Обзор параллельных вычислений

Уровни вычислений

- Параллелизм на уровне машин, кластеры
- Параллелизм на уровне ОС, процессы, потоки, параллельный код
- Параллелизм на уровне hardware, CPU

Виды параллельных архитектур

- **SMP** (Shared Memory Processing), компьютеры с общей памятью
- **MPP** (Massive Parallel Processing), системы с распределенной памятью
- **NUMA** (Non Uniform Memory Access), системы с неоднородным доступом к памяти.

Для каждого вида архитектур существуют свои инструменты (OpenMP для SMP, MPI для MPP и т.д.)

Законы Амдала

Первый закон Амдала: производительность вычислительной системы, состоящей из связанных между собой устройств, в общем случае определяется самым непроизводительным ее устройством.

Второй закон Амдала: пусть система состоит из s простых универсальных устройств. Предположим, что при выполнении параллельной части алгоритма все s устройств загружены полностью. Тогда максимально возможное ускорение равно:

s - число устройств, β - доля последовательных вычислений

$$R = \frac{s}{\beta s + (1 - \beta)}$$

Третий закон Амдала: пусть система состоит из простых одинаковых универсальных устройств. При любом режиме работы ее ускорение не может превзойти обратной величины доли последовательных вычислений.

Основные проблемы

- **Race condition** - не все варианты чередования потоков приводят к корректному результату.
- **Lock contention** - высокая конкуренция при попытке захвата lock (например, вход в критическую секцию)
- **Deadlock** - взаимная блокировка
- **Starvation** - неравномерность распределения общих ресурсов между потоками
- **Unsafety** - небезопасный доступ к несинхронизованным данным, отсутствие атомарности операций, утечка ссылок на private объекты потоков и т.д.

Признаки качественной параллельной программы

- **Scalability** - при увеличении размерности данных или количества вычислительных ресурсов (потоки, машины) производительность должна меняться по закону O(N)

- **Safety** - безопасность многопоточного кода, отсутствие доступа к несинхронизованным данным, изоляция потоков и т.д.

- **Fine-grained synchronization** - минимизировать размер синхронизованных участков кода

- **Fairness** - "справедливость" распределения ресурсов между потоками.

MapReduce

MapReduce - программная модель для осуществления распределенных вычислений над большими объемами данных, а также framework для обработки больших объемов данных на кластерах.

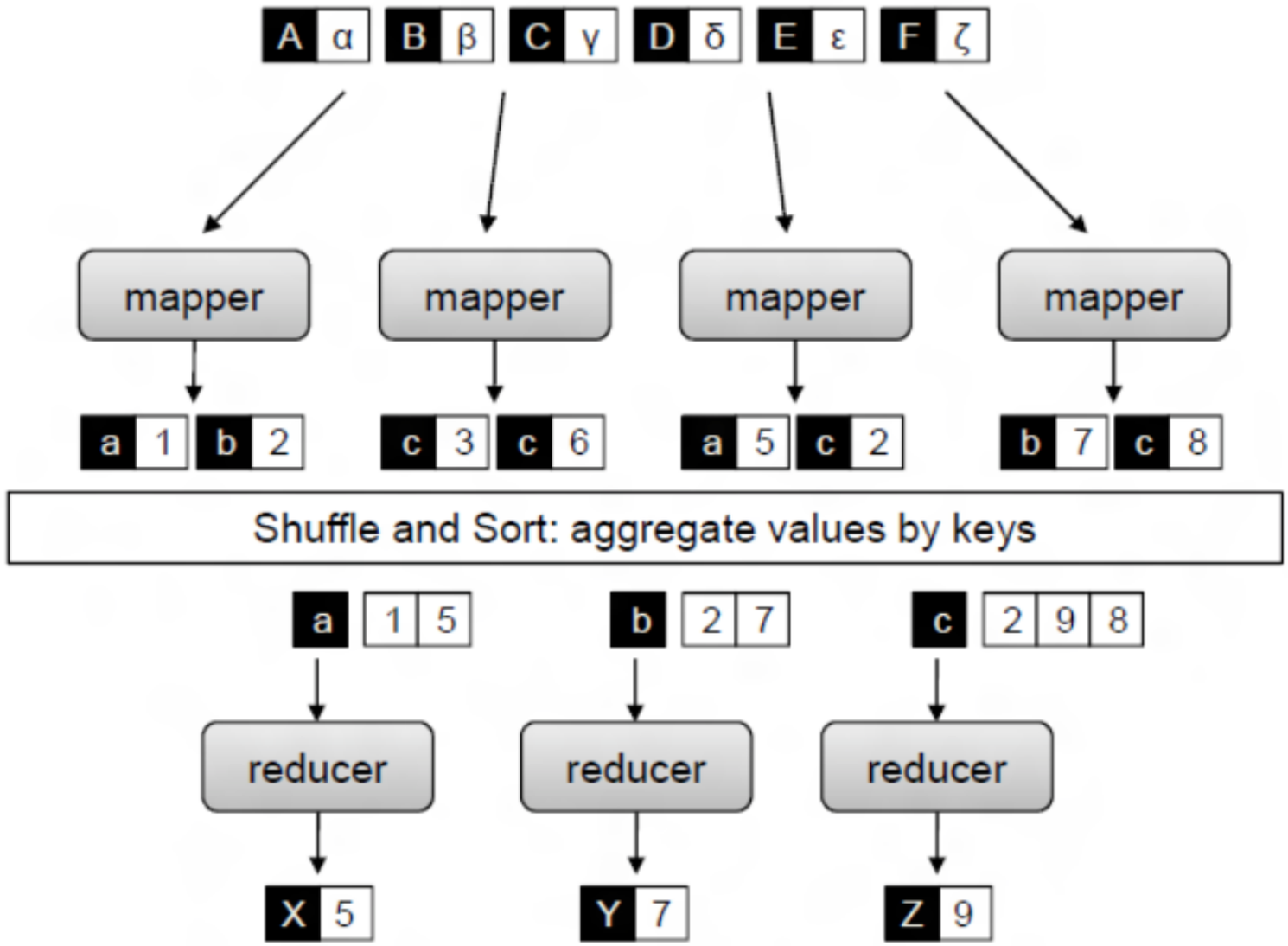
Изначально MapReduce был предложен компанией Google, которая разработала свою реализацию данной концепции. В настоящее наиболее распространенная реализация – Hadoop (проект Apache).

Применения:

- «Web-scale» обработка, связанная с анализом, фильтрацией, мониторингом и сбором веб-контента.

- Анализ поведения пользователей, включающая в себя запись действий пользователей (какие страницы смотрит, куда нажимает и т.д.)

MapReduce. Модель вычислений



MapReduce. Основные моменты

- Огромное количество данных (петабайты). Не влезаем в RAM.
- Scaling out - большое количество обычных (не мощных) серверов.
- Отказы - нормальное явление.
- Процессоры и хранилища вместе. Узел обрабатывает фрагмент общих данных на своем жестком диске.
- Последовательная обработка данных, избегать обращения к памяти по случайным адресам. Связано с особенностями жестких дисков.
- Seamless scalability (плавная масштабируемость). Speedup - при удвоении числа данных алгоритм должен затрачивать не более чем в два раза времени. Scaleup - при удвоении размера кластера алгоритм должен затрачивать не более чем половину изначального времени.
- Основной подход - разделяй и властвуй.

MapReduce. Ссылки

- https://hadoop.apache.org

- http://research.google.com/archive/mapreduce.html

- http://docs.mongodb.org/manual/core/map-reduce/

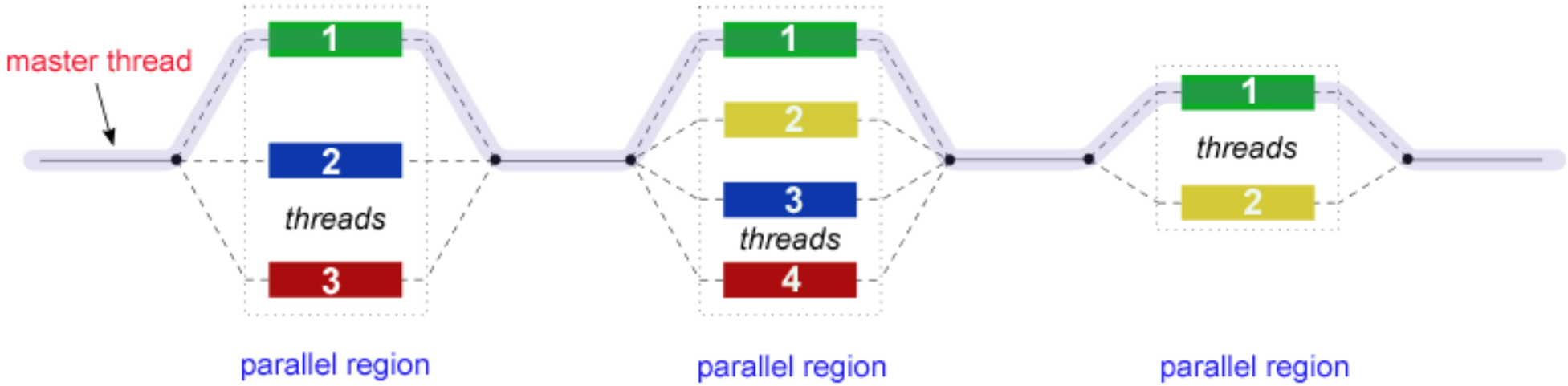
OpenMP

OpenMP - средство для программирования для компьютеров с общей памятью (SMP). За основу берется последовательная программа, а для создания ее параллельной версии пользователю предоставляются директивы, функции и переменные окружения. Разработан для C/C++ и Fortran.

OpenMP. Особенности

- Исходный код разбит на последовательные и параллельные секции.
- В начальный момент порождается поток-мастер, который выполняет последовательные секции.
- При входе в параллельную секцию порождаются дополнительные потоки, каждый из них получает свой номер. При выходе из параллельной секции мастер дожидается завершения остальных потоков.
- В параллельной секции переменные делятся на общие и локальные. Общие переменные существуют в одном экземпляре для всех потоков, локальные - по экземпляру на поток.

OpenMP. Схема работы



OpenMP. Обзор

`#pragma omp` - общий вид директив

`#pragma omp parallel { ... }` - задание параллельной области

Число потоков, которые создадутся в параллельной области, задаются переменной окружения `OMP_NUM_THREADS`.

`omp_set_num_threads(int num)` - изменить число потоков (изменяется `OMP_NUM_THREADS`)

`omp_get_num_threads()` - получить число потоков

`omp_get_thread_num()` - получить номер текущего потока

OpenMP. Обзор

```
int i, i1, i2;
#pragma omp parallel num_threads(4) private(i) shared(i1, i2)
{
    ...
}
```

4 потока, в том числе поток-мастер, будут выполнять код внутри области, с локальными переменными i и общими i1, i2.

Выполнение потоками различного кода:

```
#pragma omp parallel
{
    if (omp_get_thread_num() == 3)
    {
        ...
    }
}
```

OpenMP. Обзор

Если в параллельной области встретится оператор цикла, каждый поток выполнит все итерации цикла. Для распределения итераций цикла между потоками используется #pragma omp parallel for

```
#pragma omp parallel for
for (int i = 0; i < n; i++)
{
...
}
```

Выбор способа распределения итераций между потоками делается с помощью опции schedule:

```
#pragma omp parallel for schedule (dynamic, 3) { ... }
```

OpenMP. Обзор

Параллелизм на уровне независимых секций:

```
#pragma omp sections
{
    #pragma omp section
    {
        ...
    }
    #pragma omp section
    {
        ...
    }
}
```

Каждая секция внутри #pragma omp sections будет выполнена только один раз каким-либо потоком.

OpenMP. Обзор

```
#pragma omp barrier
```

Потоки, доходя до этой директивы, останавливаются и ждут, пока оставшиеся потоки не дойдут до нее.

```
#pragma omp master { ... }
```

Директива выделяет участок кода, который будет выполнен только мастером-потоком.

```
#pragma omp critical { ... }
```

Директива определяет критическую секцию, которую в некоторый момент времени выполняет только один поток.

```
#pragma omp atomic
```

Директива обеспечивает атомарность выполнения идущего вслед за ней оператора.

OpenMP. Преимущества

- Инкрементальное распараллеливание. Разработчик берет последовательную программу и последовательно добавляет в нее параллелизм.

- На однопроцессорной платформе OpenMP-программа может быть использована в качестве последовательной. Нет нужды делать отдельно последовательную и параллельную версии программы.

OpenMP. Ссылки

- http://openmp.org/wp/
- http://parallel.ru/tech/tech_dev/openmp.html
- http://openmp.ru

TBB

Intel **T**hreading **B**uilding **B**locks — кроссплатформенная библиотека шаблонов C++, разработанная компанией Intel для параллельного программирования.

Предоставляет абстракцию *задача*, которая позволяет избежать ручной контроль потоков.

Структура библиотеки:

- параллельные алгоритмы - for, reduce, pipeline, sort, ...
- потокобезопасные контейнеры - векторы, очереди, хэш-таблицы, ...
- мьютексы
- атомарные операции
- и т.д.

Ссылки:

- https://www.threadingbuildingblocks.org
- https://software.intel.com/en-us/intel-tbb

MPI

MPI (Message Passing Interface) - интерфейс передачи сообщений между процессами, выполняющими одну задачу.

MPI. Особенности

- Одна программа - множество процессов. Все процессы выполняют один и тот же код.

- **Коммуникатор** - группа процессов, используемых при выполнении операции передачи данных. Каждый процесс имеет номер (ранг) в рамках некоторого коммуникатора. Коммуникатор MPI_COMM_WORLD создается по умолчанию и включает в себя все процессы программы.

- Основной способ общения процессов - посылка сообщений. Сообщение - набор данных некоторого типа, имеющее несколько атрибутов - номер процесса-отправителя, номер процесса-получателя, идентификатор сообщения и т.д. Функции передачи данных делятся на коллективные и point-to-point.

- При выполнении операции передачи данных нужно указывать тип данных, которые передаются: MPI_INT, MPI_FLOAT, MPI_CHAR и т.д. MPI рассчитан на гетерогенную систему, на разных узлах могут различаться типы данных.

MPI. Обзор

```
MPI_Comm_size(MPI_Comm comm, int *size)
```

Определение общего числа процессов в коммуникаторе *comm*.

```
MPI_Comm_rank(MPI_Comm comm, int *rank)
```

Определение номера процесса в коммуникаторе *comm*.

```
MPI_Send(void *buf, int count, MPI_Data datatype, int dest, int
msgtag, MPI_Comm comm)
```

Блокирующая посылка сообщения с идентификатором *msgtag*, состоящего из *count* элементов типа *datatype*, процессу с номером *dest*. Все элементы посылаемого сообщения расположены подряд в буфере *buf*.

```
MPI_Recv(void *buf, int count, MPI_Data datatype, int source, int
msgtag, MPI_Comm comm, MPI_Status *status)
```

Прием сообщения с идентификатором *msgtag* от процесса *source* с блокировкой.

MPI. Обзор

```
MPI_Bcast(void *buf, int count, MPI_Datatype datatype,
int source, MPI_Comm comm)
```

Рассылка сообщения от процесса source всем процессам коммуникатора *comm*, включая рассылающий процесс.

```
MPI_Gather(void *sbuf, int count, MPI_Datatype stype,
void *rbuf, int rcount, MPI_Datatype rtype, int dest,
MPI_Comm comm)
```

Сборка данных со всех процессов в буфере *rbuf* процесса *dest*. Каждый процесс, включая *dest*, посылает содержимое своего буфера *sbuf* процессу *dest*, собирающий процесс сохраняет данные в *rbuf*, располагая их в порядке возрастания номеров процессов. Параметр *rcount* обозначает число элементов, получаемых от каждого процесса в отдельности.

MPI. Обзор

```
MPI_Scatter(void *sbuf, int scount, MPI_Datatype stype,
void *rbuf, int rcount, MPI_Datatype rtype, int source,
MPI_Comm comm)
```

Процесс *source* рассылает порции данных из *sbuf* всем остальным процессам. Буфер *sbuf* делится на *N* равных частей (*N* - число процессов коммуникатора), состоящих из *scount* элементов типа *stype*, после чего i-ая часть отдается i-му процессу.

```
MPI_Reduce(void *sbuf, void *rbuf, int count, MPI_Datatype
datatype, MPI_Op op, int root, MPI_Comm comm)
```

Выполняется *count* операций *op*. В буфере *sbuf* каждого процесса располагается *count* аргументов типа *datatype*. Первые элементы массивов *sbuf* участвуют в первой операции *op*, вторые - во второй и т.д. Результаты всех *count* операций записываются в буфер *rbuf* процесса *root*.

MPI. Обзор

```
MPI_Type_struct(int count, int
*blocklens, int *offsets, MPI_Datatype
*types, MPI_Datatype *newtype)
```

Создание структурного типа данных *newtype* из *count* блоков по *blocklens[i]* элементов типа *types[i]*. i-ый блок начинается через *offsets[i]* байт с начала буфера посылки.

MPI. Ссылки

- http://parallel.ru/tech/tech_dev/mpi.html
- http://www.open-mpi.org
- http://www.mpi-forum.org
- http://www.intuit.ru/studies/courses/4447/983/lecture/14927

CUDA

CUDA - программно-аппаратная архитектура параллельных вычислений, которая позволяет существенно увеличить вычислительную производительность благодаря использованию графических процессоров фирмы NVIDIA.

CUDA SDK позволяет программистам реализовывать на специальном упрощённом диалекте языка программирования Си алгоритмы, выполнимые на графических процессорах NVIDIA, и включать специальные функции в текст программы на Си. Архитектура CUDA даёт разработчику возможность по своему усмотрению организовывать доступ к набору инструкций графического ускорителя и управлять его памятью.

CUDA. Особенности и применения

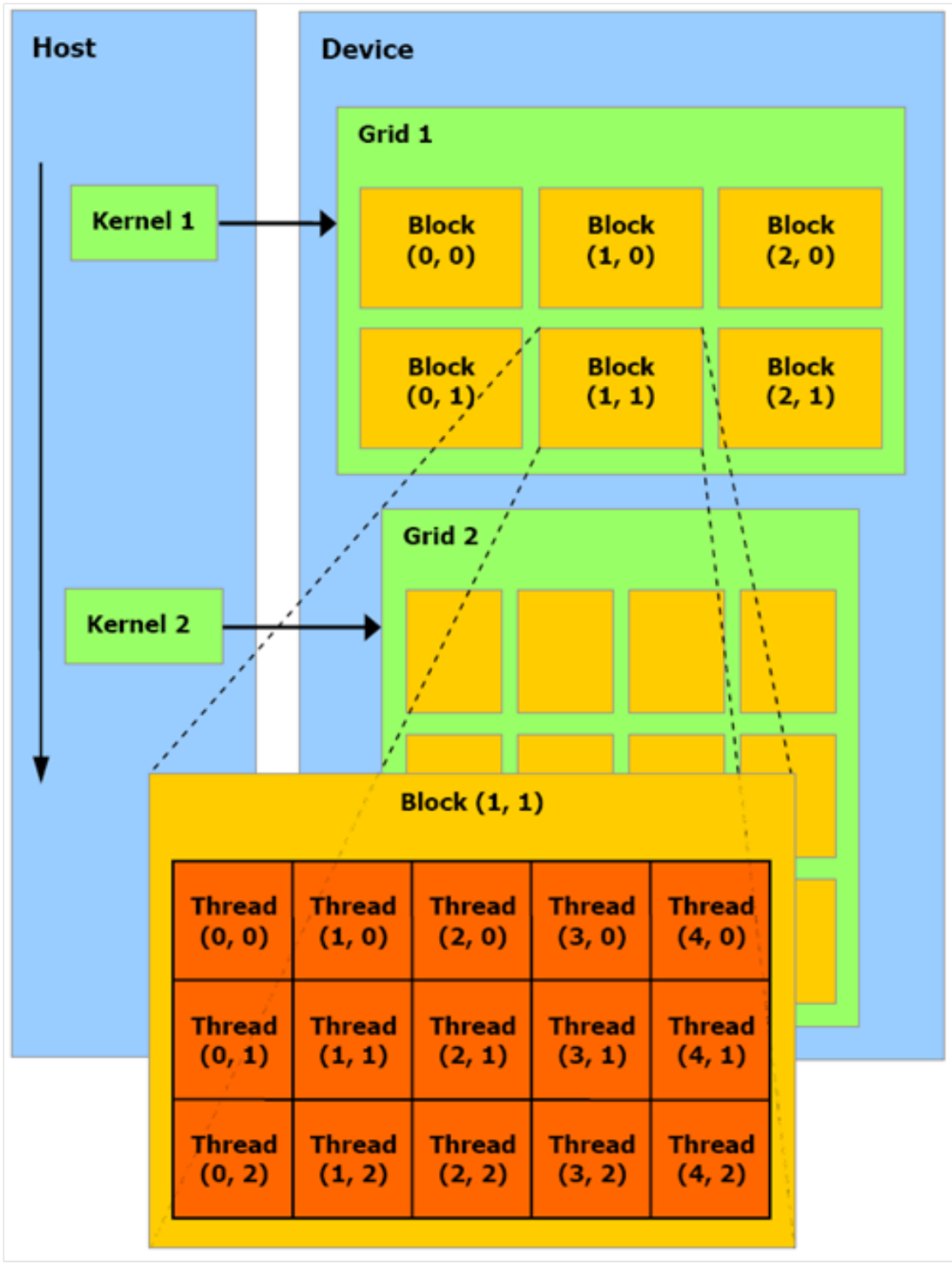
Особенности архитектуры CUDA:

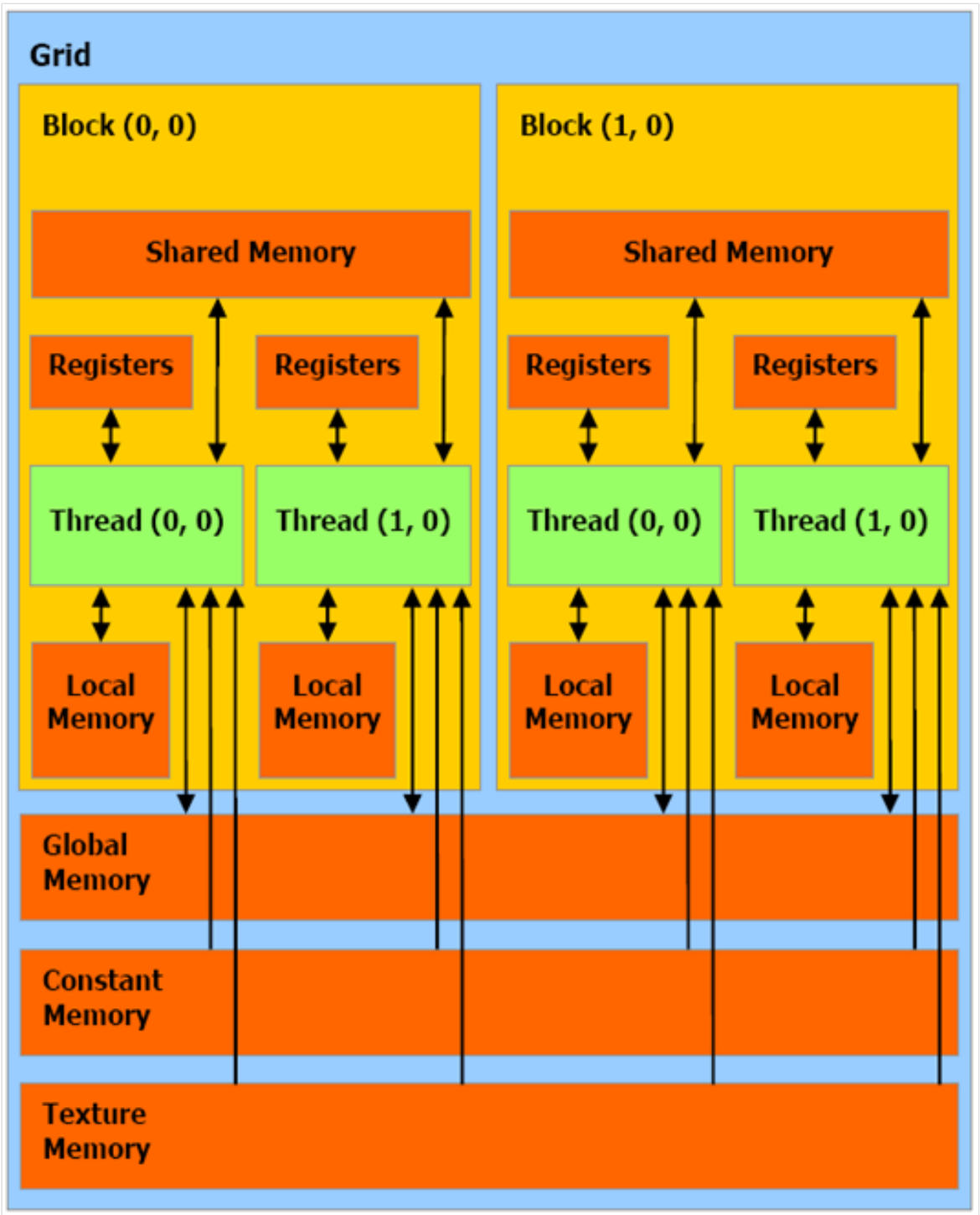
- Наличие унифицированного шейдерного конвейера, позволяющего программе задействовать любое АЛУ, входящее в микросхему.
- Набор команд, ориентированный на вычисления общего назначения, а не только на графику
- Разрешение произвольного доступа к памяти для чтения и записи исполняющим устройствам GPU, а также доступ к программно-управляющему кэшу – разделяемой памяти.
- Вместо программирования на шейдерных языках типа GLSL/HLSL через OpenGL/DirectX применяется расширенный язык C – CUDA C.

Применения:

- Обработка медицинских изображений
- Вычислительная гидродинамика
- Науки об окружающей среде (климат)

CUDA. Модель вычислений





CUDA. Hello World

```
#include <stdio.h>

__global__ void kernel() {
}

int main() {
    kernel<<<1,1>>>();
    printf("Hello, World!\n");
    return 0;
}
```

В одном файле находится код предназначенный для исполнения как CPU (host), так и GPU (device).

Функция kernel передается компилятору, обрабатывающему код для устройства. <<<x, y>>> - вызов кода (с созданием *x* параллельных блоков и *y* потоков), выполняемого GPU. В общем случае x и y - трехмерные решетки (dim3).

CUDA. Работа с памятью GPU

```
__global__ void add(int a, int b, int *c) {
      *c = a + b;
}

int main() {
      int c;
      int *dev_c;
      cudaMalloc( (void**)&dev_c, sizeof(int) );
      add<<<1, 1>>>(2, 7, dev_c);
      cudaMemcpy(&c, dev_c, sizeof(int), cudaMemcpyDeviceToHost);
      printf("2 + 7 = %d\n", c);
      cudaFree(dev_c);

      return 0;
}
```

В этой программе:

1. Передаются параметры ядру
2. Выделяется память, чтобы устройство через нее вернуло данные CPU.

CUDA. Работа с памятью GPU

cudaMalloc() выделяет память на GPU. Первый аргумент – указатель на указатель, в котором будет возвращен адрес выделенной области памяти, второй – размер этой памяти. Правила работы с указателями на память устройства:

- Разрешается передавать указатели на память, выделенную cudaMalloc() функциям, исполняемым GPU
- Разрешается использовать указатели на память, выделенную cudaMalloc() для чтения и записи в эту память в коде, исполняемом GPU.
- Разрешается передавать указатели на память, выделенную cudaMalloc() функциям, исполняемым CPU.
- Не разрешается использовать указатели на память, выделенную cudaMalloc(), для чтения и записи в эту память в коде, который исполняется CPU.

Поэтому область памяти GPU копируется функцией cudaMemcpy() в память CPU. Последний параметр – константа, описывающая направление копирования памяти:

- `cudaMemcpyDeviceToHost`
- `cudaMemcpyHostToDevice`
- `cudaMemcpyDeviceToDevice`

CUDA. Сложение матриц

Выбрана размерность блока (16, 16, 1).

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N], float C[N][N]) {
     int i = blockIdx.x * blockDim.x + threadIdx.x;
     int j = blockIdx.y * blockDim.y + threadIdx.y;
     if (i < N && j < N)
          C[i][j] = A[i][j] + B[i][j];
}

int main() {
     ...
     // Kernel invocation
     dim3 threadsPerBlock(16, 16);
     dim3 numBlocks(N/threadsPerBlock.x, N/threadsPerBlock.y);
     MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
     ...
}
```

CUDA. Ссылки

- http://www.nvidia.ru/object/cuda-parallel-computing-ru.html
- https://developer.nvidia.com/cuda-zone
- http://opencv.org/platforms/cuda.html

Java Concurrency

Язык Java содержит множества инструментов для параллельного программирования в пакете `java.util.concurrent`

Запуск кода в новом потоке:

```
public class HelloRunnable implements Runnable {
    public void run() {
        System.out.println("Hello from a thread!");
    }
    public static void main(String args[]) {
        (new Thread(new HelloRunnable())).start();
    }
}
```

Java Concurrency. Синхронизация

Java поддерживает синхронизацию на уровне методов и групп инструкций.

```
public class SynchronizedCounter {
    private int c = 0;
    public synchronized void increment() {
        c++;
    }
    public synchronized void decrement() {
        c--;
    }
    public synchronized int value() {
        return c;
    }
}
```

Методы, помеченные ключевым слово **`synchronized`**, не могут выполняться более чем одним потоком в рамках одного объекта. Если поток попытается вызвать один из **`synchronized`** методов объекта в то время как другой поток уже находится внутри **`synchronized`** метода, вызывающий поток будет ожидать, пока выполняющий **`synchronized`** метод не закончит выполнять метод.

Java Concurrency. Синхронизация

Синхронизация в Java базируется на *мониторах* (intrinsic lock, monitor lock). С каждым объектом ассоциирован монитор. Поток, которому нужен эксклюзивный доступ к объекту, захватывает монитор и освобождает впоследствии.

Когда поток вызывает **`synchronized`** метод, он неявно захватывает монитор соответствующего объекта.

Java Concurrency. Синхронизация

В Java есть возможность объявлять синхронизованные инструкции.

```
public class MsLunch {
    private long c1 = 0;
    private long c2 = 0;
    private Object lock1 = new Object();
    private Object lock2 = new Object();
    public void inc1() {
        synchronized(lock1) {
            c1++;
        }
    }
    public void inc2() {
        synchronized(lock2) {
            c2++;
        }
    }
}
```

Переменные c1 и c2 никогда не используются вместе, лучше не использовать `synchronized` метод.

Java Concurrency. Атомарность

Атомарная инструкция неделима. Обычный инкремент *i++* неатомарен. В Java гарантированно атомарны следующие инструкции:

- Чтение и запись в ссылочные переменные и большинство примитивов (кроме *long* и *double*)

- Чтение и запись во *все* переменные, в объявлении которых присутствует ключевое слово **volatile**.

Volatile обеспечивает консистентный образ переменной в памяти. Например, запись в volatile переменную, сделанную одним потоком, другие потоки гарантированно увидят.

Java Concurrency. Guarded blocks

Потокам зачастую нужно координировать действия друг с другом.

Пусть метод **`guardedJoy()`** не должен продолжать выполнение, пока переменная **`joy`** не будет установлена в **`true`**.

```
public void guardedJoy() {
    // Simple loop guard. Wastes
    // processor time. Don't do this!
    while(!joy) {}
    System.out.println("Joy has been achieved!");
}
```

Java Concurrency. Guarded blocks

Более эффективный способ - использовать **`Object.wait()`**. Поток, вызвавший **`wait()`**, не получит назад управление, пока другой поток не вызовет у этого объекта **`notify()`** или **`notifyAll()`**. Поток, вызывающий **`wait()`**, должен завладеть монитором этого объекта. Когда **`wait()`** вызван, поток возвращает монитор.

```
public synchronized void guardedJoy() {
    // This guard only loops once for each special event,
    // which may not be the event we're waiting for.
    while(!joy) {
        try {
            wait();
        } catch (InterruptedException e) {}
    }
    System.out.println("Joy has been achieved!");
}
```

Java Concurrency. Обзор

Для более сложных задач предназначены классы из `java.util.concurrent.*`

- **Lock objects** - реализуют различные блокирующие идиомы
- **Synchronizers** - вспомогательные утилиты для синхронизации потоков.
- **Executors** - определяют высокоуровневое API для запуска и управления потоками.
- **Concurrent collections** - потоко-безопасные коллекции
- **Atomic variables** - атомарные переменные

Java Concurrency. Concurrent collections

- **`CopyOnWriteArrayList<E>`** - потокобезопасный аналог ArrayList, реализованный с CopyOnWrite алгоритмом. Создает новую копию списку при каждой мутирующей операции (add, set, remove). Эффективно, если много чтений и редкие записи.

- **`ConcurrentHashMap<K, V>`** - в отличие от блоков synhronized на HashMap, данные представлены в виде сегментов, разбитых по hash'ам ключей. В результате, для доступ к данным лочится по сегментам, а не по одному объекту.

Java Concurrency. Synchronizers

- **`Semaphore`** - используются для ограничения количества потоков для доступа к некоторому ресурсу, управляется с помощью счетчика. Если он больше нуля, то доступ разрешается, а значение счетчика уменьшается. Если счетчик равен нулю, то текущий поток блокируется, пока другой поток не освободит ресурс.

- **`CyclicBarrier`** - может использоваться для синхронизации заданного количества потоков в одной точке. Барьер достигается когда N потоков вызовут метод **`await(...)`** и заблокируются. После чего счетчик сбрасывается в исходное значение, а ожидающие потоки освобождаются.

Java Concurrency. Executors

- **`Future<V>`** — интерфейс для получения результатов работы асинхронной операции. Метод **`get()`**, который блокирует текущий поток (с таймаутом или без) до завершения работы асинхронной операции в другом потоке. Также, дополнительно существуют методы для отмены операции и проверки текущего статуса.

- **`Callable<V>`** — расширенный аналог интерфейса **`Runnable`** для асинхронных операций. Позволяет возвращать типизированное значение и кидать checked exception.

Java Concurrency. Executors

- **`Executor`** — gредставляет собой базовый интерфейс для классов, реализующих запуск **`Runnable`** задач.

- **`ExecutorService`** — интерфейс, который описывает сервис для запуска **`Runnable`** или **`Callable`** задач. Методы **`submit()`** на вход принимают задачу в виде **`Callable`** или **`Runnable`**, а в качестве возвращаемого значения идет **`Future`**, через который можно получить результат.

- **`ThreadPoolExecutor`** — используется для запуска асинхронных задач в пуле потоков. Тем самым практически полностью отсутствует оверхэд на поднятие и остановку потоков. А за счет фиксируемого максимума потоков в пуле обеспечивается прогнозируемая производительность приложения.

Java Concurrency. Atomics

- `AtomicBoolean, AtomicInteger, AtomicLong, AtomicIntegerArray, AtomicLongArray` — атомарные варианты примитивов и примитивных массивов. Эффективнее, чем брать обычный примитив и совершать операции через **`synchronized`** секции. В некоторых случаях достаточно **`volatile`**.

- **`AtomicReference`** - атомарная объектная ссылка.

Java Concurrency. Ссылки

- http://docs.oracle.com/javase/tutorial/essential/concurrency/
- http://www.cs.umd.edu/class/fall2002/cmsc433-0201/lectures/cpjslides.pdf
- http://www.amazon.com/Java-Concurrency-Practice-Brian-Goetz/dp/0321349601
- http://habrahabr.ru/company/luxoft/blog/157273/

Аппаратный параллелизм

- **ILP** (Instruction-Level Parallelism) - параллелизм на уровне машинных команд, при котором последовательность инструкций одного потока выполнения может выполняться процессором одновременно.

- **Суперскалярные** процессоры самостоятельно обнаруживают параллелизм в машинном коде и распределяют его по своим устройствам.

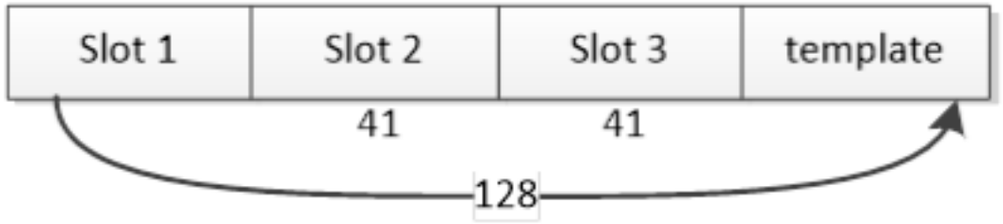
- **VLIW** (Very Long Instruction Word) процессоры выполняют команды, в которых закодировано несколько инструкций сразу. Эти инструкции исполняются одновременно. Нужен специальный компилятор, порождающий такие команды.

Intel Itanium

VLIW	**Суперскаляр**
Инструкции имеют одинаковую длину и запакованы в очень большие слова	Инструкции имеют переменную длину и никак не запакованы
Статическое планирование инструкций	Динамическое планирование инструкций
Нет переименования регистров	Есть переименование регистров
Статическая привязка инструкций к исполняющему устройству	Динамическая привязка инструкций к исполняющему устройству
Бинарная несовместимость	Бинарная совместимость

Intel Itanium. Бандлы

Бандлы - большие слова, в которые запакованы инструкции.



В **слотах** располагаются инструкции.

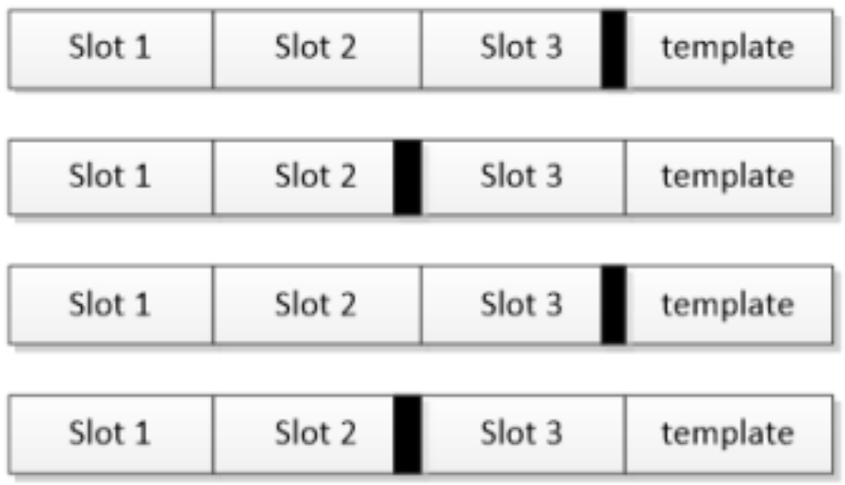
Шаблон определяет тип каждой из инструкций и позицию стопа.

Типы инструкций:

- M - Memory/Move operations
- I - Complex Integer/Multimedia operations
- A - Simple Integer/Logic/Multimedia operations
- F - Floating point operations (Normal/SIMD)
- B - Branch operations
- L - специальные

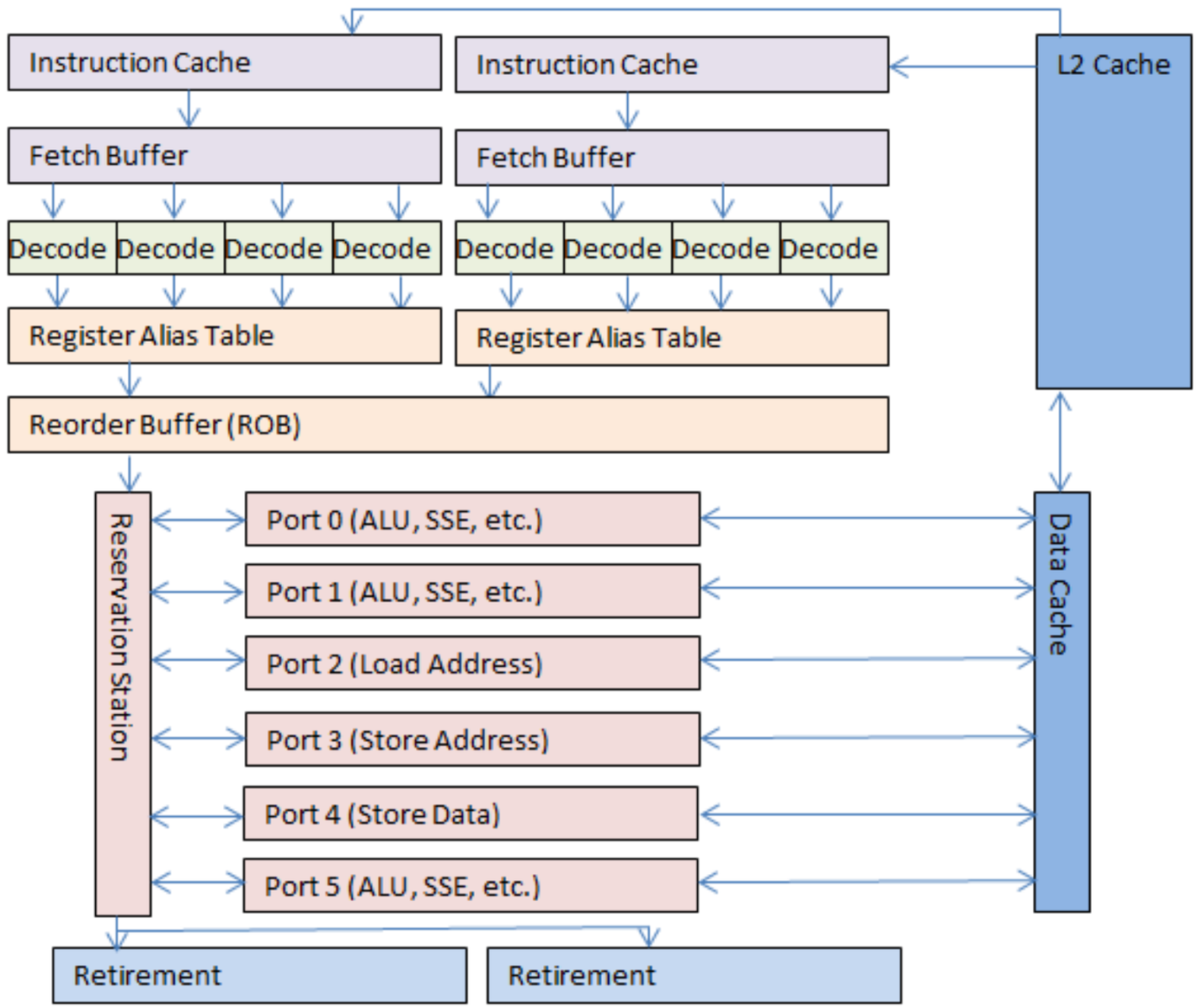
Intel Itanium. Бандлы

Itanium не умеет определять зависимости по данным, это определяется во время компиляции. Стоп указывает процессору, какие инструкции можно выполнить параллельно. Пример бандлов и позиций стопа:



Между стопами инструкции не имеют зависимости по данным, т.е. могут быть выполнены параллельно.

Вычислительный конвейер



Параллелизм в БД. ACID

- **A**tomicity — транзакции атомарны, то есть либо все изменения транзакции фиксируются (commit), либо все откатываются (rollback);

- **C**onsistency — транзакции не нарушают согласованность данных, то есть они переводят базу данных из одного корректного состояния в другое. Тут можно упомянуть допустимые значения полей, внешние ключи и более сложные ограничения целостности;

- **I**solation — работающие одновременно транзакции не влияют друг на друга, то есть многопоточная обработка транзакций производится таким образом, чтобы результат их параллельного исполнения соответствовал результату их последовательного исполнения (сериализация транзакций);

- **D**urability — если транзакция была успешно завершена, никакое внешнее событие не должно привести к потере совершенных ей изменений.

Параллелизм в БД. MVCC

В обычной реализации чтения из базы блокируют запись, и наоборот. Неэффективно при большом количестве одновременных транзакций.

MVCC (MultiVersion Concurrency Control) - механизм параллельного доступа из множества транзакций к данным через создание множества версий изменяемых данных.

Каждая версия данных имеет идентификатор *TS(Ti)* - timestamp или последовательный ID транзакции *Ti*, изменившей данные.

Если транзакция *Ti* читает данные, то она берет версию с наибольшим *TS(Tj)* таким, что *TS(Ti) > TS(Tj)*.

Если транзакция *Ti* хочет изменить данные, и существует другая транзакция *Tk*, которая хочет изменить данные, то должно быть удовлетворено условие *TS(Ti) < TS(Tk)*, чтобы обновление данных транзакции *Ti* прошло.

Параллелизм в БД. Ссылки

- http://en.wikipedia.org/wiki/Multiversion_concurrency_control

- http://www.postgresql.org/docs/9.4/interactive/mvcc.html