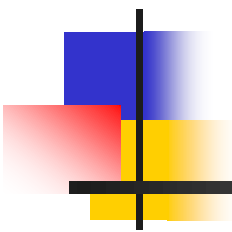


Методы ускорения программ и алгоритмов



Илья Цветков

Video Group
CS MSU Graphics & Media Lab



Содержание

- Введение
 - Тестирование производительности
- Архитектура современных процессоров
 - Особенности
 - Типичное устройство
 - Кэш
- Оптимизация
 - Выбор алгоритмов
 - Ветвления
 - Память
 - Циклы
 - Медленные операции
 - Вещественные вычисления
- SIMD
- Многопоточность

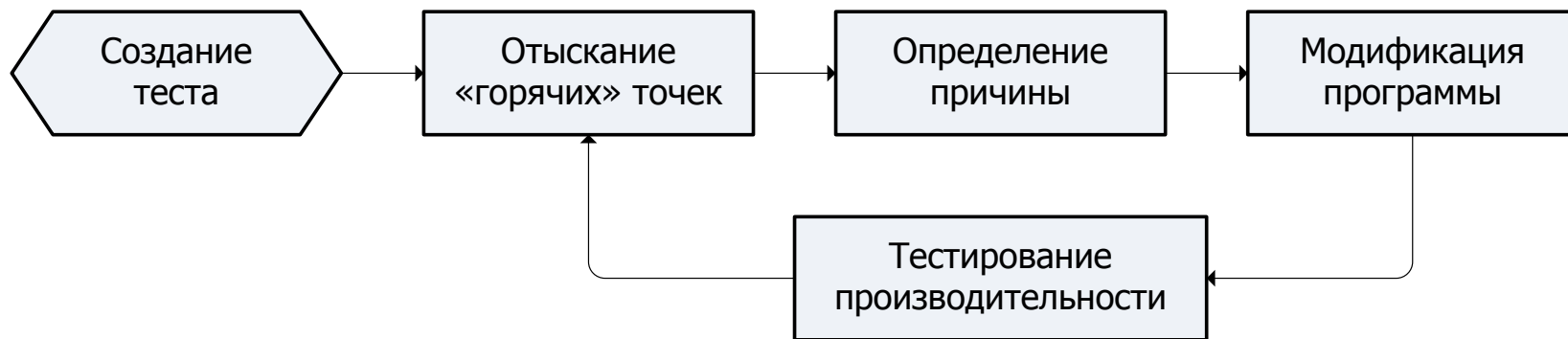
Тезисы оптимизации

- Нельзя оптимизировать программу/функцию не запуская ее
- Необходимо тестирование на различных машинах
- Один из возможных путей оптимизации — удаление дополнительных возможностей
- Отладочные сборки программы не пригодны для тестирования производительности
- Оптимизация далеко не всегда требует написание ассемблерного кода
- Тестирование должно быть частью процесса разработки

Методы ускорения

- Оптимизация с учетом особенной конкретной архитектуры
 - Векторные инструкции
- Распараллеливание задачи
 - Hyper-Threading
 - Многоядерные процессоры
- Применение специализированных процессоров
 - GPU
 - nVidia CUDA, Intel Larabee

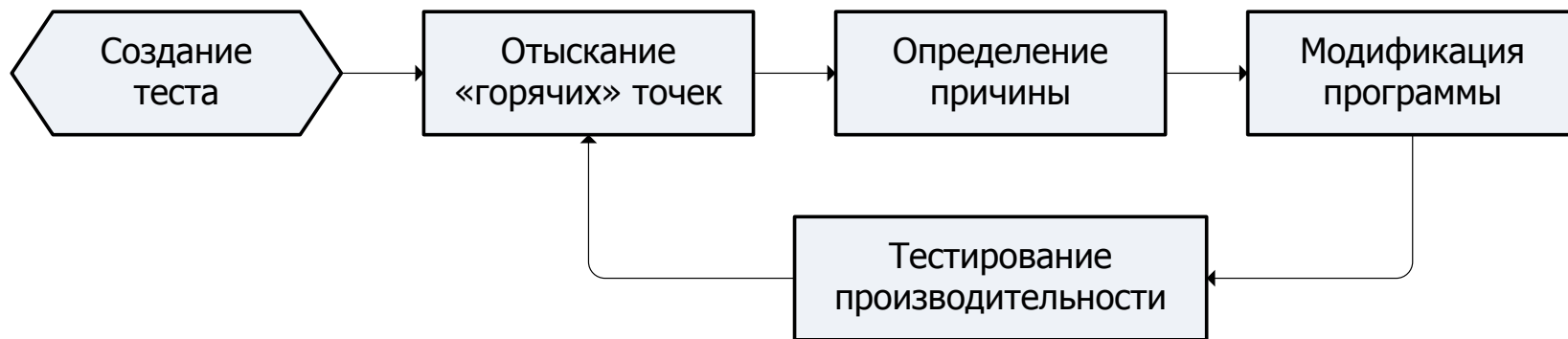
Процесс оптимизации



«Горячие» точки (hotspots):

- места наибольшей активности программы или функции;
- под активностью может пониматься:
 - время, затрачиваемое на исполнение некоторого участка кода,
 - количество неверно предсказанных ветвлений,
 - количество промахов кэша;
- показывают области требующие оптимизации.

Процесс оптимизации



Инструментарий:

- средства измерения времени исполнения программы/функции;
- программный профайлер:
 - sampling,
 - instrumenting;
- оптимизирующий компилятор.

Тестирование производительности



Свойства теста производительности:

- Повторяемость результатов
- Отражение характерного поведения программы
- Простота использования
- Адекватность результатов
- Совмещение с тестирование функциональности
- Измерение времени исполнения
- Полное покрытие программы
- Точность измерений

Измерение времени исполнения



Таймер	Точность	Максимум	Пример использования
Функции библиотеки языка C	±1 с	73 года	<pre>time_t start, elapsed; start = time(NULL); // некоторый код elapsed = time(NULL) - start;</pre>
Windows multimedia timer	±10 мс	49 дней	<pre>DWORD start, elapsed; start = timeGetTime(); // некоторый код elapsed = timeGetTime() - start;</pre>

Измерение времени исполнения



Таймер	Точность	Максимум	Пример использования
Такты CPU (32 бита)	± 1 нс (1 ГГц)	4,29 с	<pre>DWORD start, elapsed; _asm { RDTSC MOV start, eax } // некоторый код _asm { RDTSC SUB eax, start MOV elapsed, eax }</pre>
Такты CPU (64 бита)	± 1 нс (1 ГГц)	580 лет	<pre>LARGE_INTEGER start, end; __int64 elapsed; QueryPerformanceCounter(&start); // некоторый код QueryPerformanceCounter(&end); elapsed = end.QuadPart - start.QuadPart;</pre>



Outline

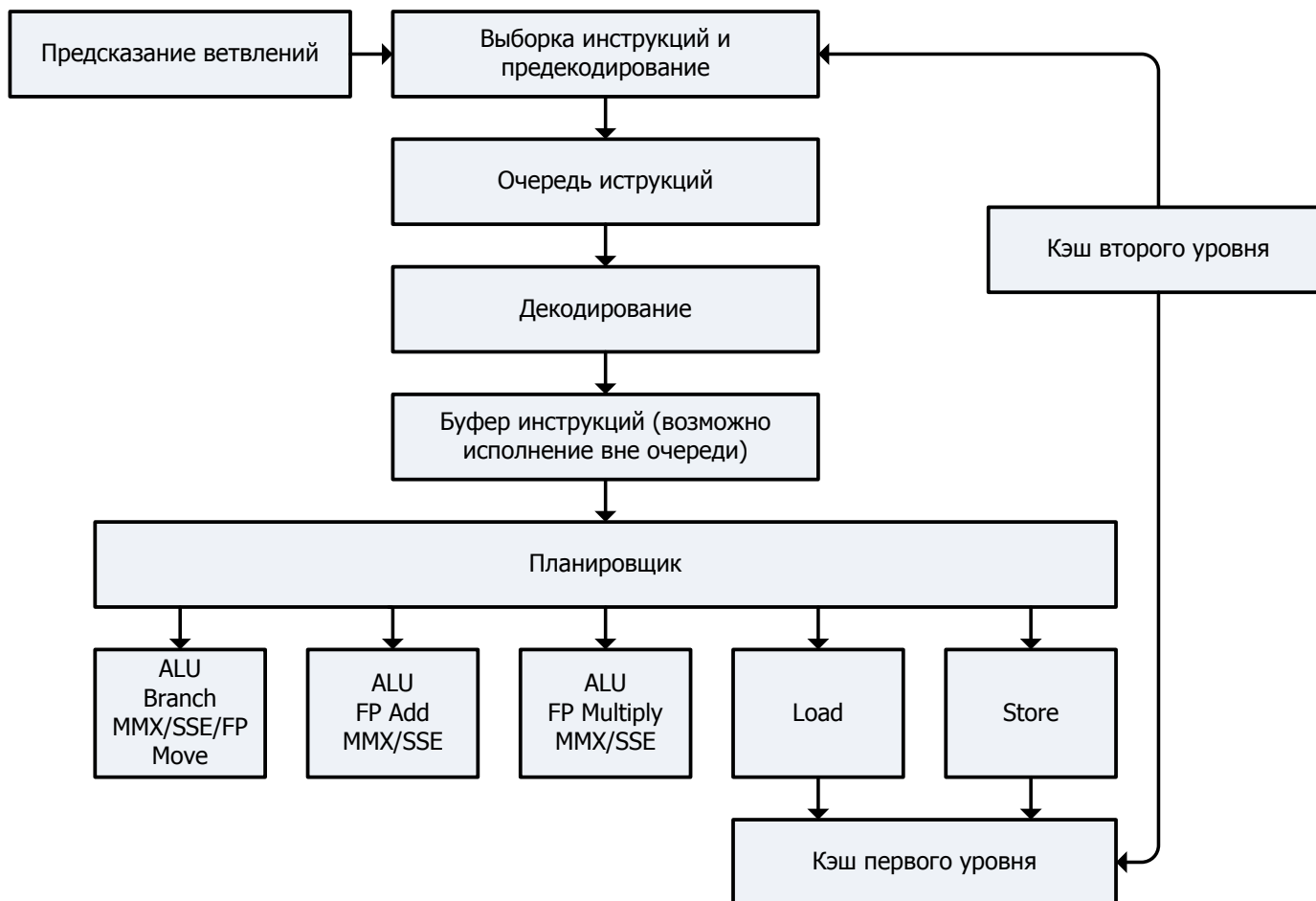
- Введение
 - Тестирование производительности
- Архитектура современных процессоров
 - Особенности
 - Типичное устройство
 - Кэш
- Оптимизация
 - Выбор алгоритмов
 - Ветвления
 - Память
 - Циклы
 - Медленные операции
 - Вещественные вычисления
- SIMD
- Многопоточность

Особенности архитектуры современных процессоров



- Конвейерная обработка инструкций
- CISC на уровне машинных инструкций, но RISC на уровне микроархитектуры
- Суперскалярная архитектура
- Выполнение команд вне очереди (out-of-order execution)
- Большое время доступа к основной памяти и наличие быстродействующего кэша
- Наличие набора векторных инструкций
- Многоядерность

Схема процессора



Архитектура многоядерного процессора



Hyper-Threading

Architectural state	Architectural state
Execution engine	
Local APIC	Local APIC
Кэш второго уровня	
Интерфейс с шиной данных	

Intel Core Duo, Core 2 Duo

Architectural state	Architectural state
Execution engine	Execution engine
Local APIC	Local APIC
Кэш второго уровня	
Интерфейс с шиной данных	

Intel Core 2 Quad

Architectural state	Architectural state	Architectural state	Architectural state
Execution engine	Execution engine	Execution engine	Execution engine
Local APIC	Local APIC	Local APIC	Local APIC
Кэш второго уровня		Кэш второго уровня	
Интерфейс с шиной данных		Интерфейс с шиной данных	

Устройство кэша L1



- Размер строки — 64 байта
- Pentium 4, Xeon
 - объем кэша — 8—16 КБ
 - 4—8 каналов, 32 ряда
- Core, Pentium M
 - объем кэша — 32 КБ
 - 8 каналов, 64 ряда

Производительность инструкций



Инструкция	Латентность	Пропускная способность
Сложение, вычитание, логические операции (AND, OR, XOR), сравнение, переход, доступ к памяти	0,5	0,5
Работа со стеком, логические сдвиги, SIMD-доступ к памяти, 64-битные целочисленные SIMD-операции (MMX, за исключением EMMS); модуль, сравнение, сложение и вычитание для вещественных чисел	1	1
128-битные целочисленные SIMD-операции; SIMD-операции над вещественными числами одинарной и двойной точности за исключением деления и корня	2	2
Целочисленное умножение	15	4
Целочисленное деление; деление и квадратный корень для вещественных чисел одинарной точности (32 бита)	23	23

Производительность инструкций



Инструкция	Латентность	Пропускная способность
SIMD деление и квадратный корень для вещественных чисел одинарной точности	32	32
Деление и квадратный корень для вещественных чисел двойной точности (64 бита)	38	38
Деление и квадратный корень для вещественных чисел повышенной точности (80 бит)	43	43
SIMD деление и квадратный корень для вещественных чисел двойной точности	62	62
Трансцендентные функции (синус, косинус, тангенс, арктангенс)	130—170	130—170



Outline

- Введение
 - Тестирование производительности
- Архитектура современных процессоров
 - Особенности
 - Типичное устройство
 - Кэш
- Оптимизация
 - Выбор алгоритмов
 - Ветвления
 - Память
 - Циклы
 - Медленные операции
 - Вещественные вычисления
- SIMD
- Многопоточность



Алгоритмы

Основные характеристики алгоритмов:

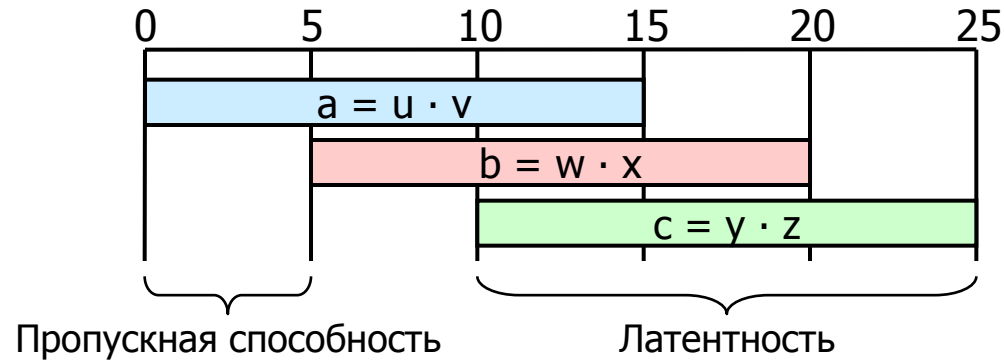
- Вычислительная сложность
- Инструкции, требуемые для реализации
- Объем используемой памяти
- Характер доступа к памяти
- Зависимости по данным

Зависимости по данным и параллелизм инструкций



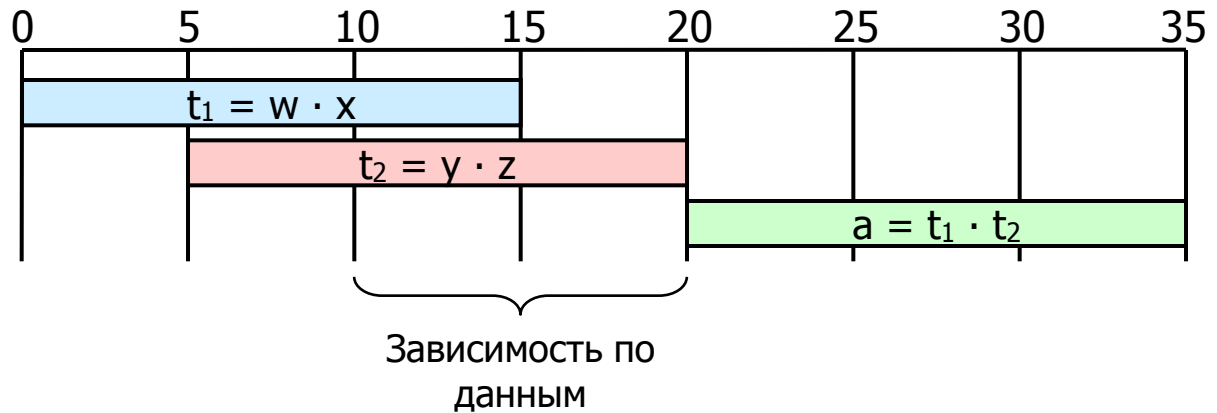
Пример 1:

```
a = u * v;  
b = w * x;  
c = y * z;
```



Пример 2:

```
a = (w * x) * (y * z);
```



Предсказание условных переходов



- Предсказываемый процессором условный переход:

```
for (int i = 0; i < 100; ++i) {  
    if (i % 2 == 0)  
        doEven();  
    else  
        doodd();  
}
```

- Переход случаен — сложно предсказуем:

```
for (int i = 0; i < 100; ++i) {  
    side = flipCoin();  
    if (side == HEADS)  
        ++numHeads;  
    else  
        ++numTails;  
}
```

Виды переходов

- Условные переходы, исполняемые впервые
 - Переходы вперед
 - Переходы назад
- Условные переходы, которые уже исполнялись
- Вызов функции и возврат из функции
- Косвенные вызовы функций и переходы (указатели на функции и таблицы переходов)
- Безусловные переходы

Удаление ветвлений с помощью инструкции CMOV

- Код использующий условный переход
 - На C

```
if (val > 255)
    val = 255;
```
 - Аналог на ассемблере

```
cmp val, 255
jle skipset
mov val, 255
skipset:
```
- Код использующий инструкцию CMOV

```
mov    eax, val
mov    ebx, 255
cmp    eax, ebx
cmovg eax, ebx
mov    val, eax
```

 - Быстрее в 2,5 раза в случае случайного перехода
 - Эквивалентны при 1 неверно предсказанном переходе из 100

Удаление ветвлений при использовании SIMD



- Использование бинарных масок

val: const: mask:

240	258	130	262
-----	-----	-----	-----

 >

255	255	255	255
-----	-----	-----	-----

 =

FFFF	FFFF	FFFF	FFFF
------	------	------	------

result = (val & ~mask) | (const & mask);

- Вычисление минимума и максимума

min(val: const: result:

240	258	130	262
-----	-----	-----	-----

 ,

255	255	255	255
-----	-----	-----	-----

) =

240	255	130	255
-----	-----	-----	-----

Удаление ветвлений

```
for (int i = 0; i < size; ++i) {  
    alpha = getAlpha(src[i]);  
    if (alpha == 255)  
        dst[i] = src[i];  
    else if (alpha != 0)  
        dst[i] = blend(src[i], dst[i], alpha);  
}
```


Таблицы переходов

```
switch (val) {  
    case 'A':  
        // вероятность 95%  
        break;  
    case 'B':  
        // вероятность 4%  
        break;  
    case 'C':  
        // вероятность 0,5%  
        break;  
    case 'D':  
        // вероятность 0,5%  
        break;  
}
```

```
if (val == 'A') {  
    // ...  
}  
else if (val == 'B') {  
    // ...  
}  
else switch (val) {  
    case 'C':  
        // ...  
        break;  
    case 'D':  
        // ...  
        break;  
}
```



Использование кэша

- Обязательная загрузка (compulsory load)
- Загрузка из-за нехватки объема кэша (capacity load)
- Загрузка вследствие конфликта (conflict load)
- Эффективность кэша (cache efficiency)

Программная предвыборка



Ассемблерная инструкция	Описание
PREFETCHNTA	Выборка данных для единичного доступа в режиме чтения
PREFETCH0	Выборка в кэши всех уровней для чтения/записи
PREFETCH1	Выборка в кэши L2 и L3, но не L1
PREFETCH2	Выборка только в кэш третьего уровня

Пример с использованием встраиваемых функций:

```
for (int i = 0; i < size; ++i) {  
    array[i] = fn(array[i]);  
    _mm_prefetch(array[i + 16], MM_HINT_T0);  
}
```

Выравнивание данных

- Доступ к данным, попадающим в различные строки кэша вызывает дополнительные накладные расходы
- Большинство современных векторных инструкций не работают с невыровненными данными

Тип данных	Выравнивание
1 байт, 8 бит, BYTE	Произвольное выравнивание
2 байта, 16 бит, WORD	2 байта
4 байта, 32 бита, DWORD	4 байта
8 байт, 64 бита, QWORD	8 байт
10 байт, 80 бит, (вещественное число расширенной точности)	16 байт
16 байт, 128 бит	16 байт

Оптимизация использования памяти



- Применение алгоритмов требующих меньше количество памяти
- Повышение эффективности использования кэша
- Предварительная загрузка данных в кэш
- Применение инструкций, позволяющих избежать накладных расходов RFO
- Устранение конфликтов кэша
- Устранение проблем связанных с нехваткой объема кэша
- Увеличение работы проводимых над одними данным



Циклы

Достоинства

Требуют меньшее количество инструкций

Нет необходимости повторно декодировать инструкции

Процессор может автоматически разворачивать циклы

Компилятор может оптимизировать циклы за счет использования SIMD-инструкций

Недостатки

Привносят накладные расходы в виде инструкций, реализующих цикл

Добавляется один неверно предсказанный переход

Циклы вносят дополнительные зависимости по данным

Без SIMD-оптимизации вычисления, которые могли бы выполняться параллельно, выполняются последовательно

Разворачивание циклов

Исходный цикл:

```
sum = 0;
for (int i = 0; i < 16; ++i)
    sum += array[i];
```

Вариант 1:

```
sum = 0;
for (int i = 0; i < 16; i += 4) {
    sum += array[i];
    sum += array[i + 1];
    sum += array[i + 2];
    sum += array[i + 3];
}
```

Вариант 2:

```
a = b = c = d = 0;
for (int i = 0; i < 16; i += 4) {
    a += array[i];
    b += array[i + 1];
    c += array[i + 2];
    d += array[i + 3];
}
sum = a + b + c + d;
```

Разворачивание циклов



Инварианты цикла

- Инвариантные арифметические выражения

```
for (int x = 0; x < length; ++x)
    array[x] = x * val / 3;
```

- Инвариантные вызовы функций

```
for (int x = 0; x < length; ++x)
    array[x] += x * val * foo(y) / 3;
```

- Инвариантные ветвления

```
for (int i = 0; i < size; ++i)
    if (blend == 255)
        dest[i] = src1[i];
    else if (blend == 0)
        dest[i] = src2[i];
    else
        dest[i] = (src1[i] * blend +
                  src2[i] * (255 - blend)) / 255;
```

Loop fusion

Исходный код:

```
for (int i = 0; i < length; ++i)
    x[i] = a[i] + b[i];
for (int i = 0; i < length; ++i)
    y[i] = b[i] + c[i];
```

Преобразованный вариант:

```
for (int i = 0; i < length; ++i) {
    x[i] = a[i] + b[i];
    y[i] = b[i] + c[i];
}
```

Unroll and jam

```

for (int i = 0; i < h; ++i) {
    t = c[i];
    for (int j = 0; j < w; ++j) {
        t += a[i][j] * b[j];
    }
    c[i] = t;
}

```

```

for (int i = 0; i < h / n; i += n) {
    for (int j = 0; j < w; ++j) {
        c[i] += a[i][j] * b[j];
    }
    for (int j = 0; j < w; ++j) {
        c[i + 1] += a[i + 1][j] * b[j];
    }
    // ...
    for (int j = 0; j < w; ++j) {
        c[i + n - 1] +=
            a[i + n - 1][j] * b[j];
    }
}

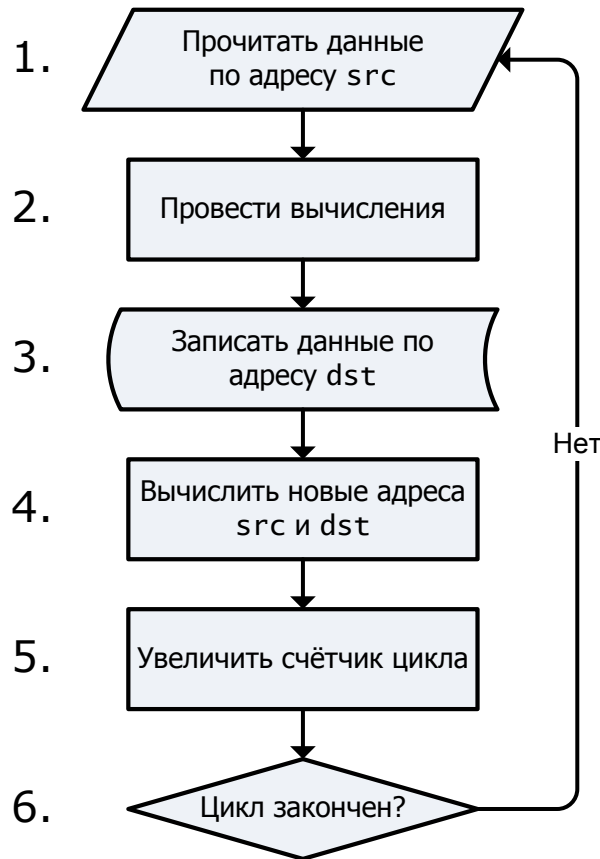
```

```

for (int i = 0; i < h / n; i += n) {
    for (int j = 0; j < w; ++j) {
        c[i] += a[i][j] * b[j];
        c[i + 1] += a[i + 1][j] * b[j];
        // ...
        c[i + n - 1] +=
            a[i + n - 1][j] * b[j];
    }
}

```

Memory Address Dependencies



■ **ТИПИЧНЫЙ ЦИКЛ:**

```
for (i = 0; i < n; ++i) {  
    *dst = f(*src);  
    ++dst;  
    ++src;  
}
```

■ **Проблемы:**

- Неизвестный адрес для загрузки данных
- Неизвестный адрес для сохранения данных



Медленные инструкции

- Инструкции с большой задержкой
- Инструкции с маленькой пропускной способностью
- Ожидание готовности операндов
- Отсутствие свободных слотов выполнения
- Инструкции, останавливающие исполнение вне очереди



Lookup Tables

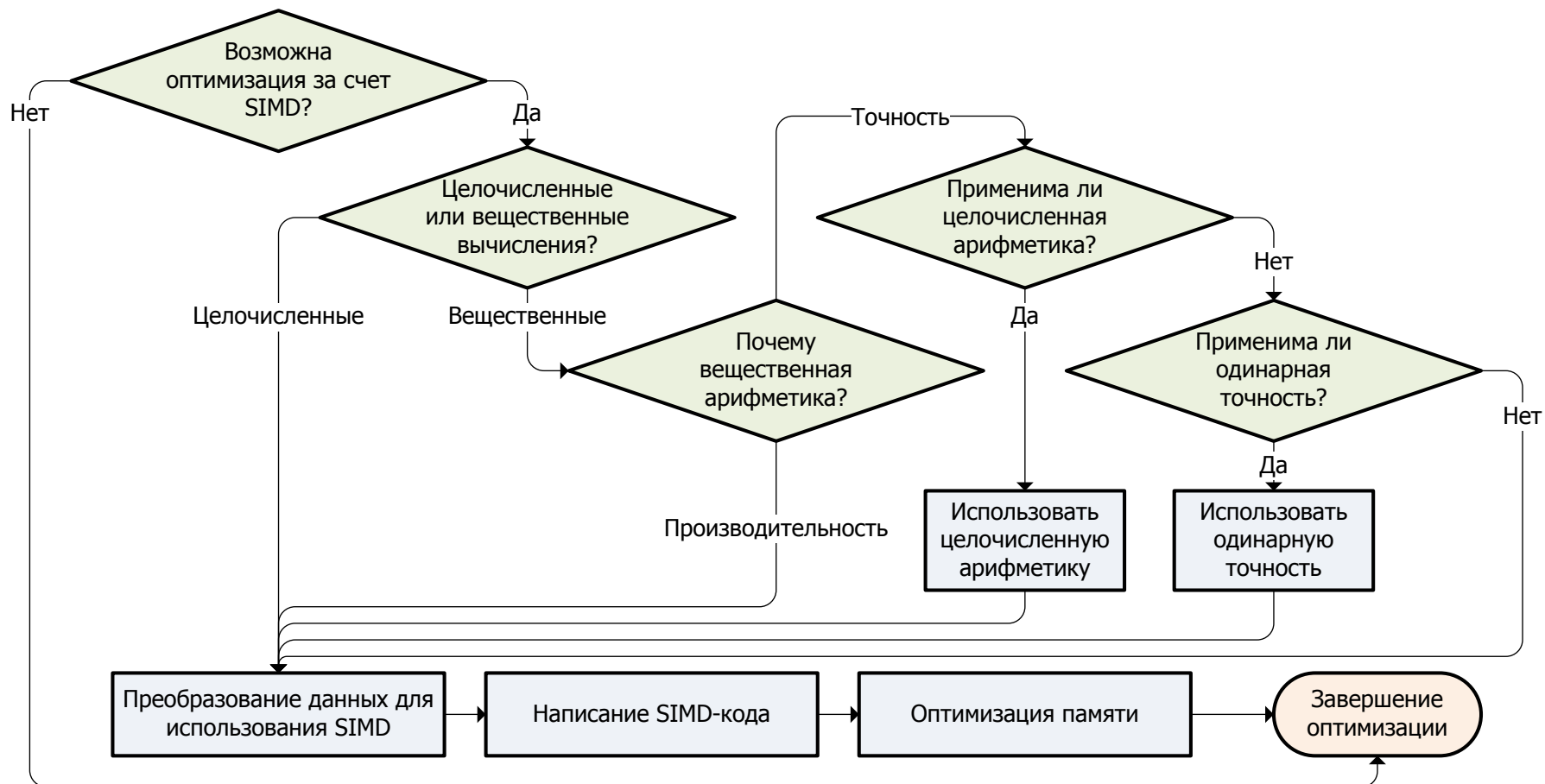
- Lookup-таблица должна быть организована так, чтобы полностью находиться в кэше
- Таблица должна быть как можно меньшего размера
- Таблица должна заменять как можно больше вычислений
- Пример: преобразование ARGB→AYYY

Floating point

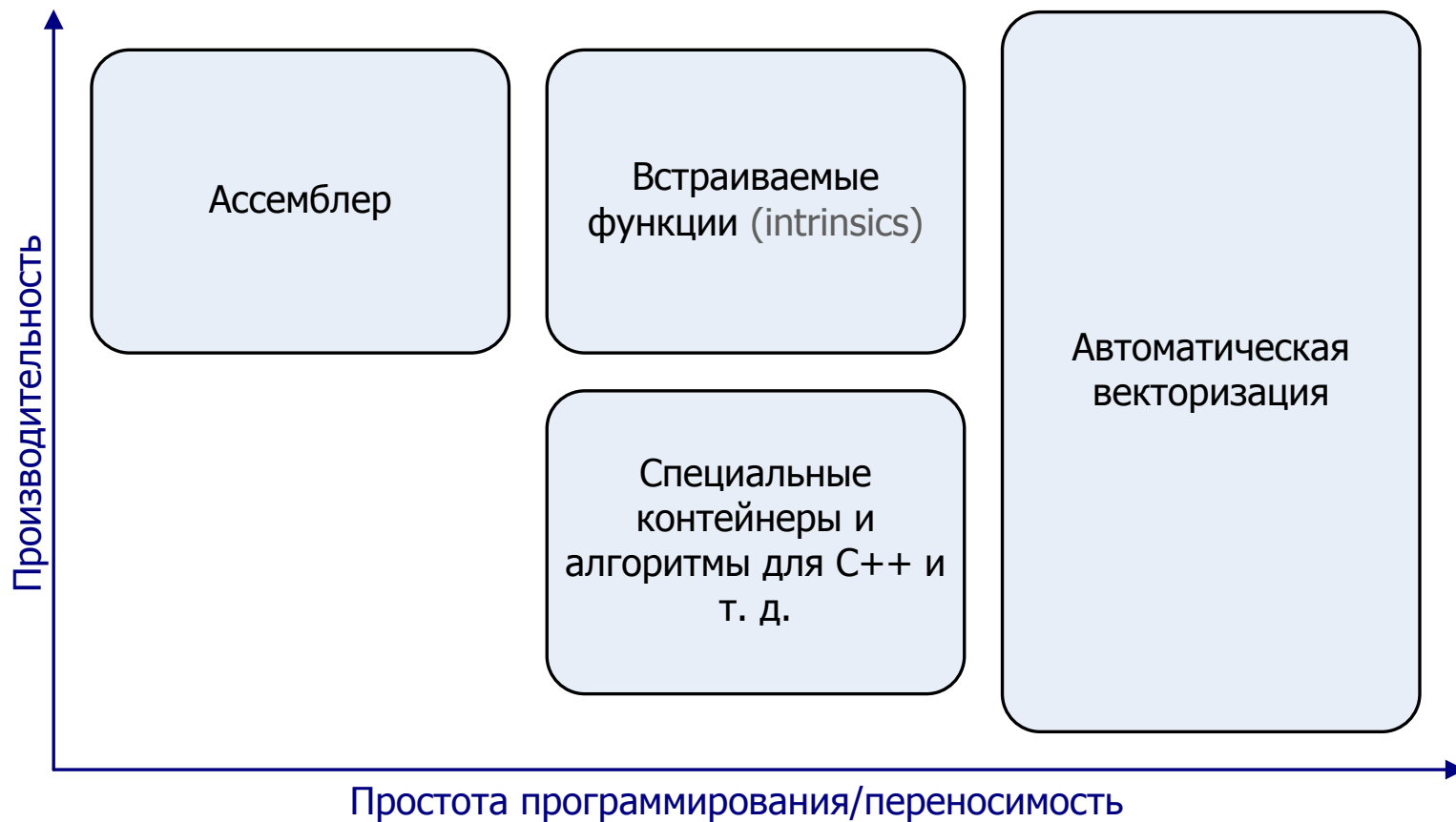
Точность	Размер, бит	Диапазон	Производительность деления в тактах		
			FDIV	DIVSS/DIVSD	DIVPS/DIVPD
Single	32	$1,18 \cdot 10^{-38}$ — $3,4 \cdot 10^{38}$	23	22	32
Double	64	$2,23 \cdot 10^{-308}$ — $1,79 \cdot 10^{308}$	38	35	62
Extended	80	$3,37 \cdot 10^{-4932}$ — $1,18 \cdot 10^{4932}$	43	—	—

- Режимы приведения к целому:
 - Округление к ближайшему целому
 - Округление «вниз»
 - Округление «вверх»
 - Отбрасывание дробной части

SIMD



Использование SIMD-инструкций



MMX, SSE, SSE2

- MMX
 - 64-битные регистры
 - Работа с упакованными байтами, словами и двойными словами
- SSE (Streaming SIMD Extensions)
 - 128-битные регистры
 - Работа с упакованными вещественными числами одинарной точности
 - Управление кэшированием
 - Упакованные 64-битные целые
- SSE2
 - Работа с упакованными вещественными числами двойной точности
 - Аналоги инструкций MMX для 128-битных регистров
 - «Перемешивание» данных

SSE3, SSSE3, SSE4.1

- SSE3
 - «Горизонтальные» вычисления
 - Преобразование к целому (x87)
 - Поддержка синхронизации потоков
- SSSE3 (Supplemental Streaming SIMD Extensions 3)
 - Вычисление модуля
 - Побайтовое «перемешивание»
- SSE4.1
 - Скалярное произведение
 - SAD

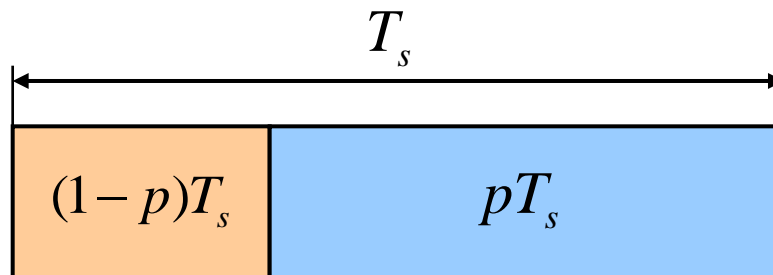


Outline

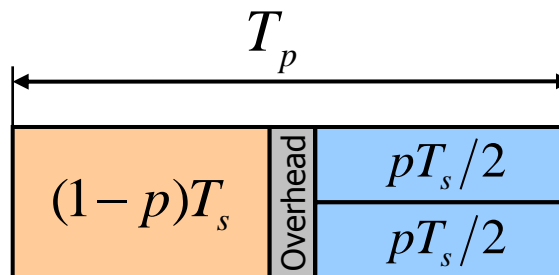
- Введение
 - Тестирование производительности
- Архитектура современных процессоров
 - Особенности
 - Типичное устройство
 - Кэш
- Оптимизация
 - Выбор алгоритмов
 - Ветвления
 - Память
 - Циклы
 - Медленные операции
 - Вещественные вычисления
- SIMD
- Многопоточность

МНОГОПОТОЧНОСТЬ

Последовательное выполнение



Параллельное выполнение



$$T_p = (1-p)T_s + \frac{pT_s}{N} + c$$

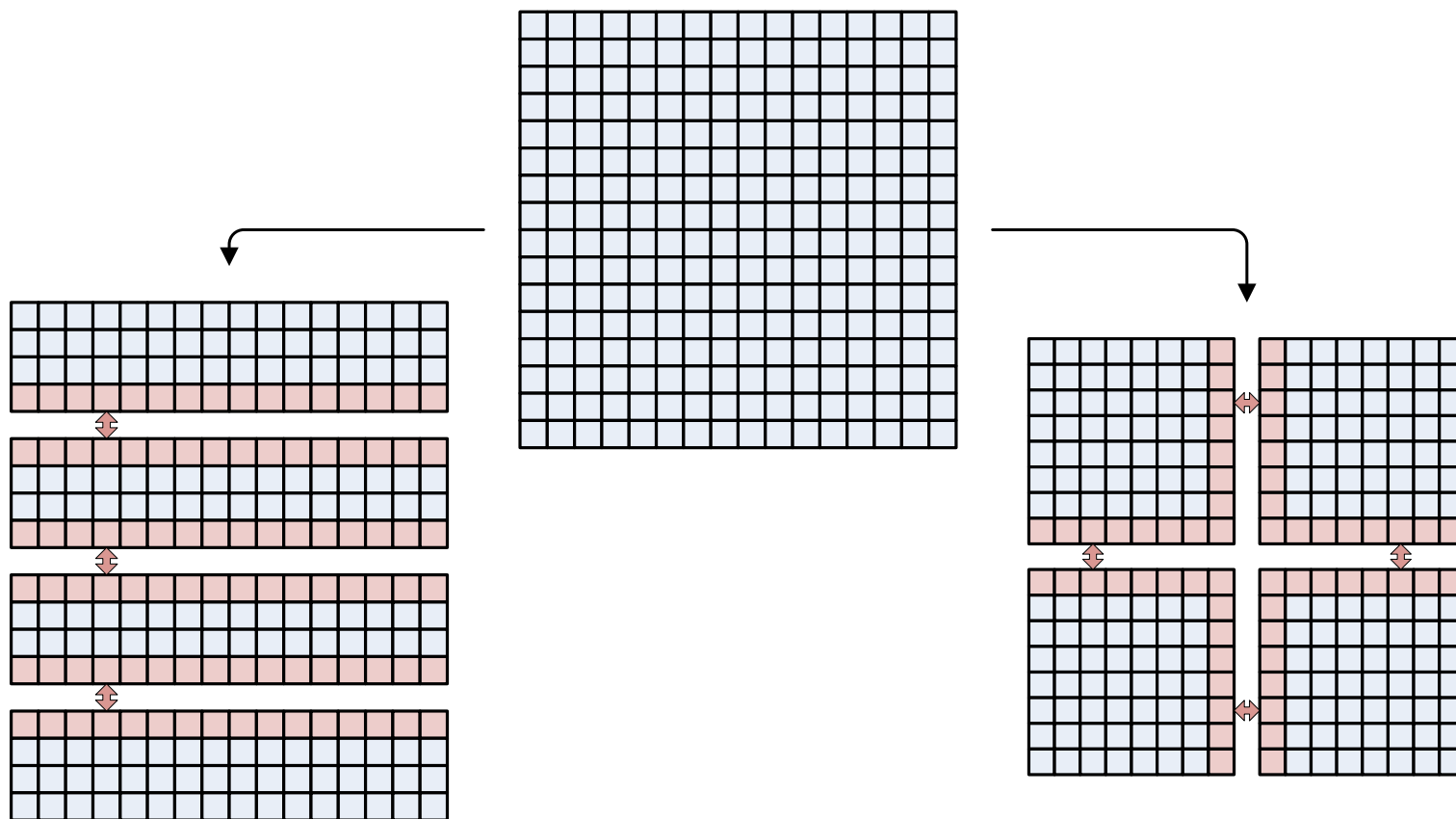
$$\frac{T_p}{T_s} \approx 1 - p + \frac{p}{N}$$



МНОГОПОТОЧНОСТЬ

- Идеи применения многопоточности
 - Распараллеливание задач, занимающих существенное время
 - Сбалансированность потоков
 - Синхронизация
 - Минимизация объема «общей» памяти
 - Определение количества потоков
- Методы распараллеливания задач
 - Functional decomposition
 - Domain decomposition

Domain decomposition



Основные проблемы

- Накладные расходы на управление потоками
- Короткие циклы
- False-sharing
- Пропускная способность памяти
- Эффективность использования кэша
- Накладные расходы на синхронизацию
 - Spin-waits
- Processor affinity
 - Cache ping-pong

OpenMP

- Исходный вариант

```
void serialApplyFoo(float a[], size_t n) {  
    for (size_t i = 0; i < n; ++i)  
        foo(a[i]);  
}
```

- Использование OpenMP

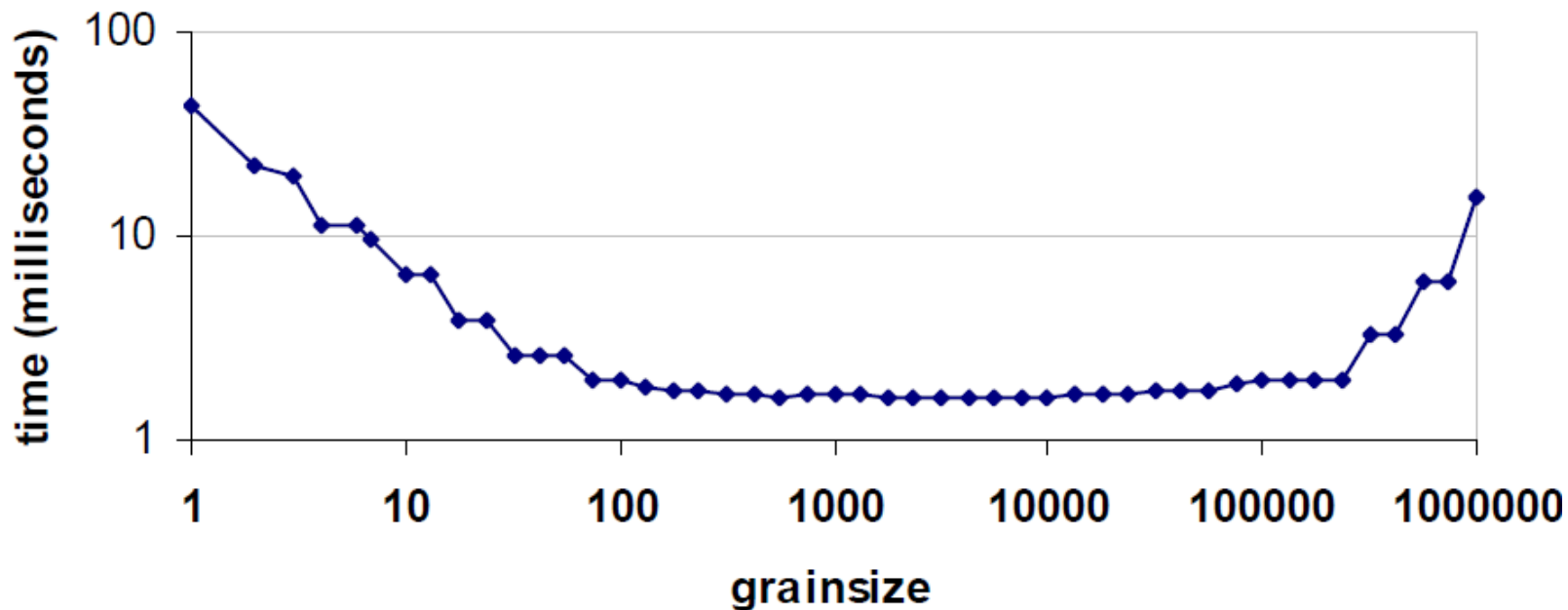
```
void parallelApplyFoo(float a[], size_t n) {  
    omp_set_num_threads(NUM_THREADS);  
    #pragma omp parallel for shared(a)  
    for (size_t i = 0; i < n; ++i)  
        foo(a[i]);  
}
```

Threading Building Blocks

```
class ApplyFoo {
    float *const m_a;
public:
    void operator() (const blocked_range<size_t> &r) const {
        float *a = m_a;
        for (size_t i = r.begin(); i != r.end(); ++i)
            foo(a[i]);
    }
    ApplyFoo(float a[]) : m_a(a) {}
};

void parallelApplyFoo(float a[], size_t n) {
    parallel_for(blocked_range<size_t>(0, n, grainSize),
        ApplyFoo(a));
}
```

Способ разбиения



```
void parallelApplyFoo(float a[], size_t n) {  
    parallel_for(blocked_range<size_t>(0, n), ApplyFoo(a),  
                auto_partitioner());  
}
```

Список литературы

- *The Software Optimization Cookbook*, Intel Press, 2002
- *Concepts of High Performance Computing*, Georg Hager, Gerhard Wellein, 2008
- *Intel 64 and IA-32 Architectures Optimization Reference Manual*, 2007,
<http://developer.intel.com/design/processor/manuals/248966.pdf>
- Intel Threading Building Blocks,
<http://threadingbuildingblocks.org/>
- OpenMP, <http://www.openmp.org/>