первые работы демонстрировали, что с помощью графических API, такие как OpenGL и DirectX, можно производить произвольные вычислений через манипуляции с текстурами и шейдерными программами. Однако этот подход требовал глубокой переработки алгоритмов вычислительных задач, дабы их можно было «уложить» в графические примитивы, что создавало значительный концептуальный барьер для программистов.

4.2 Появление CUDA

В ноябре 2006 года NVIDIA представила CUDA, которая изменила подход к программированию графических процессоров для неграфических нужд. CUDA обеспечила прямой доступ к вычислительным ресурсам GPU через расширение

языка C, освободив разработчиков от необходимости работать через графические API. Появление CUDA обозначило переход от косвенного использования графического конвейера к явной вычислительной модели[21, 31].

Архитектура CUDA ввела иерархическую модель организации вычислений, которая отражала аппаратную структуру GPU. Программист организовывал вычисления в виде сетки блоков, где каждый блок содержал группу потоков, способных взаимодействовать через быструю разделяемую память[30]. Варп (warp) – группы из 32 потоков, выполняющихся синхронно на одном мультипроцессоре – стал основной единицей выполнения. Эффективное программирование требовало понимания того, как варпы выполняют инструкции и как расхождение потоков ветвлений внутри варпа может снизить производительность.

NVIDIA приняла решение создать закрытую экосистему вокруг CUDA. Технология работала исключительно на GPU их производства, в то же время компания инвестировала значительные ресурсы в развитие библиотек, инструментов разработки и образовательных программ. Этот подход отражал убеждение, что вертикальная интеграция аппаратного обеспечения и программного стека позволит достичь максимальной производительности и ускорить внедрение новых возможностей. NVIDIA развивала богатую экосистему специализированных библиотек, таких как cuBLAS для линейной алгебры и cuFFT для преобразований Фурье, которые обеспечивали оптимизированные реализации распространенных алгоритмов.

Согласно исследованиям начала эры CUDA, производительность GPU для определенных классов задач могла превышать производительность центральных процессоров на порядки величины, что стимулировало стремительное принятие технологии в научных вычислениях и машинном обучении.

4.3 OpenCL как альтернативная модель

В ответ на доминирование CUDA консорциум Khronos Group в 2008 году представил OpenCL (Open Computing Language) как открытый стандарт для гетерогенных параллельных вычислений[29]. OpenCL воплощал в себе переносимость и универсальность, позволяя одному и тому же коду выполняться на GPU

различных производителей, центральных процессорах, и других ускорителях. Стандарт поддерживался коалицией компаний, включая AMD, Intel, Apple и даже NVIDIA.

Архитектура OpenCL строилась вокруг абстрактной модели выполнения, которая не привязывалась к специфике конкретного оборудования. Программисты описывали вычисления в терминах «рабочих групп» и «рабочих элементов», которые затем выполнялись на аппаратном обеспечении драйвером и компилятором. Эта модель обеспечивала гибкость и создавала дополнительные уровни абстракции между программой и железом.

Конкуренция между CUDA и OpenCL отражала противоречие в индустрии высокопроизводительных вычислений. CUDA предлагала более тесную интеграцию с аппаратным обеспечением NVIDIA, что часто приводило к превосходной производительности и более удобным инструментам разработки, а библиотеки CUDA были глубоко оптимизированы под конкретные архитектуры GPU – поэтому компания могла быстро внедрять поддержку новых аппаратных функций. В то же время OpenCL обещала переносимость кода между различными платформами, что было критически важно для организаций, использующих гетерогенную инфраструктуру или стремящихся избежать привязки к одному поставщику.

Практический опыт показал, что достижение сопоставимой производительности на OpenCL часто требовало куда больших усилий по оптимизации под каждую конкретную платформу, что частично нивелировало преимущества переносимости. Тем не менее OpenCL сохранила свою нишу в приложениях, где важна поддержка широкого спектра оборудования, таких как системы компьютерного зрения или графических приложениях, работающих на различных платформах.

5 Развитие высокоуровневых моделей и DSL для HPC

5.1 «Наращивание» абстракций с TensorFlow, PyTorch, MLIR и TVM

Взрывной рост машинного обучения и нейронных сетей ознаменовал трансформацию высокопроизводительных вычислений начиная с около 2010-го года. Традиционная модель разработки, требовавшая от программиста глубокого понимания аппаратной архитектуры и ручной оптимизации каждого аспекта параллельного кода, оказалась неподходящей для новой волны исследователей и инженеров в области ИИ. Это привело к созданию фреймворков, которые не просто упростили программирование, но кардинально изменили саму философию взаимодействия разработчика с вычислительными ресурсами.

Фреймворк TensorFlow, представленный Google в 2015 году, стал одним из первых масштабных проектов, реализовавших концепцию вычислительного графа как высокоуровневой абстракции над гетерогенными вычислительными архитектурами. Система позволяла описывать алгоритмы машинного обучения в терминах операций над тензорами, автоматически распределяя вычисления между центральными процессорами и графическими ускорителями. Ключевым достижением TensorFlow стала реализация автоматического дифференцирования и оптимизация графа вычислений на этапе компиляции[1].

Параллельно развивался PyTorch, выпущенный Facebook AI Research в 2016 году и предложивший альтернативную парадигму динамических вычислительных графов, которые строятся в процессе исполнения программы. Этот подход обеспечил большую гибкость для исследовательских задач и упростил отладку, сохранив при этом способность к эффективному выполнению на GPU через систему автоматической генерации CUDA-кернелов. Архитектура PyTorch продемонстрировала, что высокий уровень абстракции не обязательно требует жертв в производительности, если промежуточные представления спроектированы правильно[4].

Критической проблемой обоих фреймворков стала их привязанность к специфичным доменам и ограниченная переносимость оптимизаций между различными типами операций и аппаратных платформ. Ответом на эту проблему

стало создание MLIR (Multi-Level Intermediate Representation), анонсированного командой Google в 2019 году. MLIR представил иерархическую систему промежуточных представлений, где каждый уровень соответствует определенному уровню абстракции от высокоуровневых операций машинного обучения до низкоуровневых инструкций целевой архитектуры. Фундаментальная идея MLIR заключалась в создании расширяемой инфраструктуры, позволяющей различным доменно-специфичным языкам и компиляторам совместно использовать общие оптимизационные проходы и трансформации[5].

Проект TVM, начатый исследователями из университета Вашингтона в 2017 году, сфокусировался на автоматической оптимизации и генерации кода для разнородных аппаратных платформ. TVM ввел концепцию автоматического планирования через систему template-based code generation с последующим автоматическим тюнингом производительности. Система использует машинное обучение для предсказания оптимальных конфигураций кернелов на основе характеристик операций и целевого оборудования[2].

Роль промежуточных представлений в этой эволюции крайне велика – IR стали не просто техническим артефактом процесса компиляции, но ключевым механизмом, позволяющим разделить спецификацию алгоритма от деталей его реализации на конкретной аппаратной платформе. Многоуровневая структура современных IR позволяет проводить оптимизации на различных уровнях абстракции: от алгебраических упрощений и операторного слияния на верхних уровнях до оптимизации использования памяти и векторизации на нижних. Это создало возможность для композиции оптимизаций, где высокоуровневые трансформации открывают возможности для низкоуровневых, и наоборот.

Автоматическая генерация CUDA и OpenCL кернелов достигла уровня, когда для многих классов операций генерируемый код сравним или превосходит по производительности код, написанный вручную экспертами. Это стало возможным благодаря систематическому исследованию пространства оптимизаций, включающего выбор размера блоков, стратегии использования разделяемой памяти, развертывание циклов и другие параметры. Современные системы используют комбинацию аналитических моделей производительности и эмпириче-

ского профилирования для навигации в экспоненциально большом пространстве возможных конфигураций.

5.2 Эволюция «параллелизма по умолчанию»

Концепция «параллелизма по умолчанию» претерпела существенную эволюцию от идеи явного параллельного программирования к модели, где параллелизм извлекается и управляется автоматически компиляционной инфраструктурой. Этот переход отражает фундаментальное изменение в распределении ответственности между программистом и инструментарием.

Ранние попытки автоматического параллелизма – которые обозревались мной ранее – демонстрировали ограниченную эффективность из-за сложности анализа зависимостей и недостатка информации о намерениях программиста. Новая волна систем автотюнинга изменила подход: вместо попыток автоматически обнаружить параллелизм в последовательном коде, системы стали требовать от программиста явной спецификации параллельной структуры алгоритма на высоком уровне абстракции, беря на себя ответственность за выбор оптимальных параметров исполнения.

Проект Halide, представленный MIT в 2012 году, стал пионером в области декларативного программирования для обработки изображений. Halide ввел разделение алгоритма и расписание его исполнения: программист описывает вычисление математически, а затем отдельно выбирает стратегию параллелизации, тайлинга и векторизации. Разработчики Halide поняли, что для многих классов алгоритмов существует множество функционально эквивалентных, но различающихся по производительности способов организации вычислений[32].

Системы автотюнинга, такие как OpenTuner[16] и AutoTVM, развили эту идею дальше, применяя методы машинного обучения для автоматического исследования пространства возможных расписаний выполнения. Вместо того чтобы требовать от программиста вручную специфицировать оптимальное расписание, эти системы могут автоматически найти близкое к оптимальному решение через итеративное профилирование различных конфигураций. Этот подход особенно

ценен при портировании кода на новые аппаратные платформы, где интуиция, разработанная для одной архитектуры, может не работать для другой.

Современные системы также интегрируют адаптивные стратегии исполнения, где решения об оптимизации принимаются частично во время выполнения программы на основе фактических характеристик данных и текущей загрузки системы. Это особенно важно для гетерогенных систем, где оптимальное распределение работы между CPU и GPU может зависеть от размера входных данных и доступности вычислительных ресурсов в конкретный момент времени.

5.3 Изменение философии программирования

Трансформация подхода к высокопроизводительному программированию в 2010-х годах отражает более глубокие изменения в философии разработки программного обеспечения. Традиционная модель HPC-программирования предполагала, что оптимальная производительность достигается только через глубокое понимание аппаратной архитектуры и тщательную ручную оптимизацию каждого аспекта кода. Программист был обязан управлять иерархией памяти, распределением работы между вычислительными блоками, векторизацией и множеством других деталей низкого уровня. Это демократизировало доступ к вычислительной мощности и ускорило инновации в прикладных областях.

Новая парадигма переносит фокус программиста с оптимизации реализации на спецификацию модели и алгоритма. Программист определяет математическую структуру вычислений, а используемая инфраструктура берет на себя ответственность за оптимизацию исполнения. Это изменение не означает, что производительность стала менее важной, но что методы её достижения фундаментально изменились.

Критическим фактором, делающим этот подход возможным, стало развитие компиляторов, способных эффективно транслировать высокоуровневые спецификации в оптимизированный машинный код. Современные компиляторы для машинного обучения используют комбинацию классических оптимизационных техник, эвристик, основанных на доменных знаниях, и методов машинного обучения для навигации в сложном пространстве возможных реализаций. Система

Tiramisu, например, использует полиэдральную модель компиляции для автоматической оптимизации линейной алгебры и свёрточных операций[17]

Однако важно признать, что этот переход не является абсолютным. Для критических по производительности приложений или при работе с новыми, еще не поддержанными стандартными фреймворками паттернами вычислений, знание низкоуровневых деталей остается необходимым. Более того, разработка самих компиляционных инфраструктур требует глубокого понимания как высокоуровневых абстракций, так и низкоуровневых деталей аппаратной реализации. Можно говорить о специализации ролей: большинство программистов работают на уровне высокоуровневых абстракций, в то время как относительно небольшая группа экспертов развивает инфраструктуру, делающую это возможным.

Эволюция также изменила критерии оценки успешности программных решений. Если ранее основной метрикой было достижение максимальной производительности на конкретной аппаратной конфигурации, то современные системы оцениваются по способности обеспечить хорошую производительность на широком спектре архитектур с минимальными модификациями кода. «Портативность» производительности стала не менее важной, чем абсолютная производительность.

Будущее развитие этого направления, вероятно, будет характеризоваться дальнейшим повышением уровня абстракции и интеграцией более продвинутых методов автоматической оптимизации, включая использование самих нейронных сетей для предсказания оптимальных стратегий компиляции.

6 Смена парадигм: историко-философский анализ

6.1 Подход Куна: парадигмы и научные революции

Концепция научных революций Томаса Куна, изложенная в его фундаментальной работе «Структура научных революций»[25]. Согласно Куну, научное знание развивается не линейно и кумулятивно, а через периоды «нормальной науки», прерываемые радикальными сменами парадигм. В контексте HPC подобная динамика проявляется с особенной отчетливостью:

- переход на векторизацию в контексте высокопроизводительных вычислений;
- затем переход к массовому параллелизму;
- потом пришло поколение вычислений на GPGPU;
- приход DSL и автогенерации кода мы видим в настоящее время.

6.2 Проблема «аппаратно-зависимого программирования»

Фундаментальная проблема в истории HPC заключается в некоторой перпендикулярности взглядов, между стремлением к универсальности программных абстракций и неизбежной зависимостью от специфики аппаратного обеспечения. Этот конфликт имеет глубокие философские корни, восходящие к дихотомии между идеальным и материальным, между формой и содержанием.

История демонстрирует циклический паттерн: каждая новая эпоха начинается с повышения уровня абстракции, но постепенно программисты вынуждены спускаться к более низким уровням для достижения приемлемой производительности. Переход от Fortran к MPI казался шагом назад в плане удобства программирования, но был необходим для эффективного использования распределенных систем. Аналогично, CUDA представляет собой относительно низкоуровневый интерфейс по сравнению с традиционными последовательными языками, но обеспечивает контроль, необходимый для эффективного использования GPU.

Современные попытки создания универсальных моделей, такие как OpenCL и SYCL, стремятся найти баланс между портируемостью и производительностью. Однако практика показывает, что достижение производительности,

сопоставимой с платформо-специфичными решениями, часто требует использования расширений и оптимизаций, специфичных для конкретного оборудования. Это подтверждает тезис о том, что физические ограничения аппаратного обеспечения фундаментально влияют на структуру эффективных вычислений.

6.3 Текущие дискуссии в научном сообществе

Современный ландшафт высокопроизводительных вычислений характеризуется напряженными дебатами относительно будущего программных моделей и философии развития технологий. Центральная линия этих дискуссий проходит между проприетарными и открытыми стандартами, что отражает более широкий конфликт между коммерческими интересами и научными идеалами открытости и воспроизводимости.

CUDA быстро стала де-факто стандартом для программирования GPU в научных вычислениях и машинном обучении. Однако проприетарная природа CUDA создает существенные проблемы для научного сообщества. Исследования, основанные на CUDA, привязаны к оборудованию NVIDIA, что ограничивает воспроизводимость и создает зависимость от единственного производителя. Кроме того, закрытость CUDA противоречит принципам открытой науки, где воспроизводимость и независимая верификация являются фундаментальными ценностями.

OpenCL был задуман как универсальная альтернатива проприетарным решениям. Философия OpenCL воплощает идеалы открытости и портируемости: единый код может исполняться на процессорах различных производителей, включая CPU, GPU и FPGA. Однако на практике OpenCL столкнулся с серьезными трудностями – различия в реализациях между производителями, отсутствие единой экосистемы инструментов и библиотек, а также часто более низкая производительность по сравнению с платформо-специфичными решениями ограничили его принятие.

SYCL, разработанный Khronos Group и впервые выпущенный в 2014 году, представляет собой попытку модернизировать подход OpenCL через использование современных возможностей C++[33]. SYCL стремится соединить удобство

высокоуровневого программирования с производительностью и портируемостью. Проекты, основанные на SYCL, – например, oneAPI от Intel, – представляют видение будущего, в котором единая кодовая база может эффективно исполняться на разнородных архитектурах. Однако остается открытым вопрос, сможет ли SYCL преодолеть фундаментальное противоречие между универсальностью и оптимальной производительностью.

Дискуссия о будущем низкоуровневых языков программирования отражает более глубокий философский вопрос о природе вычислительной абстракции. С одной стороны, существует тенденция к повышению уровня абстракции, воплощенная в развитии DSL и автоматической генерации кода. Halide, TVM и подобные системы демонстрируют, что для многих классов приложений возможно автоматически генерировать высокоэффективный код из высокоуровневых спецификаций. С другой стороны, существуют веские аргументы в пользу сохранения значимости низкоуровневого программирования. Появление новых архитектур, таких как квантовые компьютеры, нейроморфные системы и специализированные ускорители для машинного обучения, требует глубокого понимания аппаратного уровня для эффективного использования этих систем.

Современное состояние можно охарактеризовать как период плюрализма и экспериментирования. Научное сообщество признает, что различные подходы имеют свои области применимости, и вместо поиска единственного универсального решения наблюдается развитие экосистемы взаимодополняющих технологий. Этот плюрализм отражает зрелость области, признающей сложность и многообразие вычислительных задач, но также создает вызовы для обучения специалистов и поддержания кодовых баз.

7 Связь исторической эволюции с современной научной задачей

7.1 Необходимость новых абстракций для гибридных CPU+GPU систем

Современные гетерогенные вычислительные системы представляют собой качественно новую парадигму, требующую переосмысления традиционных подходов к программированию и оптимизации. В отличие от классических параллельных архитектур, где все процессорные элементы были однородными, гибридные CPU+GPU системы характеризуются радикальной асимметрией вычислительных ресурсов[27].

Центральные процессоры оптимизированы для выполнения сложной логики управления, работы с нерегулярными структурами данных и минимизации задержек отдельных операций. Графические процессоры, напротив, достигают максимальной производительности на массово параллельных задачах с регулярными паттернами доступа к памяти и высокой арифметической интенсивностью[24].

Проблема усугубляется сложностью иерархии памяти в гетерогенных системах. Традиционные модели программирования не предоставляют эффективных механизмов для управления перемещением данных между различными адресными пространствами CPU и GPU, что может приводить к значительным накладным расходам. Современные абстракции должны обеспечивать автоматическую оптимизацию передачи данных, учитывая как объемы копируемой информации, так и возможности перекрытия коммуникаций с вычислениями.

Для задач машинного обучения и глубокого обучения, где инференс моделей требует выполнения последовательности разнородных операций, необходимы абстракции, способные автоматически распределять различные слои нейронных сетей между CPU и GPU в зависимости от их характеристик[3]. Некоторые операции, такие как небольшие полносвязные слои или нормализация батчей малого размера, могут эффективнее выполняться на CPU, в то время как крупные свёрточные слои лучше ложатся на массивно параллельные ядра GPU.

Заключение

Проведённый историко-философский анализ эволюции языков и моделей программирования для высокопроизводительных вычислений демонстрирует закономерную последовательность сдвигов парадигм. От Fortran как первого высокоуровневого языка для научных вычислений область прошла через эпохи векторных суперкомпьютеров, массового параллелизма, революции GPGPU-вычислений и пришла к современному этапу доменно-специфичных языков и автоматической генерации кода.

Применение концепции научных революций Томаса Куна к развитию HPC выглядит логичным и обоснованным. Каждая смена парадигм сопровождалась фундаментальным переосмыслением базовых принципов программирования и изменением распределения ответственности между программистом и инфраструктурой компиляторов.

Современное состояние области характеризуется тремя определяющими тенденциями. Первой является унификация промежуточных представлений через проекты типа MLIR, создающие универсальную инфраструктуру разработки и оптимизации. Второй стало возрастание роли фреймворков и компиляторов, способных автоматически генерировать код, конкурирующий по производительности с ручными оптимизациями. Третьей тенденцией является доминирование гибридных CPU+GPU архитектур, требующих качественно новых абстракций для автоматического распределения вычислений.

Будущее высокопроизводительных вычислений, по-видимому, будет определяться многоуровневыми компиляторами, способными транслировать высокоуровневые спецификации в оптимизированный код для конкретных платформ с использованием методов машинного обучения для автоматического исследования пространства оптимизаций. Ручное низкоуровневое программирование сохранится как удел узкоспециализированных экспертов, работающих над самой инфраструктурой компиляторов. Для прикладных задач критическое значение приобретёт автоматическая генерация кода под конкретные аппаратные конфигу-

рации, что позволит достигать портативности производительности без жертв в эффективности.

Историко-философский анализ имеет существенное значение для понимания логики смены парадигм и прогнозирования будущих направлений развития. История демонстрирует, что эволюция определяется не только технологическими возможностями, но и философскими представлениями научного сообщества о природе вычислений, что делает междисциплинарный подход особенно ценным.

СПИСОК СОКРАЩЕНИЙ

ИИ – искуственный интеллект

HPC (High-Performance Computing) – высокопроизводительные вычисления

GPU (Graphics Processing Unit) – графический процессор

SIMD (Single Instruction, Multiple Data) – вычислительный подход, когда одна инструкция исполняется над некоторым массивом данных в один проход

MIMD (Multiple Instruction, Multiple Data) – вычислительный подход, когда множество инструкций исполняется над множеством потоков данных

CUDA (Compute Unified Device Architecture) – программно-аппаратная архитектура параллельных вычислений, которая позволяет существенно увеличить вычислительную производительность благодаря использованию графических процессоров фирмы NVIDIA

IR (Intermediate Representation) – промежуточное представление высокоуровневого кода, который можно оптимизировать и компилировать в нативный код любой поддерживаемой компилятором платформы

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Abadi M. [и др.]. TensorFlow: A system for large-scale machine learning [Электронный ресурс]. URL: http://arxiv.org/abs/1605.08695 (дата обращения: 08.12.2025).

2. Chen T. [и др.]. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning [Электронный ресурс]. URL: http://arxiv.org/abs/1802.04799 (дата обращения: 08.12.2025).

3. Jia Z., Zaharia M., Aiken A. Beyond Data and Model Parallelism for Deep Neural Networks [Электронный ресурс]. URL: http://arxiv.org/abs/1807.05358 (дата обращения: 08.12.2025).

4. Paszke A. [и др.]. PyTorch: An Imperative Style, High-Performance Deep Learning Library [Электронный ресурс]. URL: http://arxiv.org/abs/1912.01703 (дата обращения: 08.12.2025).

5. Lattner C. [и др.]. MLIR: A Compiler Infrastructure for the End of Moore's Law [Электронный ресурс]. URL: http://arxiv.org/abs/2002.11054 (дата обращения: 08.12.2025).

6. Frequently Asked Questions | TOP500 [Электронный ресурс]. URL: https://top500.org/resources/frequently-asked-questions/ (дата обращения: 08.12.2025).

7. The History of the Development of Parallel Computing | PARALLEL.RU - Информационно-аналитический центр по параллельным вычислениям [Электронный ресурс]. URL: https://parallel.ru/history/wilson_history.html (дата обращения: 08.12.2025).

8. MPI Documents [Электронный ресурс]. URL: https://www.mpi-forum.org/docs/ (дата обращения: 08.12.2025).

9. CUDA Toolkit Archive [Электронный ресурс]. URL: https://developer.nvidia.com/cuda-toolkit-archive (дата обращения: 08.12.2025).

10. OpenCL - an overview | ScienceDirect Topics [Электронный ресурс]. URL: https://www.sciencedirect.com/topics/computer-science/opencl (дата обращения: 08.12.2025).

11. Fortran | IBM [Электронный ресурс]. URL: https://www.ibm.com/history/fortran (дата обращения: 08.12.2025).

12. Why ALGOL was an important programming language? [Электронный ресурс]. URL: https://bulldogjob.com/readme/why-algol-was-an-important-programming-language (дата обращения: 08.12.2025).

13. Архитектура Cray-1 | PARALLEL.RU - Информационно-аналитический центр по параллельным вычислениям [Электронный ресурс]. URL: https://parallel.ru/history/cray1.html (дата обращения: 08.12.2025).

14. Vectorization Directives | Cray Fortran Reference Manual 8.7A S-3901 [Электронный ресурс]. URL: https://support.hpe.com/hpesc/public/docDisplay?docId=a00113909en_us&page=Vectorization_Directives.html&docLocale=en_US (дата обращения: 08.12.2025).

15. Allen R., Kennedy K. Automatic translation of FORTRAN programs to vector form // ACM Transactions on Programming Languages and Systems. 1987. № 4 (9). С. 491–542.

16. Ansel J. [и др.]. OpenTuner: an extensible framework for program autotuning Edmonton AB Canada: ACM, 2014.С. 303–316.

17. Baghdadi R. [и др.]. Tiramisu: A Polyhedral Compiler for Expressing Fast and Portable Code Washington, DC, USA: IEEE, 2019.С. 193–205.

18. Chamberlain B. L. Chapel // Programming Models for Parallel Computing. 2015. С. 129–160.

19. Chang R.-G., Chuang T.-R., Lee J. K. Support and optimization for parallel sparse programs with array intrinsics of Fortran 90 // Parallel Computing. 2004. № 4 (30). С. 527–550.

20. Flynn M. Very high-speed computing systems // Proceedings of the IEEE. 1966. № 12 (54). С. 1901–1909.

21. Garland M. [и др.]. Parallel Computing Experiences with CUDA // IEEE Micro. 2008. № 4 (28). С. 13–27.

22. Hecht M. S. Flow analysis of computer programs / M. S. Hecht, New York, NY: North-Holland, 1977.

23. Hoare C. A. R. Communicating sequential processes // Communications of the ACM. 1978. № 8 (21). С. 666–677.

24. Kirk D., Hwu W.-m. W. Programming massively parallel processors: a hands-on approach / D. Kirk, W.-m. W. Hwu, Third edition-е изд., Amsterdam Boston Heidelberg: Elsevier, Morgan Kaufmann, 2017.

25. Kuhn T. S. The structure of scientific revolutions / T. S. Kuhn, 3. ed., [Nachdr.]-е изд., Chicago: Univ. of Chicago Press, 2009.

26. McCarthy J. History of LISP // History of programming languages. 1978. С. 173–185.

27. Mittal S., Vetter J. S. A Survey of CPU-GPU Heterogeneous Computing Techniques // ACM Computing Surveys. 2015. № 4 (47). С. 1–35.

28. Morgan T. P. Compiling History To Understand The Future [Электронный ресурс]. URL: https://www.nextplatform.com/2018/11/02/compiling-history-to-understand-the-future/ (дата обращения: 08.12.2025).

29. Munshi A. The OpenCL specification Stanford, CA, USA: IEEE, 2009.С. 1–314.

30. Nickolls J. [и др.]. Scalable Parallel Programming with CUDA: Is CUDA the parallel programming model that application developers have been waiting for? // Queue. 2008. № 2 (6). С. 40–53.

31. Nickolls J., Dally W. J. The GPU Computing Era // IEEE Micro. 2010. № 2 (30). С. 56–69.

32. Ragan-Kelley J. [и др.]. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines Seattle Washington USA: ACM, 2013.С. 519–530.

33. Reinders J. [и др.]. Data parallel C++: mastering DPC++ for programming of heterogeneous systems using C++ and SYCL / J. Reinders, B. Ashbaugh, J. Brodman, M. Kinsner, J. Pennycook, [и др.]., New York: Apress open, 2021.