Новосибирский государственный университет
Факультет информационных технологий
Кафедра параллельных вычислений

Эффективное программирование современных микропроцессоров и мультипроцессоров

Введение. Базовые способы оптимизации

Преподаватели:
Киреев С.Е.
Калгин К.В.

Цели курса

- Изучить архитектурные особенности микропроцессоров, как их использовать при создании программ
- Изучить способы оптимизации программ с учётом архитектуры, получить навыки оптимизации
- Изучить средства анализа производительности программ, получить навыки их использования

Обзор курса

- Введение
- Базовые способы оптимизации
- Оптимизация работы с данными
- Оптимизация использования многоядерных систем с общей памятью
- Оптимизация управляющих конструкций
- Векторизация вычислений
- Возможности компиляторов по оптимизации программ
- Профилирование программ
- Моделирование производительности программ
- Оценочное тестирование программ и оборудования

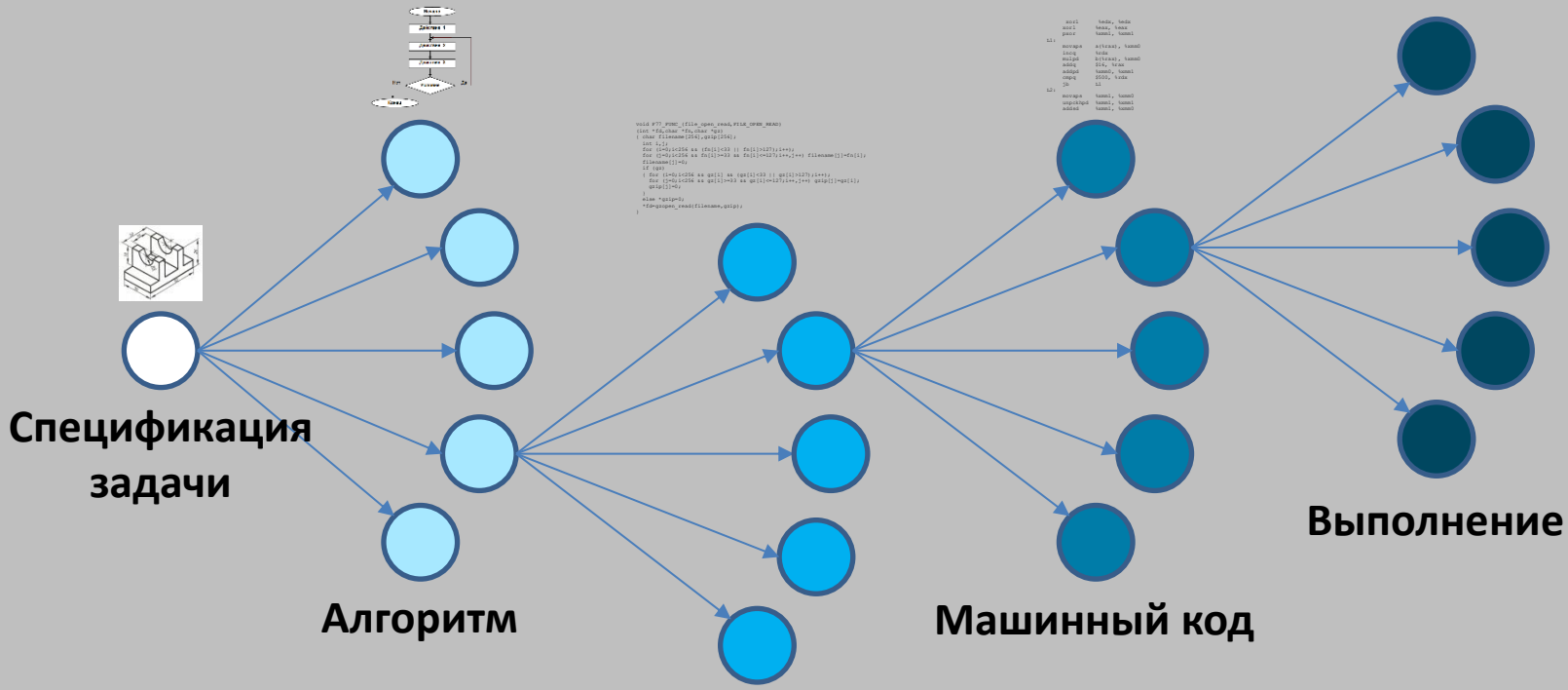
ВВЕДЕНИЕ

Оптимизация программ

- Оптимизация – модификация системы для улучшения её эффективности.

- Цели оптимизации программ:
  1. **Ускорение работы программы**
  2. Уменьшение объёма ресурсов, требующейся для работы программы или потребляющихся в ходе работы программы (память, эл.энергия, пропускная способность сети, …)
  3. Уменьшение побочных явлений (тепловыделение, …)
  4. Уменьшение размера кода программы

Оптимизация программ



Человек — Алгоритмическая оптимизация

Человек — Программная оптимизация

Компилятор — Программная оптимизация

Процессор — Аппаратная оптимизация

Виды оптимизации

По стадии оптимизации:

- Алгоритмическая
- Программная
- Аппаратная

По области применения:

- Общая
- Зависимая от окружения
  - архитектуры
  - компилятора
  - библиотек
  - …

Что оптимизировать?

- **Правило 80/20**: 80% времени работают 20% кода программы
- **Hot spot** – место в программе, работа которого занимает наибольшую долю времени (обычно это цикл).
  - Оптимизация hot spots даёт наибольший выигрыш.
  - Как их найти? Путём профилирования

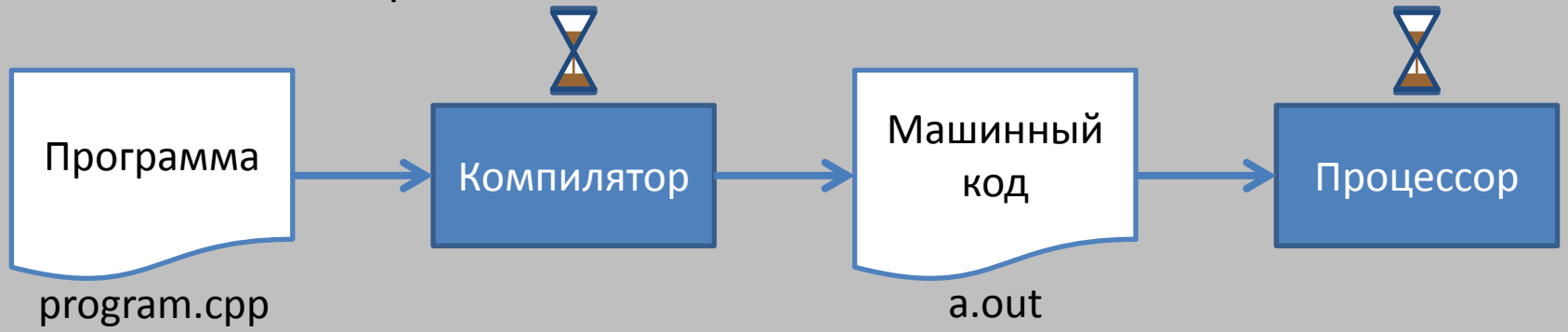
БАЗОВЫЕ СПОСОБЫ ОПТИМИЗАЦИИ

Базовые способы оптимизации

- Устранение «ненужных» вычислений
- Замена «долгих» вычислений на «быстрые»

Устранение «ненужных» вычислений

- «Ненужные» вычисления:
  - Вычисления, результат которых не используется
  - Повторные вычисления
    - Запоминание результата вычислений (мемоизация)
    - Вынос инвариантных подвыражений за скобки, из цикла
    - …
  - Вычисления, которые могут быть сделаны на этапе компиляции



Устранение «ненужных» вычислений

- Примеры

x = func1(); *// результат не используется*
x = func2();
use(x);
	x = func2();
	use(x);

x = func(); *// результат может не понадобиться*
if (y) use(x);
	if (y)
	{ x = func();
	use(x);
	}

// избыточные вычисления x = c*(a+b) + d*(a+b);	x = (c+d) * (a+b);

// избыточные вычисления
for (i=0; i<n; i++)
x[i] = a*b + i;
	ab = a*b;
	for (i=0; i<n; i++)
	x[i] = ab + i;

// избыточное управление
for (i=0; i<n; i++)
if (a>b) work_a(i);
else work_b(i);
	if (a>b)
	for (i=0; i<n; i++) work_a(i);
	else
	for (i=0; i<n; i++) work_b(i);

Устранение «ненужных» вычислений

- Примеры

y = 2.0 * x / 3.0;	*// компилятор вычислит константу* y = (2.0 / 3.0) * x;

- Lookup table (Agner Fog, p.132)

Устранение «ненужных» вычислений

- Что может делать компилятор
  - Устранение операций, результат которых не используется
  - Предвычисление константных выражений
  - Вынос общих подвыражений
  - …
  - Не имеет права делать преобразования с вещественными числами, изменяющие порядок операций
    - Разрешается специальным ключом компилятора

Замена «долгих» вычислений на «быстрые»

- Применяется на всех уровнях программирования:
  - Программы, подпрограммы, блоки кода, машинные операции
- На уровне машинных операций:
  - Латентности разных операций процессора сильно отличаются
    - (см. Intel64 and IA-32 Architectures Optimization Reference Manual, Software Optimization Guide for AMD Family ### Processors, …)
- Примеры
  - Замена умножений на сложения и вычитания
  - Замена делений на умножения
  - Замена обращения в память на быстрое вычисление (или наоборот)
  - Понижение точности вещественных вычислений
  - …

Замена «долгих» вычислений на «быстрые»

- Примеры

for (i=0; i<n; i++) x[i] = (double) i / n;	double r = 1.0 / n; for (i=0; i<n; i++) x[i] = i * r;

for (i=0; i<n; i++) x[i] = i+1;
…
for (i=0; i<n; i++) y[i] = x[i]*2;
	for (i=0; i<n; i++) x[i] = i+1;
	…
	for (i=0; i<n; i++) y[i] = (i+1)*2;

	for (i=0; i<n; i++)
	{ int k = i+1;
	  x[i] = k;
	  y[i] = k*2;
	}
	…

Замена «долгих» вычислений на «быстрые»

- Примеры

#define PI 3.141592653589793 double x[N]; for (i=0; i<N; i++) x[i] = sin(2*PI*i/N);	#define PI 3.14159265f float x[N]; for (i=0; i<N; i++) x[i] = sinf(2*PI*i/N);
	#define PI 3.14159265f #define A (2*PI/N) float x[N]; for (i=0; i<N; i++) x[i] = sinf(A*i);

	#define PI 3.14159265f
	const float a = 2*PI/N;
	float x[N];
	for (i=0; i<N; i++) x[i] = sinf(a*i);

Замена «долгих» вычислений на «быстрые»

- Примеры
  - Проверка границ (Agner Fog, p.134)
  - Проверка на несколько значений (p.135)
  - Оптимизация умножений и делений (p.136)
  - Смешивание float и double (p.140)

#define PI 3.14159265f
const float a = 2*PI/N;
float x[N];
for (i=0; i<N; i++) x[i] = sin(a*i);
	#define PI 3.14159265f
	const float a = 2*PI/N;
	float x[N];
	for (i=0; i<N; i++) x[i] = sinf(a*i);

Замена «долгих» вычислений на «быстрые»

- Что может делать компилятор
  - Выбирает наиболее быстрые реализации операций языка в машинном коде из множества допустимых
  - Может делать множество арифметических преобразований (см. Agner Fog)
  - …
  - Не имеет права делать преобразования с вещественными числами, изменяющие порядок операций
    - Разрешается специальным ключом компилятора

Пример из практики

- Было: 15 делений, Xeon: 95 с,Xeon Phi: 1351 с
  - `double tmp1 = Teta1/(( 1.0-xx-yy )*Rho_10), tmp1__2 = tmp1*tmp1, tmp1__3 = tmp1__2*tmp1;`
  - `double tmp2 = Teta2/(  xx  *Rho_20), tmp2__2 = tmp2*tmp2, tmp2__3 = tmp2__2*tmp2;`
  - `double tmp3 = Teta3/(  yy  *Rho_30), tmp3__2 = tmp3*tmp3, tmp3__3 = tmp3__2*tmp3;`
  - `double t1 = ( P_10 + pi1*delta_s + Kbig1*( Teta1/( 1.0-xx-yy )-Rho_10 )/Rho_10 )*tmp1__2;`
  - `double t2 = ( P_20 + pi2*delta_s + Kbig2*( Teta2/  xx  -Rho_20 )/Rho_20 )*tmp2__2;`
  - `double t3 = ( P_30 + pi3*delta_s + Kbig3*( Teta3/  yy  -Rho_30 )/Rho_30 )*tmp3__2;`
  - `f = t3 - t2;`
  - `g = t3 - t1;`
  - `fx =  2.0*t2/xx + tmp2__3 * Kbig2/xx;`
  - `fy = - 2.0*t3/yy - tmp3__3 * Kbig3/yy;`
  - `gx = - 2.0*t1/( 1.0-xx-yy ) - tmp1__3 * Kbig1 / ( 1.0-xx-yy );`
  - `gy = fy + gx;`
- Стало: 3 деления, Xeon: 40 с,Xeon Phi: 509 с
  - `double rxx = 1.0/xx;`
  - `double ryy = 1.0/yy;`
  - `double rxxyy = 1.0/(1.0 - xx - yy);`
  - `double tmp1 = Teta1_Rho_10*rxxyy, tmp1__2 = tmp1*tmp1, tmp1__3 = tmp1__2*tmp1;`
  - `double tmp2 = Teta2_Rho_20*rxx,  tmp2__2 = tmp2*tmp2, tmp2__3 = tmp2__2*tmp2;`
  - `double tmp3 = Teta3_Rho_30*ryy,  tmp3__2 = tmp3*tmp3, tmp3__3 = tmp3__2*tmp3;`
  - `double t1 = ( P_10 + pi1*delta_s + Kbig1*( tmp1 - 1.0 ) )*tmp1__2;`
  - `double t2 = ( P_20 + pi2*delta_s + Kbig2*( tmp2 - 1.0 ) )*tmp2__2;`
  - `double t3 = ( P_30 + pi3*delta_s + Kbig3*( tmp3 - 1.0 ) )*tmp3__2;`
  - `f = t3 - t2;`
  - `g = t3 - t1;`
  - `fx = (  2.0*t2 + tmp2__3 * Kbig2 ) * rxx;`
  - `fy = ( -2.0*t3 - tmp3__3 * Kbig3 ) * ryy;`
  - `gx = ( -2.0*t1 - tmp1__3 * Kbig1 ) * rxxyy;`
  - `gy = fy + gx;`