

# ***Sistemas de Tempo-real***

## **Aula 10**

# **Otimização de código**

**Técnicas de otimização de código**  
**Ferramentas de *profiling***

**Paulo Pedreiras**  
**DETI/UA**  
**[pbrp@ua.pt](mailto:pbrp@ua.pt)**  
**V1.3, Nov/2013**

Slides parcialmente baseados no módulo “Optimizations” da disciplina “Fundamentals of Embedded Systems”, por Rajeev Ghandy, bem como nos materiais pedagógicos que acompanham o livro “Embedded Systems Design”, por Peter Marwedel

# Agenda

- Técnicas de otimização de código
  - Introdução
  - Técnicas independentes do processador
  - Técnicas dependentes da arquitetura de memória
  - Técnicas dependentes do processador
- *Profiling*
  - *Objetivos e metodologias*
  - *Ferramentas*



# ***Porquê otimizar o código?***

- Muitos sistemas tempo-real são usados em mercados:
  - Altamente sensíveis a custos
  - Consumo de energia deve ser minimizado
  - Espaço físico é restrito, ...
- Otimização de código permite produzir:
  - Programas mais rápidos:
    - Possibilita o uso de processadores mais lentos, com menor custo, menor consumo de energia.
  - Programas mais curtos:
    - Menos memória, logo menores custos e menor consumo de energia

# *Técnicas de otimização*

- Programação em linguagem *assembly*
  - Potencialmente permite um nível de eficiência muito elevado, mas:
    - Difícil *debug* e manutenção
    - Ciclo de desenvolvimento longos
    - Dependentes do processador – código não portátil!
    - Requerem ciclos de aprendizagem longos
- Programação em linguagens de alto nível (e.g. “C”)
  - Mais fáceis de desenvolver e manter, relativamente independente do processador, etc. mas:
    - Código potencialmente menos eficiente que o escrito em *assembly*



# “C” vs Assembly

- A programação em *assembly*, apesar de potencialmente mais eficiente, acaba por em geral não ser uma boa abordagem:
  - Foco em detalhes de implementação e não em questões algorítmicas fundamentais, onde residem habitualmente as melhores oportunidades de otimização
    - E.g.: em vez de dispende-se horas a construir uma biblioteca “ultra-eficiente” para manipulação de listas, será preferível usar “hash-tables”;
    - Compiladores de boa qualidade produzem código mais eficiente do que o código que um programador “médio” consegue produzir;
    - E finalmente,  
“Compilers make it a lot easier to use complex data structures, compilers don’t get bored halfway through, and generate reliably pretty good code.”  
[John Levine, on [comp.compilers](http://comp.compilers)]

# *E qual a melhor abordagem?*

Como em quase tudo na vida, “no meio é que está a virtude”, ou de outra forma, na “guerra” entre “C” e “assembly” ganha ... quem optar por ambas!

## Procedimento:

- O programador escreve a aplicação numa linguagem de alto nível (e.g. “C”)
- O programador usa ferramentas para detetar os “hot spots” (pontos em que a aplicação despende mais recursos)
- O programador analisa o código gerado e ...
  - Re-escreve as secções críticas em *assembly*
  - e/ou re-estrutura o código de alto nível para gerar um código *assembly* mais adequado



# Técnicas de otimização independentes do CPU

- Eliminação de sub-expressões comuns
  - Formalmente, a ocorrência de uma expressão “E” denomina-se sub-expressão comum se “E” foi previamente calculada e os valores das variáveis em “E” não sofreram alterações desde o último cálculo de “E”.
  - O benefício é óbvio: menos código para executar!

Antes

```
b:
    → t6 = 4 * i
      x = a[t6]
    → t7 = 4 * i
    → t8 = 4 * j
      t9 = a[t8]
    → a[t7] = t9
    → t10 = 4 * j
    → a[t10] = x
      goto b
```

Depois

```
b:
      t6 = 4 * i
      x = a[t6]
      t8 = 4 * j
      t9 = a[t8]
      a[t6] = t9
      a[t8] = x
      goto b
```

# Técnicas de otimização independentes do CPU

- Eliminação de “código morto”
  - Se um certo conjunto de instruções não é executado em nenhuma circunstância, é denominado “código morto” e pode ser removido

```
debug = 0;  
...  
if (debug){  
    print .....  
}
```

Substituindo os “if” por “#ifdef” permite ao compilador remover código de *debug*



# Técnicas de otimização independentes do CPU

- Variáveis de indução e redução de força
  - Uma variável “X” denomina-se variável de indução de um ciclo “L” se, de cada vez que “X” é alterada, é incrementada ou decrementada de um valor constante
    - Quando existem duas ou mais variáveis de indução num ciclo, poderá ser possível remover uma delas
    - Por vezes é também possível reduzir a sua “força”, i.e., o seu custo de execução
    - Benefícios: menos cálculos e menos custos

Antes

```
j = 0
label_XXX
    j = j + 1
    t4 = 11 * j
    t5 = a[t4]
    if (t5 > v) goto label_XXX
```

Depois

```
t4 = 0
label_XXX
    t4 += 11
    t5 = a[t4]
    if (t5 > v) goto label_XXX
```

# Técnicas de otimização independentes do CPU

- Expansão de ciclos
  - Consiste em efetuar múltiplas iterações dos cálculos em cada iteração do ciclo
    - Benefícios: redução do *overhead* devidos ao ciclo e oportunidade para outras otimizações
    - Problemas: maior quantidade de memória, divisão do ciclo não inteira
    - Apropriada para pequenos ciclos

**Antes**

```
int checksum(int *data, int N)
{
    int i, sum=0;
    for(i=0;i<N;i++)
    {
        sum += *data++;
    }
    return sum;
}
```

**Depois**

```
int checksum(int *data, int N)
{
    int i, sum=0;
    for(i=0;i<N;i+=4)
    {
        sum += *data++;
        sum += *data++;
        sum += *data++;
        sum += *data++;
    }
    return sum;
}
```



# Exemplo de expansão de ciclos

```
0x00:  MOV    r3,#0          ; sum =0
0x04:  MOV    r2,#0          ; i= 0
0x08:  CMP    r2,r1          ; (i < N) ?
0x0c:  BGE    0x20          ; go to 0x20 if i >= N
0x10:  LDR    r12,[r0],#4     ; r12 <- data++
0x14:  ADD    r3,r12,r3      ; sum = sum + r12
0x18:  ADD    r2,r2,#1       ; i=i+1
0x1c:  B      0x8           ; jmp to 0x08
0x20:  MOV    r0,r3          ; sum = r3
0x24:  MOV    r1,#1         ; i = 1
```

Ciclo original

Overhead do ciclo  
calculado N vezes

```
0x00:  MOV    r3,#0          ; sum = 0
0x04:  MOV    r2,#0          ; i = 0
0x08:  B      0x30          ; jmp to 0x30
0x0c:  LDR    r12,[r0],#4     ; r12 <- data++
0x10:  ADD    r3,r12,r3      ; sum = sum + r12
0x14:  LDR    r12,[r0],#4     ; r12 <- data++
0x18:  ADD    r3,r12,r3      ; sum = sum + r12
0x1c:  LDR    r12,[r0],#4     ; r12 <- data++
0x20:  ADD    r3,r12,r3      ; sum = sum + r12
0x24:  LDR    r12,[r0],#4     ; r12 <- data++
0x28:  ADD    r3,r12,r3      ; sum = sum + r12
0x2c:  ADD    r2,r2,#4       ; i = i + 4
0x30:  CMP    r2,r1          ; (i < N) ?
0x34:  BLT    0xc           ; go to 0x0c if i < N
0x38:  MOV    r0,r3          ; r0 <- sum
0x3c:  MOV    pc,r14         ; return
```

Ciclo após expansão (4 vezes)

Overhead do ciclo  
calculado N/4 vezes

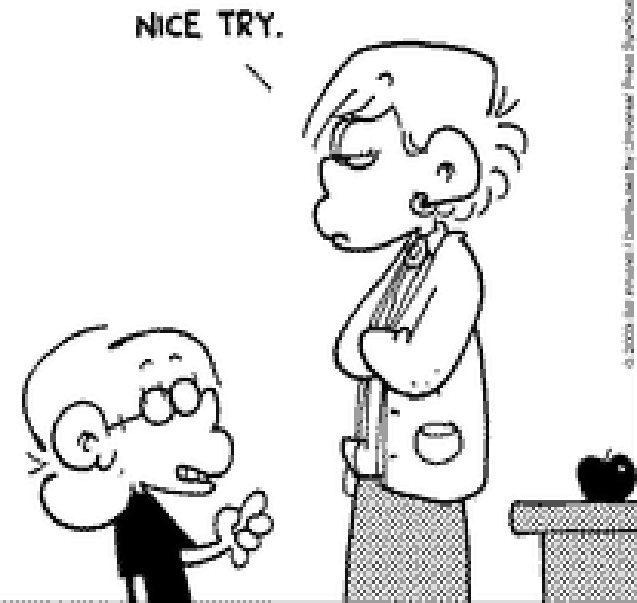
# Técnicas de otimização independentes do CPU

- Expansão de ciclos (cont.)
  - Como iremos ver mais à frente pode haver problemas associados a esta técnica ...

```
#include <stdio.h>
int main(void)
{
    int count;

    for (count = 1; count <= 500; count++)
        printf("I will not throw paper airplanes in class.");
    return 0;
}
```

AVOID 10.3





# Técnicas de otimização independentes do CPU

- *Inlining* de funções

- Substituir a chamada a uma função pelo código da função
  - Benefícios: redução do *overhead* associado à chamada de uma função e oportunidade para outras otimizações
  - Problemas: (possível) aumento do tamanho do código
  - Adequado quando funções pequenas são chamadas muitas vezes de um número reduzido de locais

```
void t(int x, int y)
{
    int a1=max(x,y);
    int a2=max(x+1,y);

    return max(a1+1,a2);
}
```

```
    inline int max(int a, int b)
{
    int x;
    x=(a>b ? a:b);
    return x;
}
```

# Exemplo de inlining de funções (sem)

```
void t(int x, int y)
{
    int a1=max(x,y);
    int a2=max(x+1,y);

    return max(a1+1,a2);
}
int max(int a, int b)
{
    int x;
    x=(a>b ? a:b);
    return x;
}
```

```
max
$a
0x00:    CMP        r0,r1;    (x > y) ?
0x04:    BGT        0x0c;    return if (x > y)
0x08:    MOV        r0,r1;    else r0 <- y
0x0c:    MOV        pc,r14    return
t
0x10:    STMFD       r13!,{r4,r14}; save registers
0x14:    MOV        r2,r0;                r2 <- x
0x18:    MOV        r3,r1;                r3 <- y
0x1c:    MOV        r1,r3;                r1 <- y
0x20:    MOV        r0,r2;                r0 <- x
0x24:    BL          max ;                r0 <- max(x,y)
0x28:    MOV        r4,r0;                r4 <- a1
0x2c:    MOV        r1,r3;                r1 <- y
0x30:    ADD        r0,r2,#1;            r0 <- x+1
0x34:    BL          max ;                r0 <- max(x+1,y)
0x38:    MOV        r1,r0 ;                r1 <- a2
0x3c:    ADD        r0,r4,#1 ;            r0 <- a1+1
0x40:    LDMFD       r13!,{r4,r14} ; restore
0x44:    B            max ;
```



# Exemplo de inlining de funções (com)

```
void t(int x, int y)
{
    int a1=max(x,y);
    int a2=max(x+1,y);

    return max(a1+1,a2);
}

inline int max(int a, int b)
{
    int x;
    x=(a>b ? a:b);
    return x;
}
```

0x00:	CMP	r0,r1 ; (x<= y) ?
0x04:	BLE	0x10 ; jmp to 0x10 if true
0x08:	MOV	r2,r0 ; a1 <- x
0x0c:	B	0x14 ; jmp to 0x14
0x10:	MOV	r2,r1 ; a1 <- y if x <= y
0x14:	ADD	r0,r0,#1; generate r0=x+1
0x18:	CMP	r0,r1 ; (x+1 > y) ?
0x1c:	BGT	0x24 ; jmp to 0x24 if true
0x20:	MOV	r0,r1 ; r0 <- y
0x24:	ADD	r1,r2,#1 ; r1 <- a1+1
0x28:	CMP	r1,r0 ; (a1+1 <= a2) ?
0x2c:	BLE	0x34 ; jmp to 0x34 if true
0x30:	MOV	r0,r1 ; else r0 <- a1+1
0x34:	MOV	pc,r14


# Impacto negativo na cache

O uso de técnicas como a expansão de ciclos ou o *inline* de funções pode causar degradação do desempenho em sistemas com cache!

```
int checksum(int *data, int N)
{
    int i;
    for(i=N; i>=0; i--)
    {
        sum += *data++;
    }
    return sum;
}
```

*Antes de efetuar a expansão do ciclo*

0x0000000c:	LDR	r3, [r2], #4
0x00000010:	ADD	r0, r3, r0
0x00000014:	SUB	r1, r1, #1
0x00000018:	CMP	r1, #0
0x0000001c:	BGE	0xc



0x0000000C
0x00000010
0x00000014
0x00000018
0x0000001C

Cache de instruções



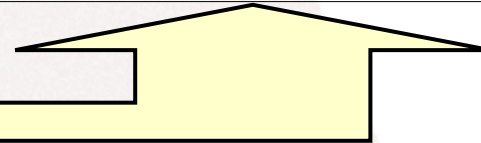
# Impacto negativo na cache (cont)

*Após a expansão do ciclo*

```
int checksum(int *data, int N)
{
    int i;
    for(i=N; i>=0; i-=4)
    {
        sum += *data++;
        sum += *data++;
        sum += *data++;
        sum += *data++;
    }

    return sum;
}
```

```
0x00000008: LDR    r3,[r0],#4
0x0000000c: ADD     r2,r3,r2
0x00000010: LDR     r3,[r0],#4
0x00000014: ADD     r2,r3,r2
0x00000018: LDR     r3,[r0],#4
0x0000001c: ADD     r2,r3,r2
0x00000020: LDR     r3,[r0],#4
0x00000024: ADD     r2,r3,r2
0x00000028: SUB     r1,r1,#4
0x0000002c: CMP     r1,#0
0x00000030: BGE     0x8
```

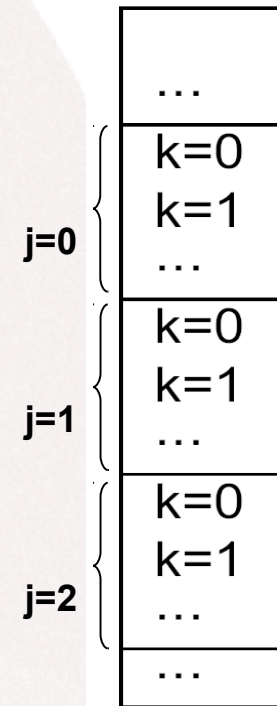


Dimensão da cache inferior ao tamanho do código; gerados “cache misses”

# *Técnicas de otimização dependente da arquitetura de memória*

- Ordem de acesso à memória
  - Em matrizes a linguagem “C” define que o índice mais à direita define posições de memória adjacentes
  - Impacto grande na memória cache de dados em estruturas de elevada dimensão

Array  $p[j][k]$





# *Técnicas de otimização dependente da arquitetura de memória*

- Melhor desempenho quando o ciclo interno corresponde ao índice mais à direita

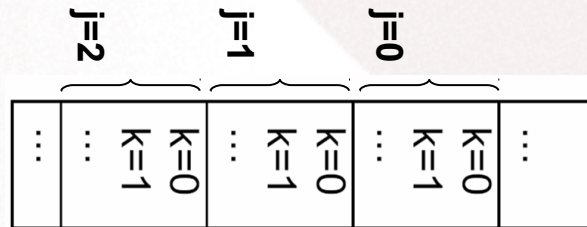
```
for (k=0; k<=m; k++)  
  for (j=0; j<=n; j++) )  
    p[j][k] = ...
```

```
for (j=0; j<=n; j++)  
  for (k=0; k<=m; k++)  
    p[j][k] = ...
```

Se acesso à memória homogêneo, o desempenho é idêntico, mas:

↑ Mau desempenho  
com cache

Bom desempenho ↑  
com cache



□ Otimização dependente da arquitetura de memória

# ***Técnicas de otimização dependentes da arquitetura***

Dependendo da família de processadores usados bem como do tipo de co-processadores disponíveis, tornam-se possíveis diversas otimizações

- Conversão de virgula flutuante para virgula fixa
  - Benefícios
    - Menor custo computacional,
    - Menor consumo de energia,
    - Relação sinal-ruído suficiente se corretamente escalado,
    - Adequado e.g. para aplicações móveis.
  - Problemas:
    - Redução da gama dinâmica,
    - Possíveis “overflows”.



# Técnicas de otimização dependentes da arquitetura

- Uso de particularidades do *assembly*
  - Exemplo: na arquitetura ARM é possível afetar as *flags* ao fazer uma operação aritmética

```
int checksum_v1(int *data)
{
    unsigned i;
    int sum=0;

    for(i=0;i<64;i++)
        sum += *data++;

    return sum;
}
```

```
MOV    r2, r0; r2=data
MOV    r0, #0; sum=0
MOV    r1, #0; i=0
L1 LDR  r3, [r2], #4; r3=*(data++)
ADD    r1, r1, #1; i=i+1
CMP    r1, 0x40; cmp r1, 64
ADD    r0, r3, r0; sum +=r3
BCC    L1; if i < 64, goto L1
MOV    pc, lr; return sum
```

```
int checksum_v2(int *data)
{
    unsigned i;
    int sum=0;

    for(i=63;i >= 0;i--)
        sum += *data++;

    return sum;
}
```

```
MOV    r2, r0; r2=data
MOV    r0, #0; sum=0
MOV    r1, #0x3f; i=63
L1 LDR  r3, [r2], #4; r3=*(data++)
ADD    r0, r3, r0; sum +=r3
SUBS   r1, r1, #1; i--, set flags
BGE    L1; if i >= 0, goto L1
MOV    pc, lr; return sum
```

# ***Técnicas de otimização dependentes da arquitetura***

- Existem imensas outras técnicas que, por limitações de espaço e tempo não podem ser cobertas:
  - Algumas classes:
    - Exploração do paralelismo
    - Múltiplos bancos de memória
    - Instruções multimédia
    - Partição de tarefas (múltiplos processadores)
    - ....



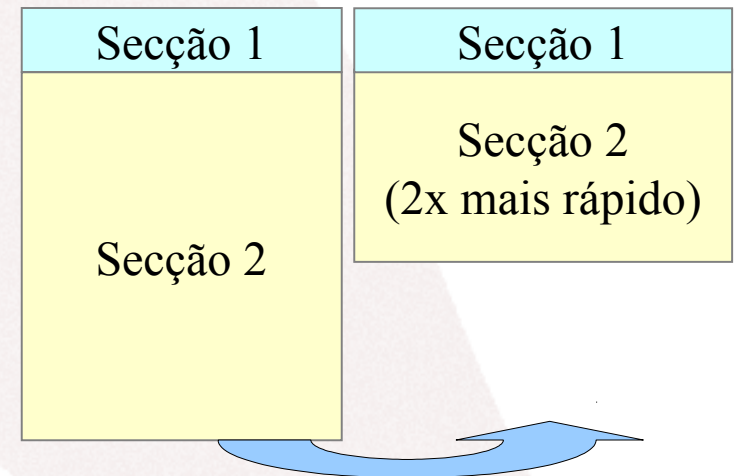
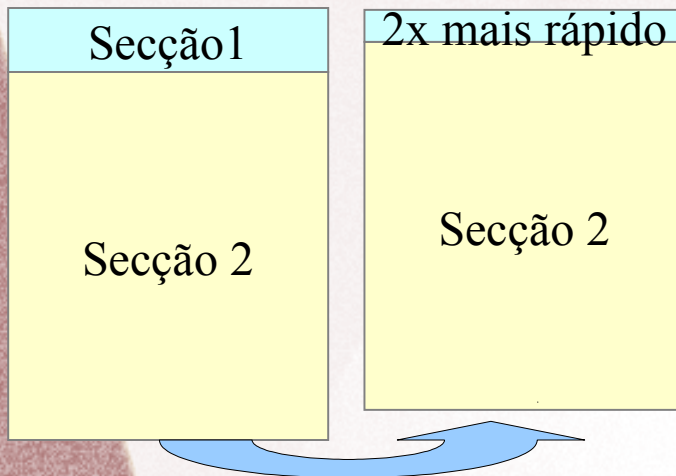
# Profiling

- Tarefa: Dado o código de um programa, eventualmente escrito por outra pessoa, efetuar a sua otimização
- Por onde começar?
  - Analisar o código fonte e detetar código “C” ineficiente
  - Re-escrever algumas secções em *assembly*
  - Usar algoritmos mais eficientes
- Como determinar quais as secções otimizar?
  - Uma aplicação típica consiste em muitas funções espalhadas por diferentes ficheiros-fonte
  - A inspeção manual de toda a aplicação para determinar quais as secções de código a otimizar é na prática impossível!

# Profiling

- A lei de Amdahl

*O ganho de desempenho que se obtém ao otimizar uma secção de código mais eficiente está limitado à fração do tempo total que é gasto nessa secção particular.*



- **Questão: Como determinar as partes de código em que a aplicação gasta a maior parte do tempo?**



# Profiling

- *Profiling*: recolha de dados estatísticos efetuada sobre a execução de uma aplicação
  - Fundamental para determinar o peso relativo de cada função
- Abordagens
  - *Call graph profiling*: a invocação de funções é instrumentada
    - Intrusivo, requer acesso ao código fonte, computacionalmente pesado (*overhead* pode chegar aos 20%)
  - *Flat profiling*: o estado da aplicação é amostrado em intervalos de tempo regulares
    - Rigoroso desde que as funções tenham duração >> que o período de amostragem

# Profiling

- Exemplo:

Routine	% of Execution Time
function_a	60%
function_b	27%
function_c	4%
...	
function_zzz	0.01%

Verifica-se que numa aplicação “típica” cerca de 80% do tempo é despendido em cerca de 20% do código (“lei do 80/20”)



# ***Profiling - ferramentas***

## Utilização do gprof (exemplo em Linux)

- O *profiling* requer vários passos
  - Compilação e “linkagem” da aplicação com o *profiling* activo
    - `gcc -pg -o sample sample.c`
  - Executar o programa para gerar dados estatísticos (*profiling data*)
    - `./sample`
  - Executar o programa gprof para analisar os dados
    - `gprof ./sample`

*-pg Generate extra code to write profile information suitable for the analysis program gprof.*

# ***Profiling - ferramentas***

## Utilização do gcov (exemplo em Linux)

- Teste de cobertura; complementar ao gprof. Indica o número de vezes que cada linha é executada
  - maior granularidade
  - Compilação e “linkagem”
    - `gcc -lm -pg -fprofile-arcs -ftest-coverage -o sample sample.c`
  - Executar o programa para gerar dados estatísticos (*profiling data*)
    - `./sample`
    - `gcov sample.c`
  - O ficheiro “sample.c.cov” contem os dados de execução



# Profiling - ferramentas

```
... (main) ...
1:      5:{
-:      6: int i;
1:      7: int colcnt = 0;
200000: 8: for (i=2; i <= 200000; i++)
199999: 9: if (prime(i)) {
17984: 10: colcnt++;
17984: 11: if (colcnt%9 == 0) {
1998: 12: printf("%5d\n",i);
1998: 13: colcnt = 0;
-:     14: }
-:     15: else
15986: 16: printf("%5d ", i);
-:     17: }
1:     18: putchar('\n');
1:     19: return 0;
-:     20: }
199999: 21: int prime (int num) {
-:     22: /* check to see if the number is a prime? */
-:     23: int i;
1711598836: 24: for (i=2; i < num; i++)
1711580852: 25: if (num %i == 0)
182015: 26: return 0;
17984: 27: return 1;
-:     28: }
```

Número de execuções  
muito elevado!

Ponto de otimização

# Profiling - ferramentas

Analisando o código identificou-se uma otimização ...

```
....  
199999: 22:int prime (int num) {  
-:      23: /* check to see if the number is a prime? */  
-:      24: int i;  
7167465: 25: for (i=2; i < (int) sqrt( (float) num); i++)  
7149370: 26: if (num %i == 0)  
181904: 27: return 0;  
....
```

Redução do número  
de execuções por um  
fator de 238!!!!



# Profiling - ferramentas

- Resultados com o gprof

## Sem otimização:

Call graph

granularity: each sample hit covers 4 byte(s) for 0.02% of 40.32 seconds

...

index	% time	self	children	called	name
[1]	100.0	0.01	40.31		main [1]
		40.31	0.00	199999/199999	prime [2]
...					

## Com otimização:

...

Call graph

granularity: each sample hit covers 4 byte(s) for 2.63% of 0.38 seconds

...

index	% time	self	children	called	name
[2]	100.0	0.00	0.38		main [2]
		0.38	0.00	199999/199999	prime [1]
...					



Redução do  
tempo de execução  
por um fator de  
106!!!!

# Sumário

- Melhorando a performance das aplicações
  - Porquê otimizar
  - Programação Assembly vs programação em “C”
  - Otimizações independentes da arquitetura
    - Eliminação de sub-expressões comuns, eliminação de código morto, redução de variáveis de indução, expansão de ciclos, *inlining*
  - Impacto na cache
  - Acesso à memória



# Sumário

- Otimizações dependentes da arquitetura
  - Virgula flutuante-> fixa, especificidades do *assembly*
- Otimização/profiling
  - Metodologia geral
  - *Case study*: uso do gprof e gconv