Técnicas para Otimização de Código

Sarita Mazzini Bruschi

Laboratório de Sistemas Distribuídos e Programação Concorrente (LaSDPC)
Departamento de Sistemas de Computação (SSC)
Instituto de Ciências Matemáticas e de Computação (ICMC)
Unidersidade de São Paulo (USP)

Novembro, 2024

Agenda

Motivação

Conceitos importantes

Otimizando código sequencial

Ferramentas

Técnicas para melhorar o desempenho

Motivação

O que fazer para melhorar desempenho das aplicações?

- Paralelizar

Mas antes de paralelizar...

- Avaliar o desempenho
- Comparar os resultados

Conceitos Importantes

Conceitos Importantes - Desempenho

O que é desempenho?

- Quantidade de trabalho útil realizado por um computador
- Deve ser mensurável

Pode ser

- Absoluto: tempo de execução, latência, vazão (*throughput*), consumo energético
- Relativo: composição de várias métricas (*Passmark rating*)

Conceitos importantes – Avaliação de Desempenho

Vários fatores influenciam o desempenho de um código

- Hardware: arquitetura (ISA, velocidade), hierarquia de memória
- Software: estrutura do código, compilador/interpretador, sistema operacional

Importante definir corretamente como o desempenho pode ser medido e analisado: Avaliação de Desempenho

- Definição dos objetivos da avaliação
- Definição das métricas
- Definição da técnica de avaliação a ser usada: aferição ou modelagem

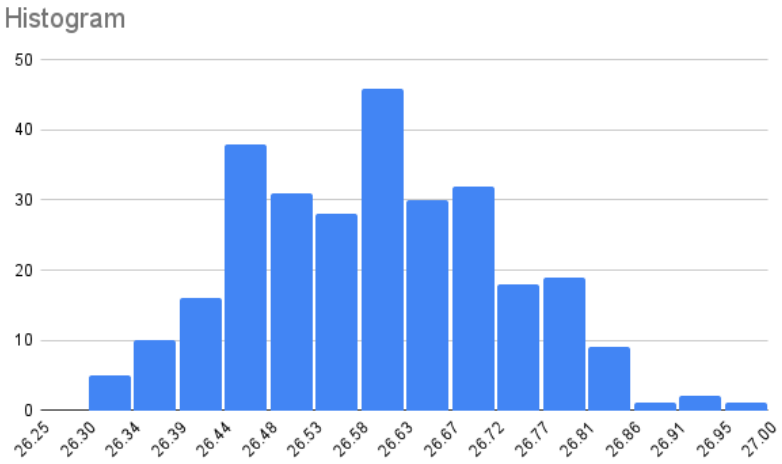
Conceitos importantes – Planejamento de Experimentos

- Fatores e níveis
- Planejamento fatorial parcial ou fatorial completo
- Análise da influência dos fatores
- Exemplo:
  - Fator 1: método para multiplicação de matrizes
    - Níveis: algoritmo tradicional, algoritmo otimizado
  - Fator 2: alocação da memória
    - Níveis: 100 x 100, 1000 x 1000
  - Planejamento total (todas as combinações): 4 experimentos:
    - Algoritmo tradicional, matriz 100 x 100
    - Algoritmo tradicional, matriz 1000 x 1000
    - Algoritmo otimizado, matriz 100 x 100
    - Algoritmo otimizado, matriz 1000 x 1000

Conceitos importantes – Análise dos resultados

Análise

- Precisão dos resultados: média, desvio padrão, intervalo de confiança
- Importante verificar se os dados seguem um padrão de distribuição



Conceitos importantes – Organização e Arquitetura

- ILP: Instruction Level Paralellism
- IPC: Instruction Per Clock
- CPI: Clock Per Instruction
- Pipeline
  - Parada do pipeline (*Stalls*)
  - Predição de desvio
  - Dependência de dados
- Cache

Otimizando código sequencial

Técnicas básicas para otimização de código serial

- Faça menos trabalho! (*Do less work!*)

```
bool flag = false;
for (i=0; i<N; i++)
{
    if (A[i] > 5.0)
    flag = true;
}
return flag;
```

```
for (i=0; i<N; i++)
{
if (A[i] > 5.0)
    return true;
}
return false;
```

Conteúdo basedo em:
Introduction to High Performance Computing for Scientists and Engineers
Georg Hager e Gerhard Wellein – CRC Press - 2011

Técnicas básicas para otimização de código serial

- Evite operações custosas! (*Avoid expensive operations!*)

```
int ang;
for (...) {
    ang = calcula_angulo();
    res += tan(ang);
}
```

```
int ang;
for (...) {
    ang = calcula_angulo();
    res += tan_table[ang%360];
}

double tan_table[] = {
    0.0,
    ....
}
```

Conteúdo basedo em:
Introduction to High Performance Computing for Scientists and Engineers
Georg Hager e Gerhard Wellein – CRC Press - 2011

Técnicas básicas para otimização de código serial

- Encolha o conjunto de dados! (*Shrink the working set!*)
  - O “conjunto de dados” de um código é a quantidade de memória que o programa usa para realizar as computações
  - Aumenta a probabilidade de *cache hit* pois cabe mais dados na cache e também pode acarretar em mais operações por ciclo quando se tem uma instrução SIMD
  - Exemplo:
    - double => float
    - int => short

Conteúdo basedo em:
Introduction to High Performance Computing for Scientists and Engineers
Georg Hager e Gerhard Wellein – CRC Press - 2011

Técnicas básicas para otimização de código serial

- Elimine as expressões comuns (*Elimination of common subexpression*)

```
int r, s;
...
for (i=0; i<N; i++)
{
    A[i] = A[i] + r + s + sin(x);
}
```

```
int r, s;
...
int tmp = r + s + sin(x);
for (i=0; i<N; i++)
{
    A[i] = A[i] + tmp;
}
```

Normalmente o compilador consegue fazer isso, mas se for preciso usar regras de associatividade (principalmente com valores em ponto flutuante), normalmente o compilador não faz

Conteúdo basedo em:
Introduction to High Performance Computing for Scientists and Engineers
Georg Hager e Gerhard Wellein – CRC Press - 2011

Técnicas básicas para otimização de código serial

- Evite os desvios condicionais (*Avoiding branches*)
  - Loops com poucas instruções são candidatos a otimizações automáticas, tais como *loop unrolling*
  - Se o loop possui um desvio condicional dentro dele, o compilador pode falhar na otimização

```
float sinal;
for (i=0; i<N; i++)
    for (j=0; j<N; j++) {
       if(i>j)
         sinal = 1.0;
       else if(i<j)
         sinal = -1.0;
       else
         sinal = 0.0;
       acc += sinal x A[i][j];
    }
```

```
float sinal;
for (i=0; i<N; i++)
   for (j=0; j<i; j++)
     acc += A[i][j];

for (i=0; i<N; i++)
  for (j=i+1; j<N; j++)
    acc -= A[i][j];
```

Conteúdo basedo em:
Introduction to High Performance Computing for Scientists and Engineers
Georg Hager e Gerhard Wellein – CRC Press - 2011

Ferramentas

Códigos no Github

- https://github.com/saritabruschi/erad-co

Ferramentas

Time:

- real: tempo total entre o início e o fim da execução
- user: tempo gasto pelas instruções do programa
- sys: tempo usado nas chamadas ao sistema (*kernel*)
- Vantagem: não é invasivo

Profiling (Perfilador)

- Profiling: aquisição de informações sobre o comportamento de um programa, especialmente no uso dos recursos computacionais (memória, CPU, etc.)
- Dois tipos:
  - Instrumentação: compilador insere instruções no código binário para que informações sejam coletadas
  - Amostragem: o programa é interrompido de tempos em tempos para coleta das informações
- Ferramentas:
  - Gprof (vinculada ao gcc) - instrumentação
  - Perf - amostragem

Ferramenta Perf

- `perf record ./código`
  - Salva as amostragens do PC no arquivo perf.data
- `perf script`
  - Inspeciona as amostras
- `perf script | cut d: -f3 | sort –n | uniq –c`
  - Gera um “histograma” do número de ocorrências de cada função
- `perf report`

Ferramenta Perf

- Eventos que podem ser amostrados (exemplos):
  - Cache misses
  - Branch misprediction
  - Page fault
- `perf list`
  - Lista todos os eventos
- `perf stat` *`command`*
  - Mostra o contador de estatísticas para o *`command`*
  - `perf stat –e L1-dcache-loads`
- `perf record –e L1-dcache-loads`

Ferramenta Perf

- http://www.brendangregg.com/perf.html
  - Seção 2 (One-Liners), counting events

Ferramenta Gprof

- Ferramenta classificada como perfilador por instrumentação
  - Periodicamente o programa é interrompido e o valor do PC é salvo
- Ferramenta do projeto GNU e faz parte do conjunto de ferramentas binárias GNU Binutils
- Códigos com várias funções
  - Conforme o código aumenta, o número de funções aninhadas pode aumentar muito
  - É preciso entender quanto tempo cada função está demorando para executar
- A ferramenta constrói também um grafo de chamadas

Ferramenta Gprof

- Se não tiver instalado:
  - *apt get install binutils*
- Basta compilar o código com a diretiva –pg
- Executar o código normalmente que será gerado um arquivo com o nome *gmon.out*
- Para analisar o arquivo gerado basta executar a ferramenta gprof:
  - *gprof OPÇÕES [<nome executável> gmon.out] > <arquivo_saída>*

Ferramenta Gprof

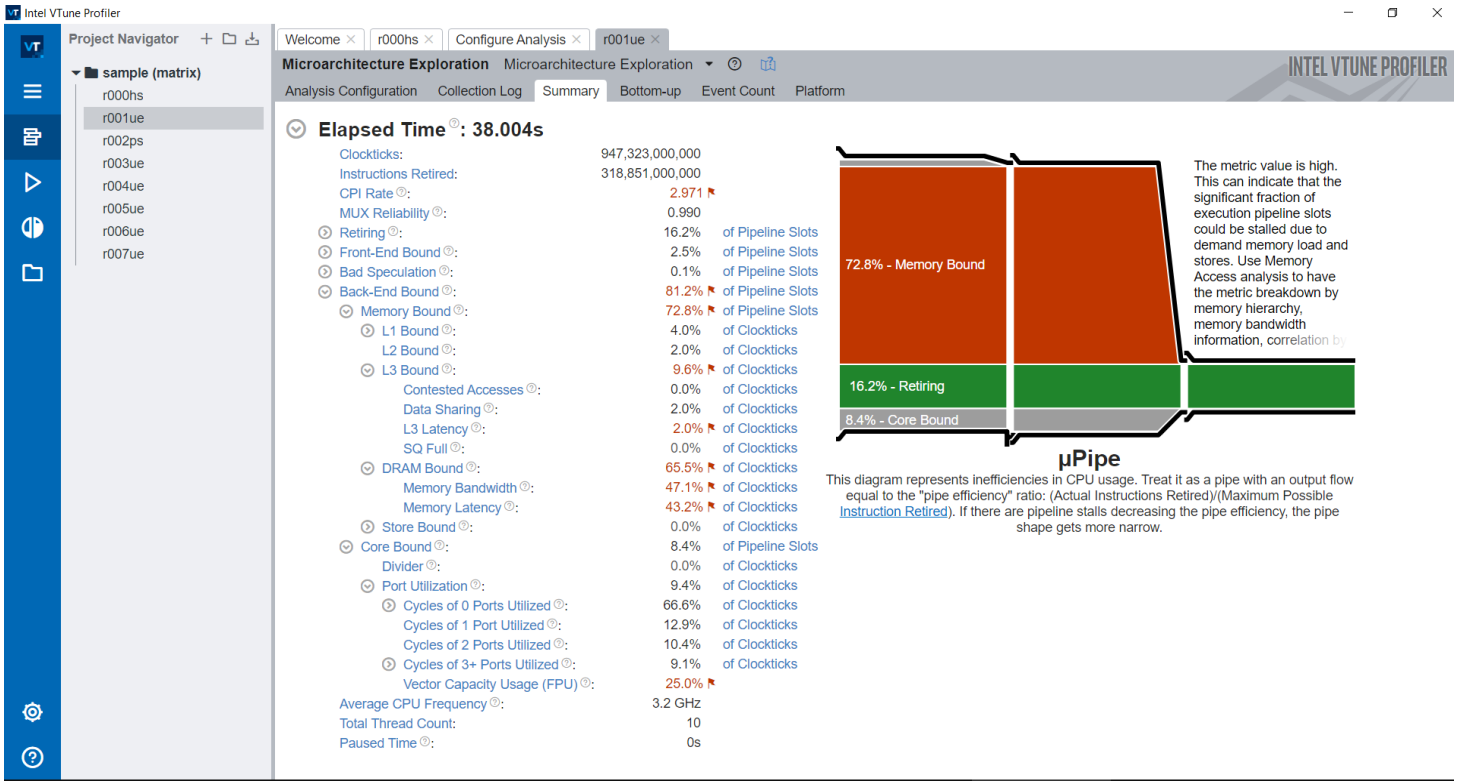
- Como interpretar o arquivo de saída?
  - Composto de duas tabelas: **flat profile** e **call graph**

- **Flat profile**: mostra informações sobre o tempo que o programa gastou em cada método ou rotina e quantas vezes aquela rotina foi executada
- **Call graph**: mostra informações no formato de um grafo direcionado acíclico

Ferramenta Gprof

- Para poder visualizar o grafo das chamadas as funções (call graph) de maneira gráfica deve-se utilizar:
  1. Programa gprof2dot.py, que converte a saída gerada pelo gprof em um arquivo .dot (https://pypi.org/project/gprof2dot/)
     python gprof2dot.py saida_exemplo.txt > saida_exemplo.dot
  2. Biblioteca grahpviz para gerar uma imagem a partir do .dot
     dot -Tpng -o saida_grafo.png saida_exemplo.dot

Ferramenta Intel VTune Profiles



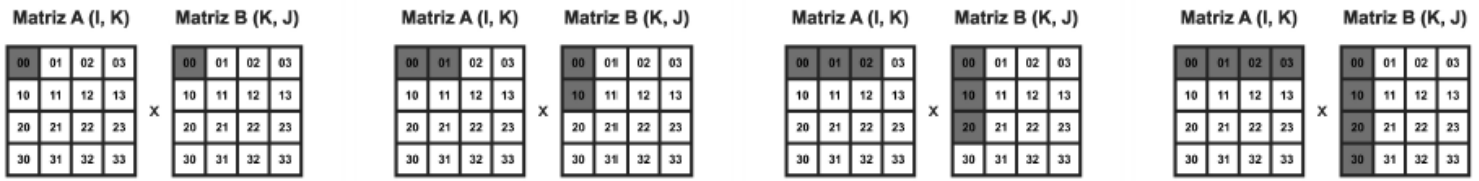
Técnicas para melhorar o desempenho

Técnicas para melhorar o desempenho

- Multiplicação de matrizes:
- Versão tradicional
- Otimizações possíveis
  - Loop interchange
  - Loop unrolling
  - Loop tiling (Blocagem)

Multiplicação de Matrizes Tradicional

```
1 for(i = 0; i < SIZE; i++)
2    for(j = 0; j < SIZE; j++)
3         for(k = 0; k < SIZE; k++){
4              C(i, j) += A(i, k) * B(k, j);
5         }
```

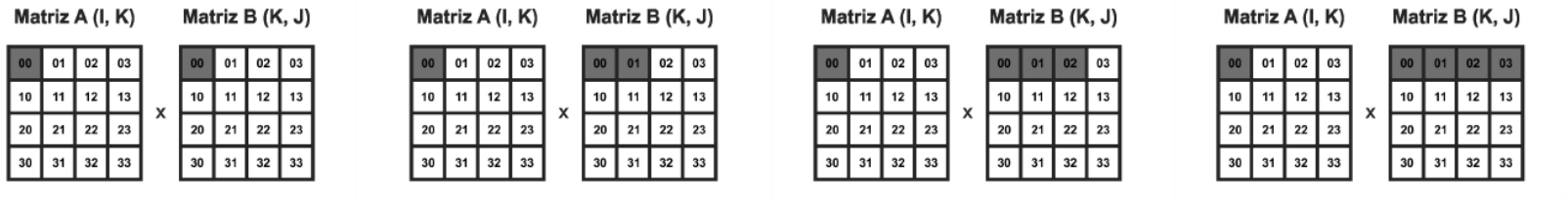


(a) Iteração 1 (b) Iteração 2 (c) Iteração 3 (d) Iteração 4

Conteúdo basedo em:
Derrube Todos os Recordes de Ganho de Desempenho Otimizando seu Código
Sherlon Almeida da Silva, Matheus S. Serpa, Claudio Schepke - Minicurso ERAD-RS 2017

Loop Interchange

```
1 for(i = 0; i < SIZE; i++)
2    for(k = 0; k < SIZE; k++)
3         for(j = 0; j < SIZE; j++){
4              C(i, j) += A(i, k) * B(k, j);
5         }
```

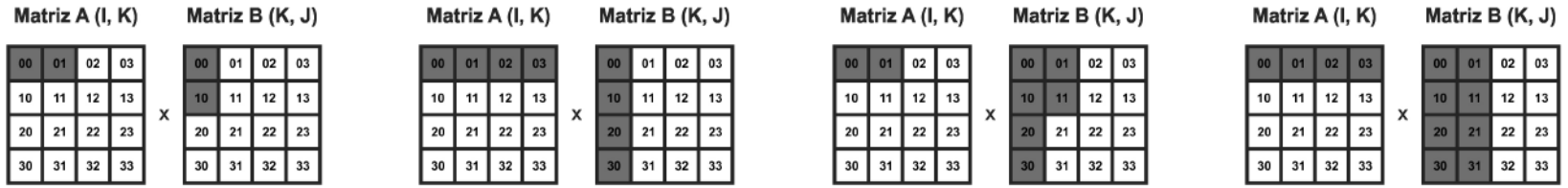


(a) Iteração 1 (b) Iteração 2 (c) Iteração 3 (d) Iteração 4

Conteúdo basedo em:
Derrube Todos os Recordes de Ganho de Desempenho Otimizando seu Código
Sherlon Almeida da Silva, Matheus S. Serpa, Claudio Schepke - Minicurso ERAD-RS 2017

Loop Unrolling

```
1 for(i = 0; i < SIZE; i++)
2    for(j = 0; j < SIZE; j++)
3         for(k = 0; k < SIZE; k+=2){
4              C(i, j) += A(i, k) * B(k, j);
5              C(i, j) += A(i, k+1) * B(k+1, j);
6         }
```



(a) Iteração 1 (b) Iteração 2 (c) Iteração 3 (d) Iteração 4

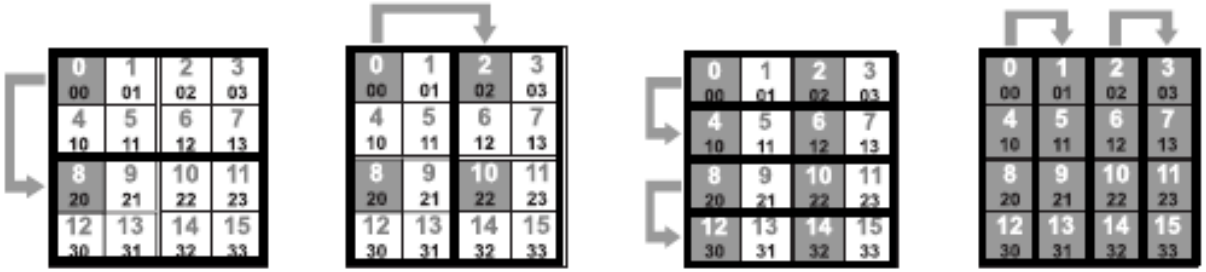
Conteúdo basedo em:
Derrube Todos os Recordes de Ganho de Desempenho Otimizando seu Código
Sherlon Almeida da Silva, Matheus S. Serpa, Claudio Schepke - Minicurso ERAD-RS 2017

Loop tiling (Blocagem)

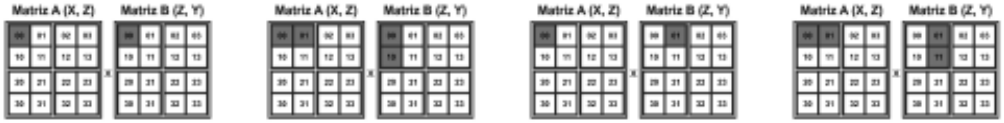
```
1 for(x = 0; x < SIZE; x += TILE_SIZE)
2   for(y = 0; y < SIZE; y += TILE_SIZE)
3       for(z = 0; z < SIZE; z += TILE_SIZE)
4           for(i = 0; i < TILE_SIZE; i++)
5               for(j = 0; j < TILE_SIZE; j++)
6                   for(k = 0; k < TILE_SIZE; k++)
7                   {
8                       C(i, j) += A(i, k) * B(k, j);
9                   }
```

Conteúdo basedo em:
Derrube Todos os Recordes de Ganho de Desempenho Otimizando seu Código
Sherlon Almeida da Silva, Matheus S. Serpa, Claudio Schepke - Minicurso ERAD-RS 2017

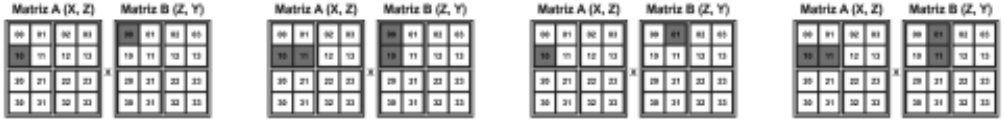
Loop tiling (Blocagem)



(a) Blocos (x) (b) Blocos (y) (c) Linhas (i) (d) Colunas (j)



(a) Iteração 1 (b) Iteração 2 (c) Iteração 3 (d) Iteração 4



(e) Iteração 5 (f) Iteração 6 (g) Iteração 7 (h) Iteração 8

Conteúdo basedo em:

Derrube Todos os Recordes de Ganho de Desempenho Otimizando seu Código

Sherlon Almeida da Silva, Matheus S. Serpa, Claudio Schepke - Minicurso ERAD-RS 2017

Técnicas para melhorar o desempenho

Diretivas do compilador (GCC)

Opções:

-O0

No optimization (the default); generates unoptimized code but has the fastest compilation time. Note that many other compilers do fairly extensive optimization even if “no optimization” is specified.
With gcc, it is very unusual to use -O0 for production if execution time is of any concern, since -O0 really does mean no optimization at all. This difference between gcc and other compilers should be kept in mind when doing performance comparisons.

-O1

Moderate optimization; optimizes reasonably well but does not degrade compilation time significantly.

-O2

Full optimization; generates highly optimized code and has the slowest compilation time.

-O3

Full optimization as in -O2; also uses more aggressive automatic inlining of subprograms within a unit and attempts to vectorize loops.

-Os

Optimize space usage (code and data) of resulting program.

Diretivas do Compilador (GCC)

Algumas flags importantes:

- -finline-functions: coloca todas as funções inline
- -ftree-vectorize: realiza vetorização na fase de geração da árvore sintática
- -fif-conversion: transforma desvios condicionais

Outras opções que afetam desempenho:

- -march: ativa a otimização para uma determinada arquitetura
- -mtune: ativa a otimização para uma determinada microarquitetura

Lista completa:

https://gcc.gnu.org/onlinedocs/gcc-7.2.0/gcc/Optimize-Options.html

Qual conjunto de flags trará o melhor desempenho para um determinado código?

Otimizações do compilador

- Estudo:

  Finding Best Compiler Options for Critical Software Using Parallel Algorithms

  International Symposium on Intelligent and Distributed Computing, 2018

- Utiliza:

  The Computer Language Benchmarks Game (CLBG)

  https://benchmarksgame-team.pages.debian.net/benchmarksgame/

Técnicas para melhorar o desempenho

- Vetorização utilizando instruções *intrinsics*
  - How to Write Fast Numerical Code - Markus Püschel - ETH - Swiss Federal Institute of Technology Zurich
  - https://people.inf.ethz.ch/markusp/teaching/263-2300-ETH-spring11/slides/class17.pdf
- Intel Intrinsics Guide
  - https://software.intel.com/sites/landingpage/IntrinsicsGuide/

Técnicas para melhorar o desempenho

- Uso de Bibliotecas Otimizadas
- BLAS (*Basic Linear Algebra Subprogram*)
  - define várias funções para operações de álgebra linear sobre vetores e matrizes. Existem várias implementações dessas funções, como por exemplo, a MKL (Math Kernel Library) (https://software.intel.com/content/www/us/en/develop/documentation/mkl-developer-reference-c/top.html) da Intel e a GSL CBLAS (GNU Scientific Library) (https://www.gnu.org/software/gsl/doc/html/)
- VOLK (*Vector Optimized Library of Kernels*)

Técnicas para melhorar o desempenho

- Módulo para profiling do código em Python:
  - timeit (https://docs.python.org/3/library/timeit.html)

```
import timeit

timeit.timeit('"-".join(str(n) for n in range(100))', number=10000)
```

```
import timeit

timeit.timeit('"-".join([str(n) for n in range(100)])', number=10000)
```

```
import timeit

timeit.timeit('"-".join(map(str, range(100)))', number=10000)
```

Otimizações em Python

- timeit

```
import timeit

def cube():
  cube_numbers = []
  for n in range(0,100):
    if n % 2 == 1:
      cube_numbers.append(n**3)
  return cube_numbers

print(timeit.timeit(cube))
```

Otimizações em Python

- timeit

```
import timeit

def cube():
  cube_numbers = [n**3 for n in range(1,100) if n%2 == 1]
  return cube_numbers

print(timeit.timeit(cube))
```

Otimizações em Python

- timeit

```
import hashlib
import timeit

def hash_func():
      return hashlib.sha256(b'Exemplo de profiling de codigo com timeit').hexdigest()

#hash_func()

#print(timeit.timeit(hash_func, number=1000))
print(timeit.timeit(hash_func, number=1000000))
```

Otimizações em Python

- Módulo para profiling do código em Python:
  - cProfile e profile (https://docs.python.org/3/library/profile.html)
    - cProfile: extensão escrita em C, com um overhead razoável, e recomendado para programas de longa execução
    - profile: módulo em Python que adiciona um significante overhead

Otimizações em Python

- Memoization
  - Método usado para armazenar os resultados de chamadas de funções anteriores

```
@functools.lru_cache(maxsize=128)
```

```
import cProfile
def fibonacci(n):
      if n == 0: # There is no 0'th number
            return 0
      elif n == 1: # We define the first number as 1
            return 1
      return fibonacci(n-1) + fibonacci(n-2)

cProfile.run("fibonacci(36)")
```

```
import cProfile
import functools

@functools.lru_cache(maxsize=128)
def fibonacci(n):
      if n == 0:
            return 0
      elif n == 1:
            return 1
      return fibonacci(n-1) + fibonacci(n-2)

cProfile.run("fibonacci(36)")
```