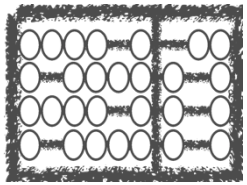
Análise de Aplicações e Otimização de Desempenho

Prof. Edson Borin
edson@ic.unicamp.br



Institute of Computing

University of Campinas

Trajetória


Venezuela
Bogota
Colombia
Guyana
French Guiana
Suriname
Roraima
Amapá
Amazonas
Pará
Maranhão
Rio Grande do Norte
Ceará
Paraiba
Piaui
Pernambuco
Brazil
Peru
Acre
Rondônia
Tocantins
Alagoas
Mato Grosso
Bahia
Sergipe
Goiás
Brasilia
La Paz
Bolivia
Minas Gerais
Espirito Santo
Mato Grosso do Sul
Belo Horizonte
São Paulo
Rio de Janeiro
Paraguay
Sao Paulo
Rio de Janeiro
Asuncion
Paraná
Chile
Santa Catarina
Rio Grande do Sul

Trajetória

PARANAVAÍ

Trajetória

PARANAVAÍ

PORTO VELHO

Trajetória

PARANAVAÍ

PORTO VELHO

CAMPO GRANDE

Trajetória

PARANAVAÍ

PORTO VELHO

CAMPO GRANDE

UFMS

CAMPINAS

UNICAMP

Trajetória

PARANAVAÍ

PORTO VELHO

CAMPO GRANDE

UFMS

CAMPINAS

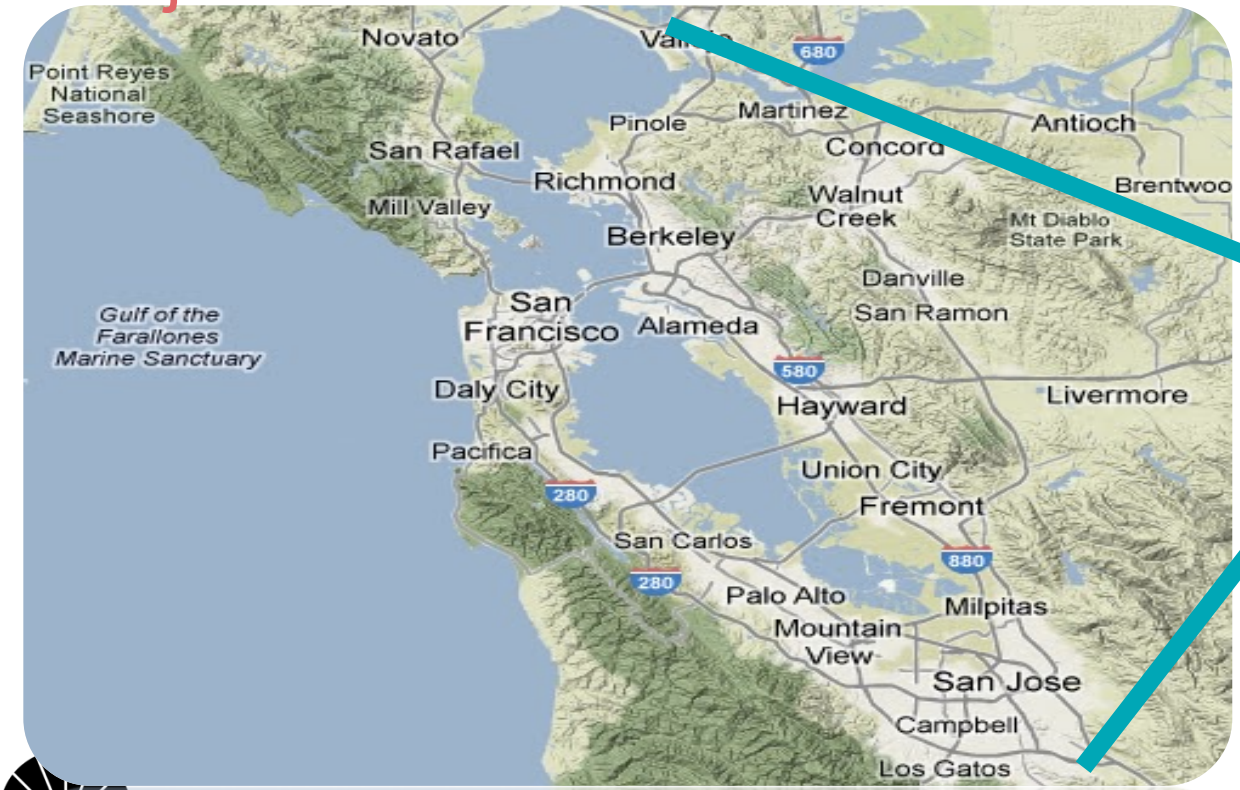
UNICAMP

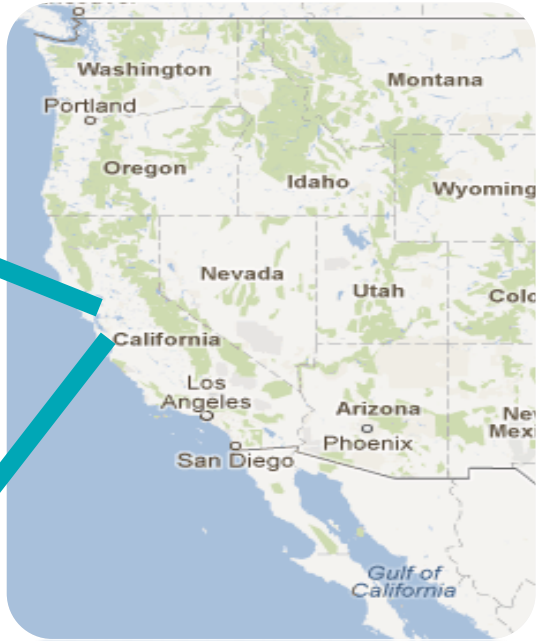
VALE DO SILÍCIO

Trajetória


Washington
Portland
Montana
North Dakota
Minnesota
Montrea
Oregon
Idaho
Wyoming
South Dakota
Wisconsin
Michigan
Toronto
New York
Chicago
Iowa
Nebraska
Illinois
Pennsylvania
Ohio
New York
Nevada
Utah
Colorado
United States
Kansas
Missouri
Indiana
West Virginia
Kentucky
Virginia
California
Los Angeles
Arizona
Phoenix
San Diego
New Mexico
Oklahoma
Arkansas
Tennessee
North Carolina
Dallas
Mississippi
South Carolina
District of Columbia
Alabama
Georgia
Texas
Louisiana
Houston
San Antonio
Florida
Gulf of California
Gulf of Mexico

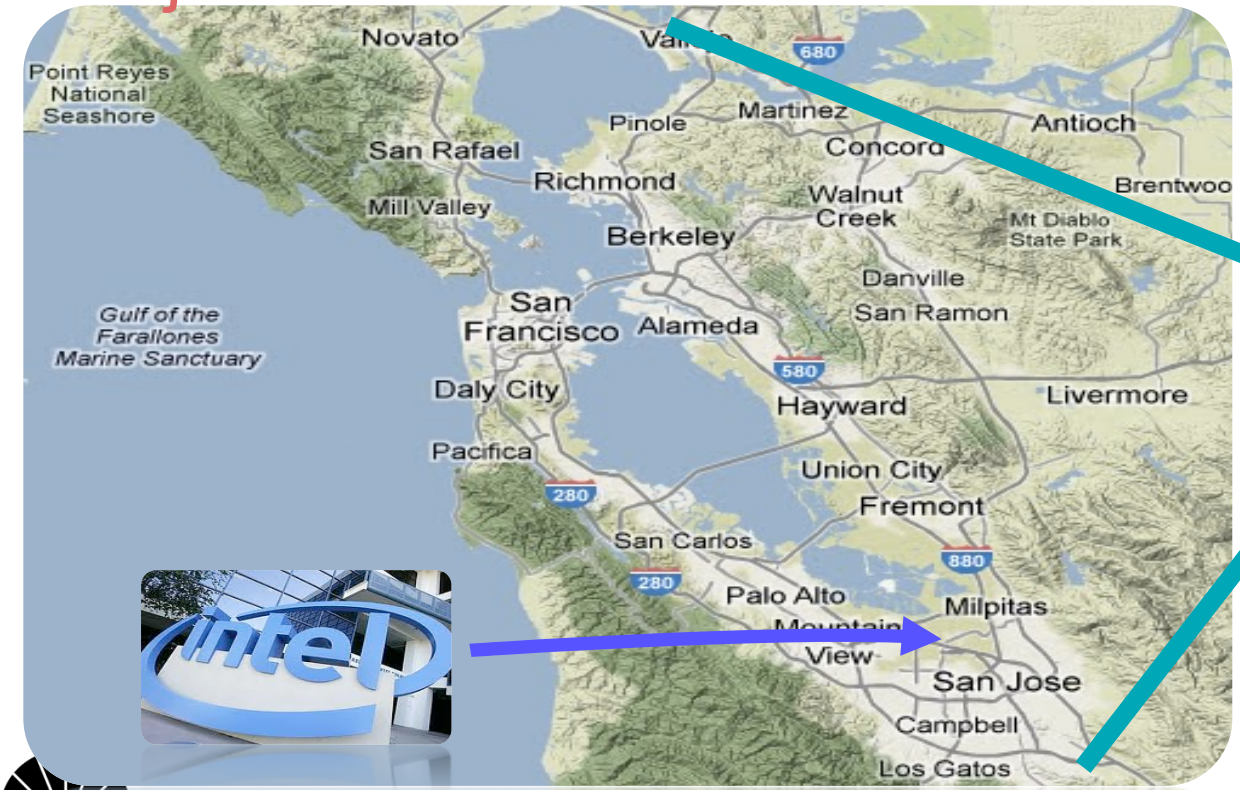
Trajetória

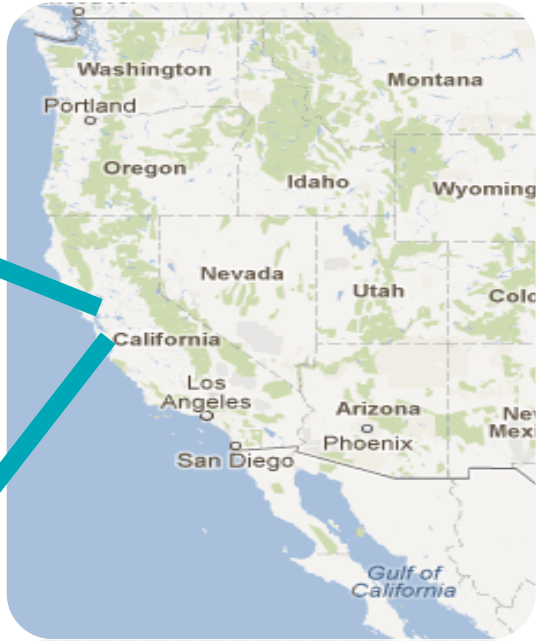




VALE DO SILÍCIO

Trajetória





VALE DO SILÍCIO

Trajetória

PARANAVAÍ

PORTO VELHO

CAMPO GRANDE

UFMS

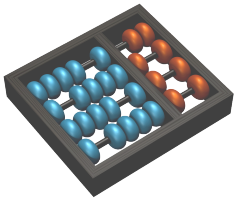
CAMPINAS

UNICAMP

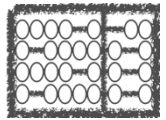
VALE DO SILÍCIO

Atualmente

- Instituto de Computação da Unicamp
- Laboratório Multidisciplinar de Computação de Alto Desempenho - LMCAD
- Computação paralela e de alto desempenho, aceleração de aplicações, arquitetura e microarquitetura de computadores, compiladores, IoT, ...

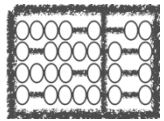


LMCAD

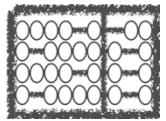


Ensino

Disciplina de pós-graduação: Técnicas para desenvolvimento e aceleração de aplicações científicas (Teoria e prática)

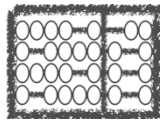


Análise de Aplicações e Otimização de Desempenho



Ciclo de otimização de código

Enquanto (**o desempenho não for bom o suficiente**)
{
  Identifique o trechos de código **frequentemente executados**;
  Otimize o trecho de código;
}

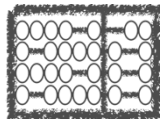


Agenda

Análise de desempenho

Detecção de código quente

Otimização de Código

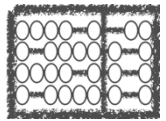


Análise de desempenho

Enquanto (**o desempenho não for bom o suficiente**)
{
  Identifique o trechos de código **frequentemente executados**;
  Otimize o trecho de código;
}

Como medir o desempenho?

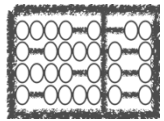
Como saber se já está bom o suficiente?



Análise de desempenho

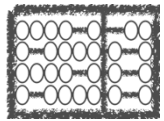
Como medir o desempenho? Usando uma métrica!

- GFlop/s,
- GB/s,
- MTransactions/s,
- Frames/s,
- ...



Análise de desempenho

Qual métrica? Depende do problema!



Análise de desempenho

Qual métrica? Depende do problema!

Multiplicação de matrizes: GFlop/s

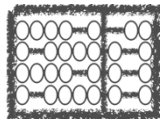
C[M][N] = A[M][K] x B[K][N]

de operações de ponto flutuante:

M * N * K * (1 + 1)

flop/s = M*N*K*2 / tempo de exec.

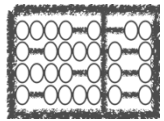
```
for (i = 0; i < (m); i++) {
   for (j = 0; j < (n); j++) {
     t = 0;
     for (p = 0; p < (k); p++) {
       t += A[i*k + p] * B[n*p + j];
     }
     C[j+i*n] = t ;
   }
 }
```



Análise de desempenho

Exemplo: Multiplicação de matrizes com M=1000, N=1000, e K=1000

de operações de ponto flutuante = M * N * K * (1 + 1) = $2x10^9$



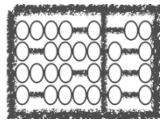
Análise de desempenho

Exemplo: Multiplicação de matrizes com M=1000, N=1000, e K=1000

de operações de ponto flutuante = M * N * K * (1 + 1) = $2x10^9$

Supondo que o tempo de execução foi 0.5s

Desempenho = $2x10^9$ / 0.5s = $4x10^9$ Flop/s = **4 GFlop/s**



Análise de desempenho

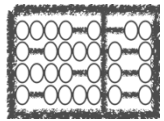
Exemplo: Multiplicação de matrizes com M=1000, N=1000, e K=1000

de operações de ponto flutuante = M * N * K * (1 + 1) = $2x10^9$

Supondo que o tempo de execução foi 0.5s

Desempenho = $2x10^9$ / 0.5s = $4x10^9$ Flop/s = **4 GFlop/s**

Como saber se já está bom o suficiente?



Análise de desempenho

Exemplo: Multiplicação de matrizes com M=1000, N=1000, e K=1000

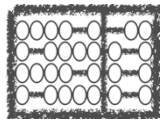
de operações de ponto flutuante = M * N * K * (1 + 1) = $2x10^9$

Supondo que o tempo de execução foi 0.5s

Desempenho = $2x10^9$ / 0.5s = $4x10^9$ Flop/s = **4 GFlop/s**

Como saber se já está bom o suficiente?

Ideia: Compare com o desempenho de pico da máquina utilizada!



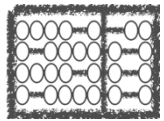
Análise de desempenho

Ideia: Compare com o desempenho de pico da máquina utilizada!

Minha máquina: 2.8 GHz, 2 núcleos, instruções AVX2 (16 operações de PF)

Total = $2.8 \times 10^9 \times 2 \times 16 = 89.6 \times 10^9$ Flop/s = 89.6 GFlops

89.6 GFlop/s vs 4 GFlop/s



Análise de desempenho

Ideia: Compare com o desempenho de pico da máquina utilizada!

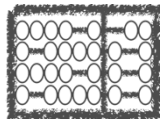
Minha máquina: 2.8 GHz, 2 núcleos, instruções AVX2 (16 operações de PF)

Total = $2.8 \times 10^9 \times 2 \times 16 = 89.6 \times 10^9$ Flop/s = 89.6 GFlops

89.6 GFlop/s vs 4 GFlop/s

É muito difícil atingir o desempenho de pico!!!

Como saber se já está bom o suficiente?



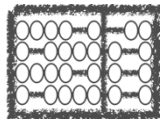
Análise de desempenho

Como saber se já está bom o suficiente?

Modelo de desempenho da aplicação!

O modelo deve levar em consideração:

- os recursos computacionais do *hardware*: Unidades vetoriais (SSE/AVX), número de núcleos computacionais, taxa de vazão da memória, ... ; e
- as otimizações aplicadas na aplicação: Paralelização, vetorização, ...



Análise de desempenho

Exemplo 2:

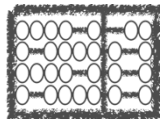
Minha máquina: 2.8 GHz, 2 núcleos, instruções AVX2 (16 operações de PF)

Total = 2.8 x 10^9 x 2 x 16 = 89.6 x 10^9 Flop/s = 89.6 GFlops

89.6 GFlop/s (Pico) vs 4 GFlop/s (Medido)

Se a aplicação não foi vetorizada e não foi paralelizada, então o desempenho de pico deveria ser: 3.3 GHz x 1 núcleo x 4 instruções de PF / ciclo = 13.2 GFlop/s

Podemos fazer melhor?

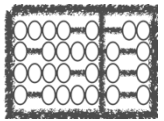


Análise de desempenho

Podemos olhar para as instruções do núcleo computacional (laços da multiplicação) e computar as dependências e o paralelismo entre instruções e verificar como isso afeta o número máximo de instruções que podem ser executadas em paralelo (ILP = Instruction Level Parallelism).

Supondo que o ILP seja 2:

Desempenho: 3.3 GHz x 1 núcleo x ~~4~~ 2 instruções de PF / ciclo = ~~13.2~~ 6.6 GFlop/s



Análise de desempenho

Memória: fonte comum de gargalo de desempenho.

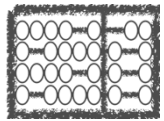
Quando for este o caso, otimizações que afetam o desempenho do código no processador não causam muito efeito!

Exemplo: Minha máquina

CPU = 2.8 x 10^9 x 2 x 16 = 89.6 x 10^9 Flop/s = 89.6 GFlops

Vazão da memória: 25.6 GB/s

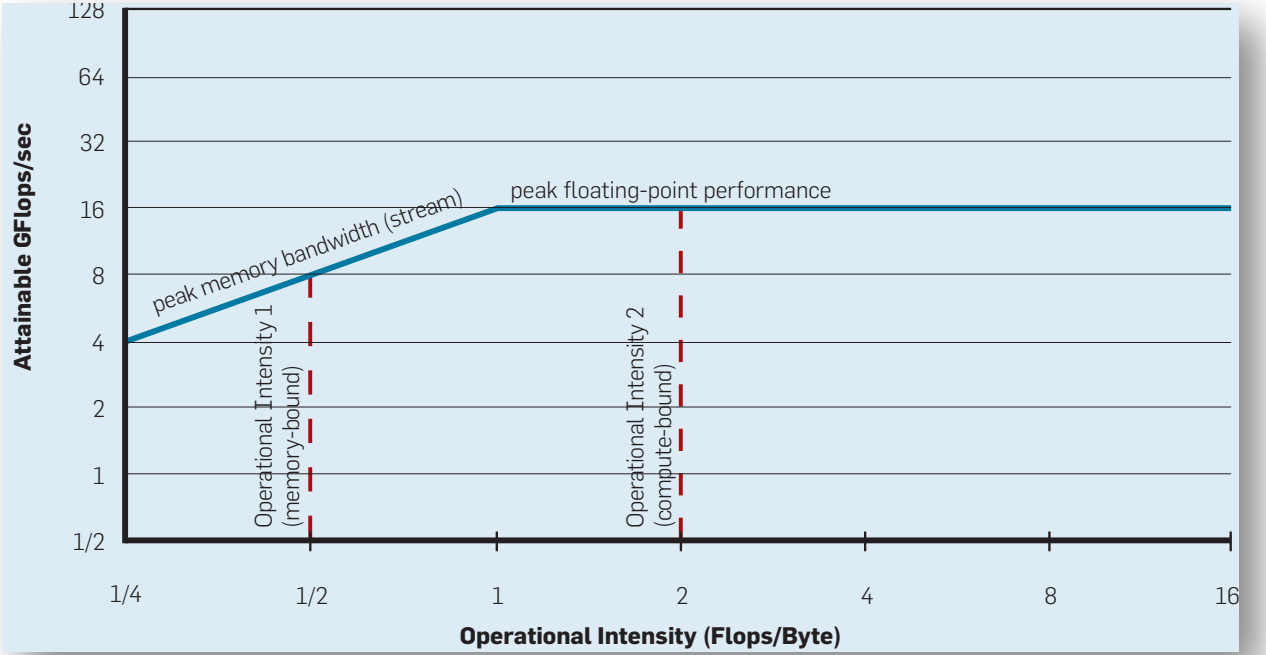
Relação entre CPU e memória: 3.5 operações de PF para cada *byte* trazido da memória



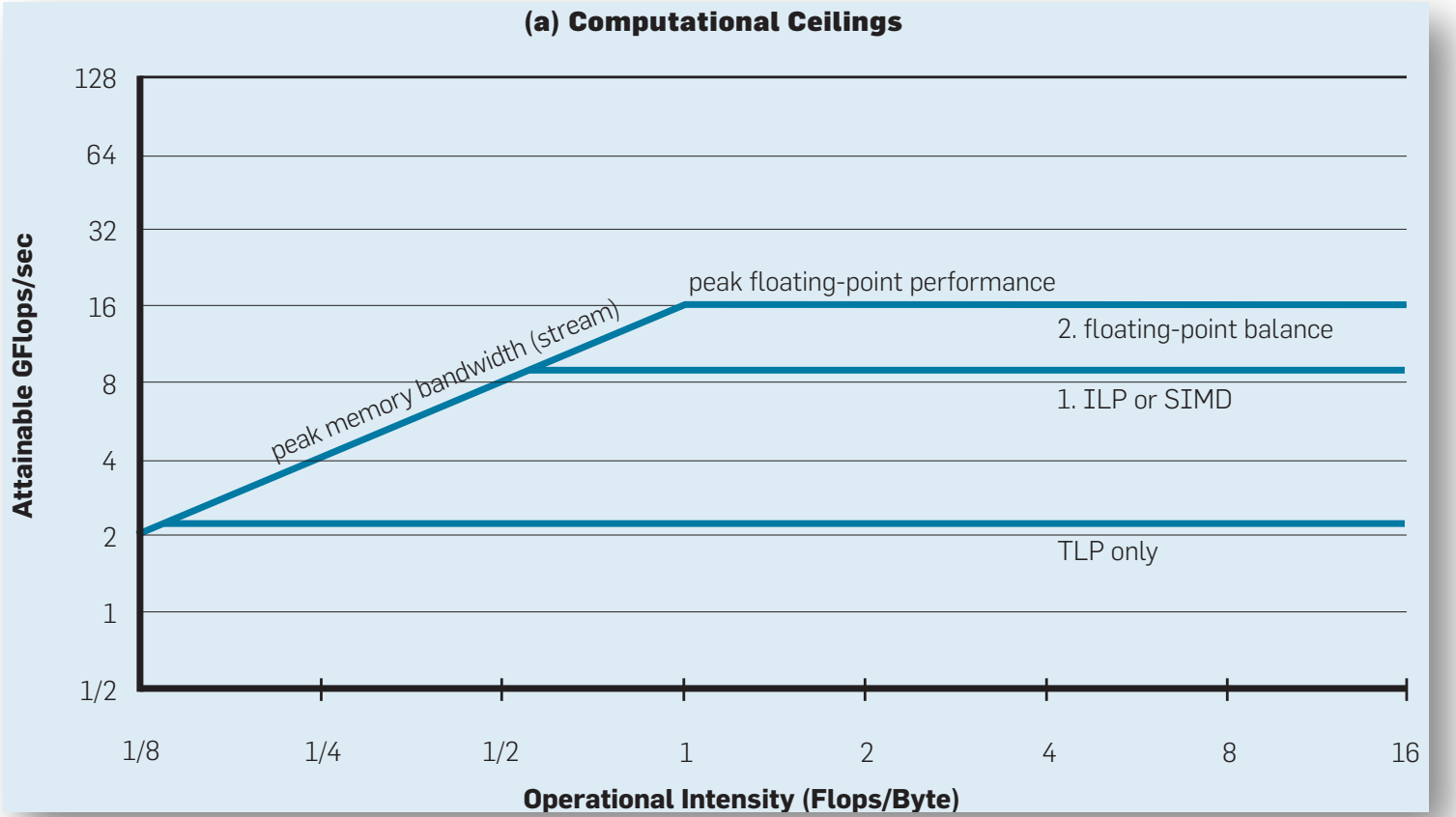
Análise de desempenho

Modelo Roofline

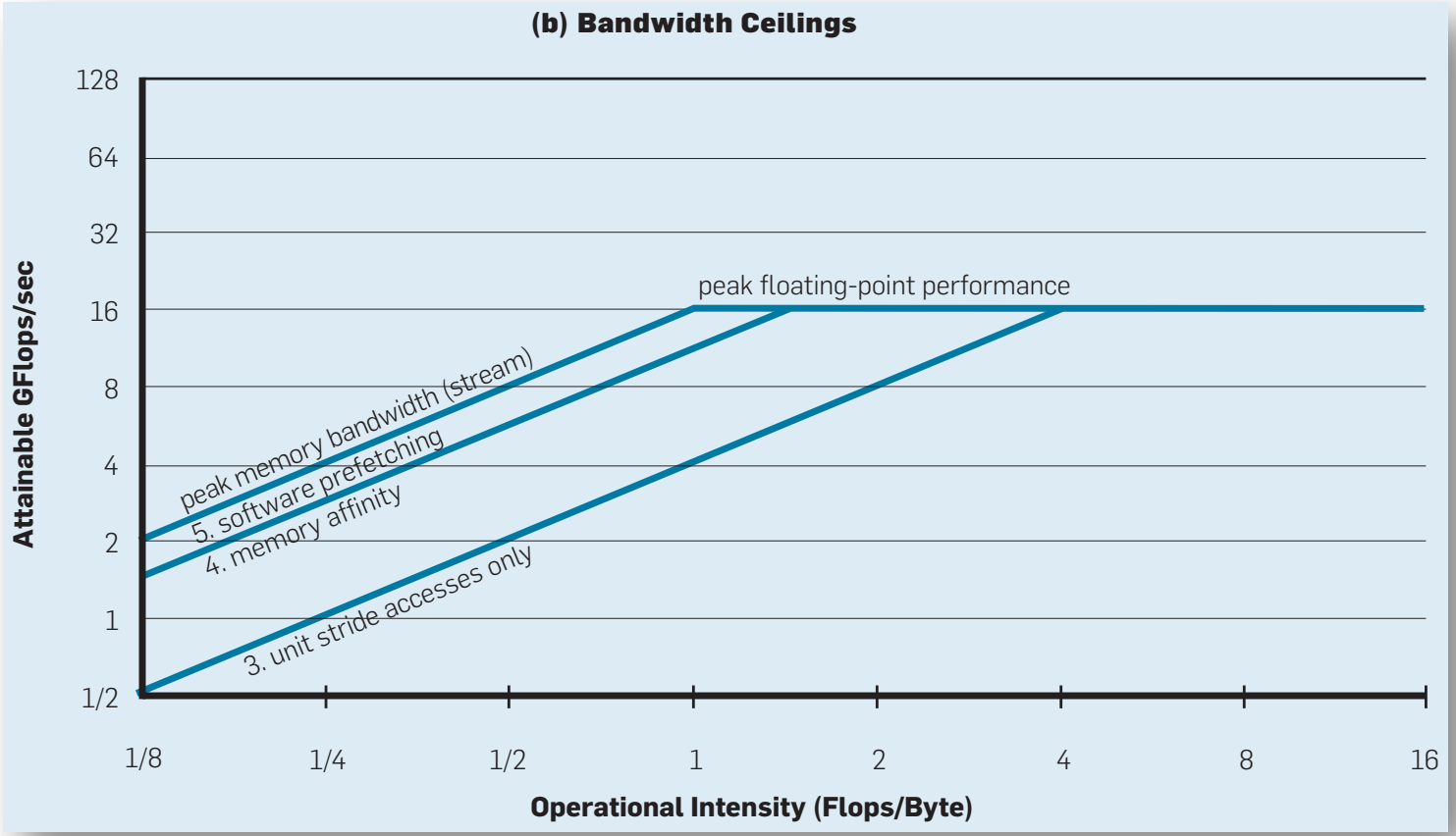
Williams, Waterman and Patterson. Roofline: An Insightful Visual Performance Model for Multicore Architectures. Communications of the ACM. Apr. 2009



Análise de desempenho

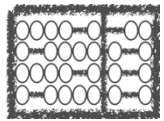


Análise de desempenho



Cuidados com o uso de aceleradores

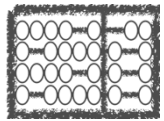
Barramento PCI é lento => Ganhos de processamento obtido pelo acelerador (GPU) pode ser sobreposto pela perda de desempenho acarretada pela transferência de dados da memória principal para a memória do acelerador!



Medindo tempo de execução

Métricas de desempenho: X / s

- X pode ser Flop, Transactions, Frames, ...

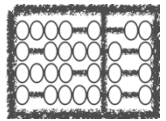


Medindo tempo de execução

Métricas de desempenho: X / s

- X pode ser Flop, Tr[illegible]tions, Frames, ...

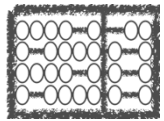
Temos que medir o tempo de execução da aplicação ou de trechos de interesse na aplicação.



Medindo tempo de execução

Várias abordagens:

- Cronômetro manual!
- /usr/bin/time ./my_app.bin
- gettimeofday()
- rdtsc
- libraries: PAPI, ...



Medindo tempo de execução

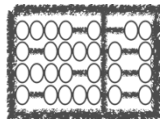
A ferramenta /usr/bin/time:

$ /usr/bin/time –p ./my_prog.bin

real 0.62

user 0.55

sys 0.07



Medindo tempo de execução

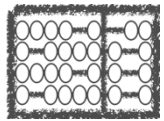
A ferramenta /usr/bin/time:

$ /usr/bin/time –p ./my_prog.bin

real 1.14

user 2.00

sys 0.14



Medindo tempo de execução

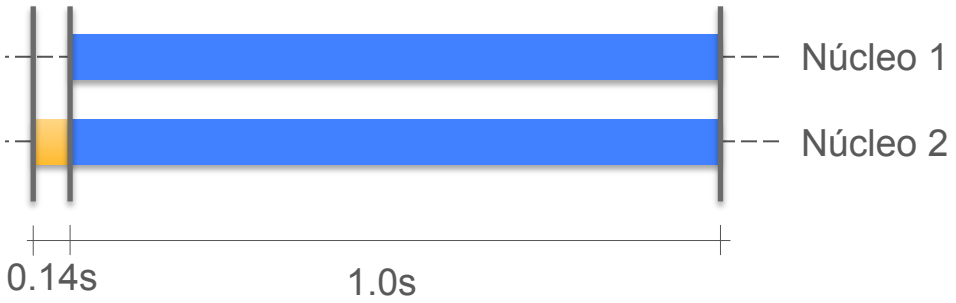
A ferramenta /usr/bin/time:

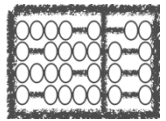
$ /usr/bin/time –p ./my_prog.bin

real 1.14

user 2.00

sys 0.14

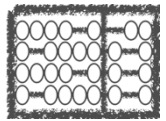




Medindo tempo de execução

gettimeofday() – *UNIX-like systems* => Chama o sistema operacional

```
#include <sys/time.h>
double mysecond() {
  struct timeval tp;
  struct timezone tzp;
  gettimeofday(&tp,&tzp);
  return ((double) tp.tv_sec +
          (double) tp.tv_usec * 1.e-6 );
}
```

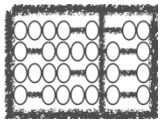


Medindo tempo de execução

gettimeofday() – *UNIX-like systems* => Chama o sist

```
#include <sys/time.h>
double mysecond() {
  struct timeval tp;
  struct timezone tzp;
  gettimeofday(&tp,&tzp);
  return ((double) tp.tv_sec +
          (double) tp.tv_usec * 1.e-6 );
}
```

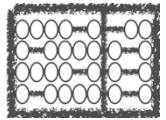
Tempo de chamada ao sistema (*system call*) não é desprezível, portanto, não é adequado para se medir o tempo de trechos de código que executam em pouco tempo (< ms)



Medindo tempo de execução

rdtsc – instrução do processador => Lê o TSC (*Time Stamp Counter*)

```
unsigned long long getticks(void)
{
  unsigned a, d;
  asm("cpuid");
  asm volatile("rdtsc" : "=a" (a), "=d" (d));
  return (((ticks)a) | (((ticks)d) << 32));
}
```

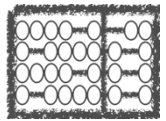


Medindo tempo de execução

rdtsc – instrução do processador => Lê o TSC (*Time Stamp*

Cuidado com interrupções entre medidas!

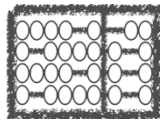
```
unsigned long long getticks(void)
{
  unsigned a, d;
  asm("cpuid");
  asm volatile("rdtsc" : "=a" (a), "=d" (d));
  return (((ticks)a) | (((ticks)d) << 32));
}
```



Cuidados com variabilidade e média

Diversos fatores em um sistema computacional podem afetar o tempo de execução:

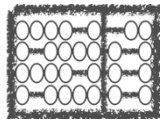
- **Estruturas de *caching* do *hardware* (Processador, Hard Drive, etc...) e de *software* (sistema operacional, bibliotecas, ...)**



Cuidados com variabilidade e média

Diversos fatores em um sistema computacional podem afetar o tempo de execução:

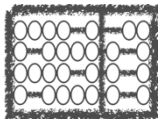
- Estruturas de *caching* do *hardware* (Processador, Hard Drive, etc...) e de *software* (sistema operacional, bibliotecas, ...)
- **Ajustes dinâmicos de frequência de operação para regular o consumo de energia (Turbo Boost, ...)**



Cuidados com variabilidade e média

Diversos fatores em um sistema computacional podem afetar o tempo de execução:

- Estruturas de *caching* do *hardware* (Processador, Hard Drive, etc...) e de *software* (sistema operacional, bibliotecas, ...)
- Ajustes dinâmicos de frequência de operação para regular o consumo de energia (Turbo Boost, ...)
- **Outros programas sendo executados ao mesmo tempo que seu programa (*kernel* e módulos do sistema operacional, ...)**
- ...

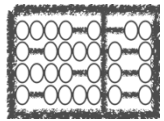


Cuidados com variabilidade e média

Experimento:

- Aplicação test.c
- 50 amostras

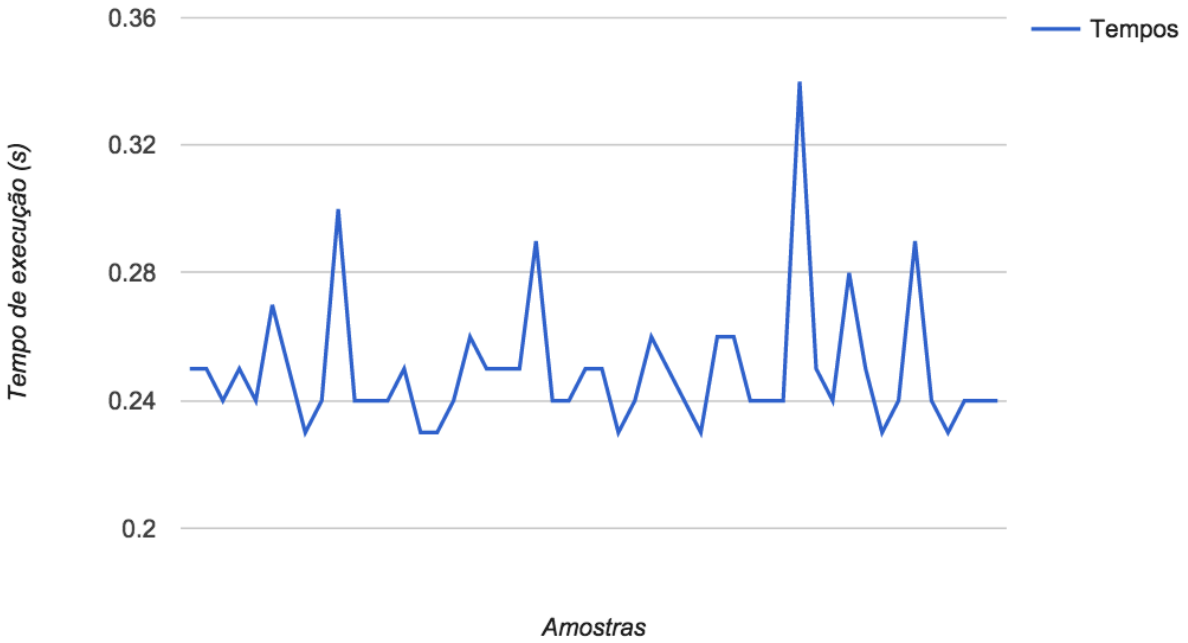
```
int a = 0;
int main(){
  for (int i=0; i<100000000; i++ ) {
       a = a+i;
  }
  return 0;
}
```

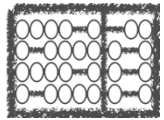


Cuidados com variabilidade e média

Experimento:

- Aplicação test.c
- 50 amostras

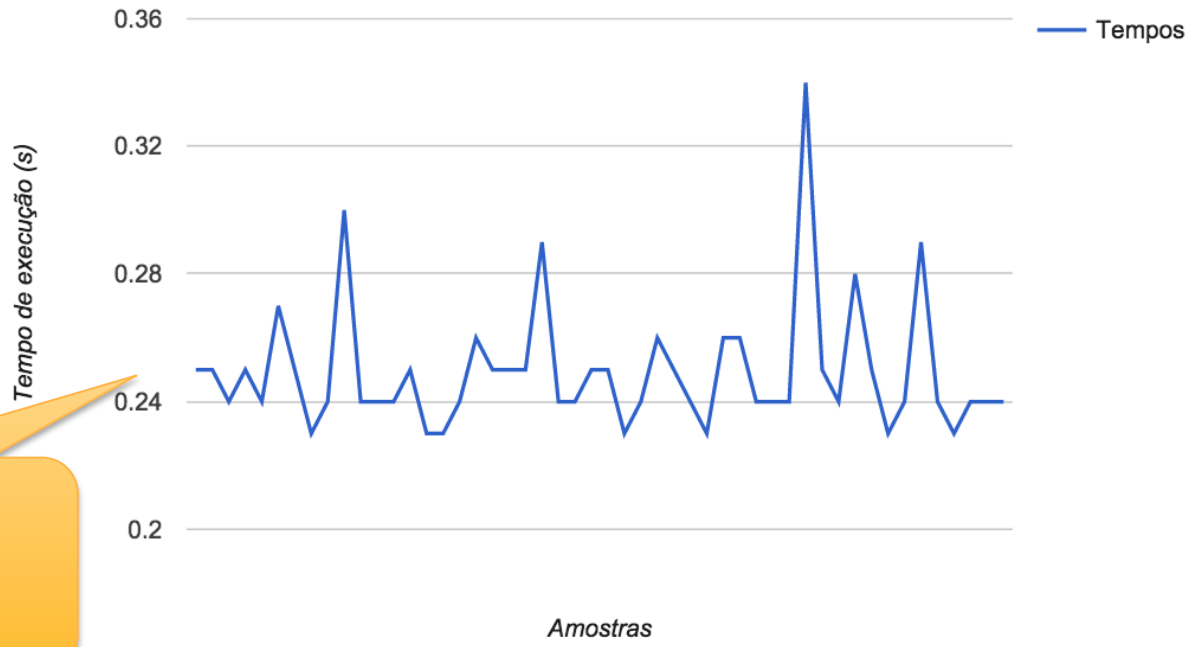




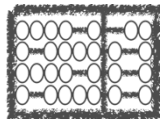
Cuidados com variabilidade e média

Experimento:

- Aplicação test.c
- 50 amostras



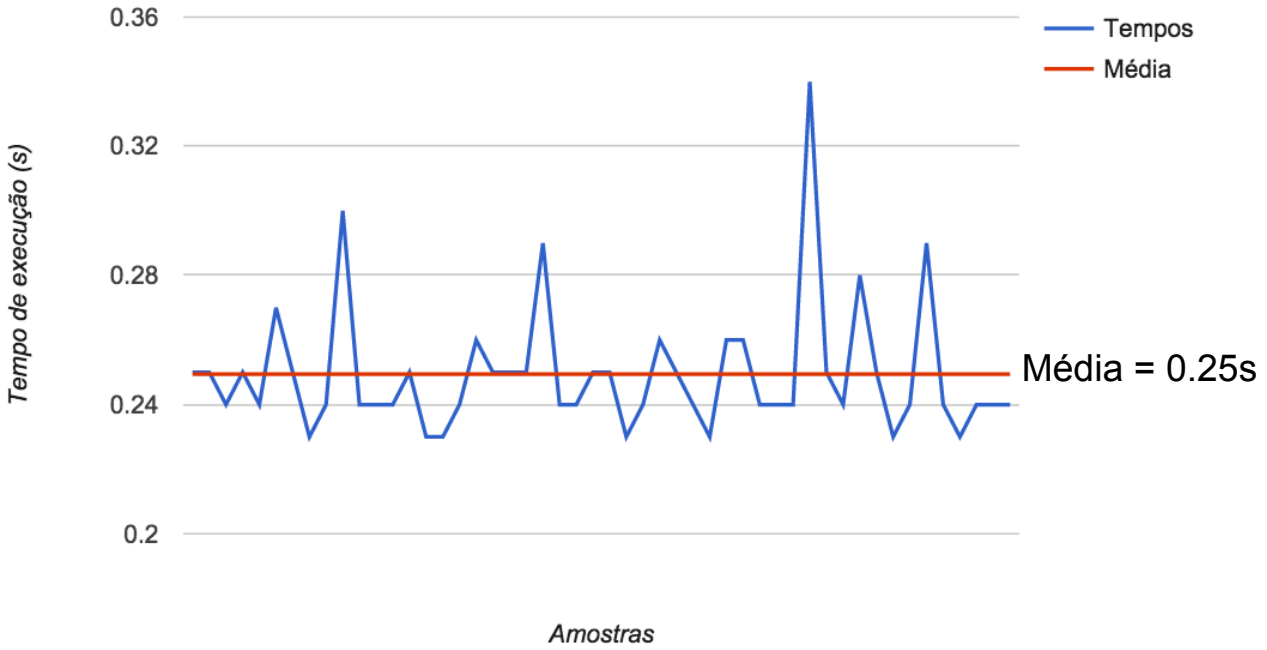
Como resumir o resultado do experimento?

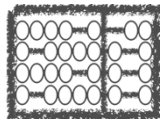


Cuidados com variabilidade e média

Experimento:

- Aplicação test.c
- 50 amostras
- Média = 0.25s

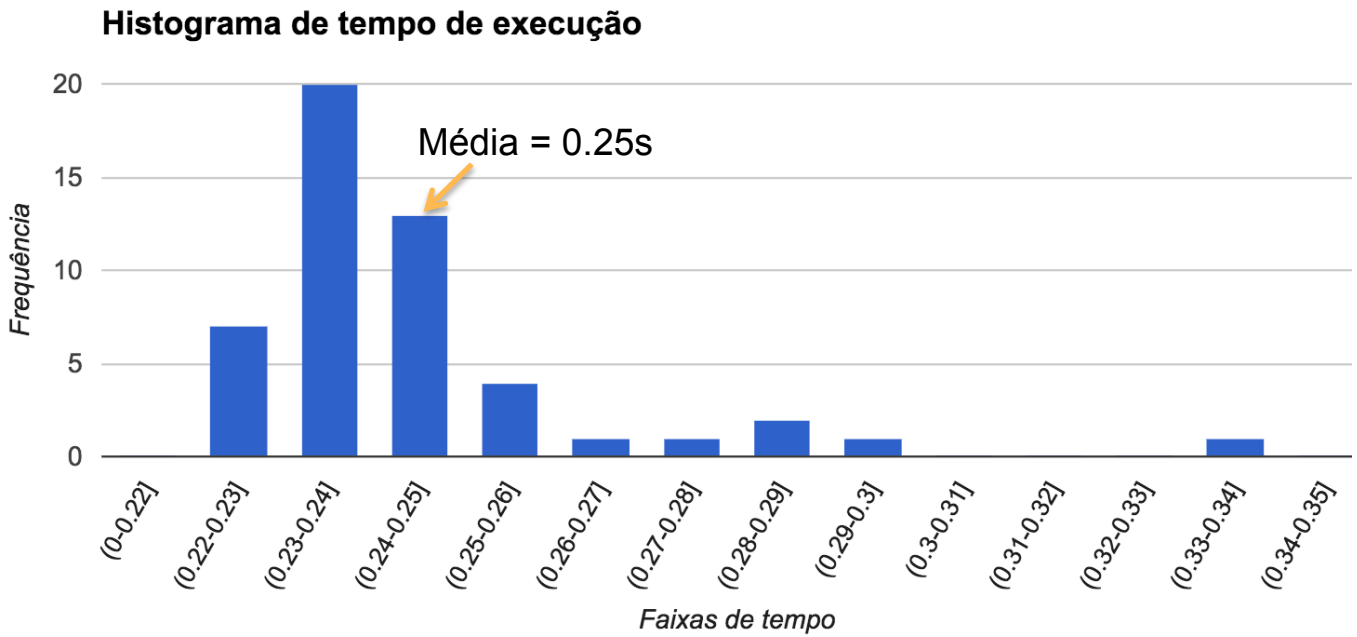




Cuidados com variabilidade e média

Experimento:

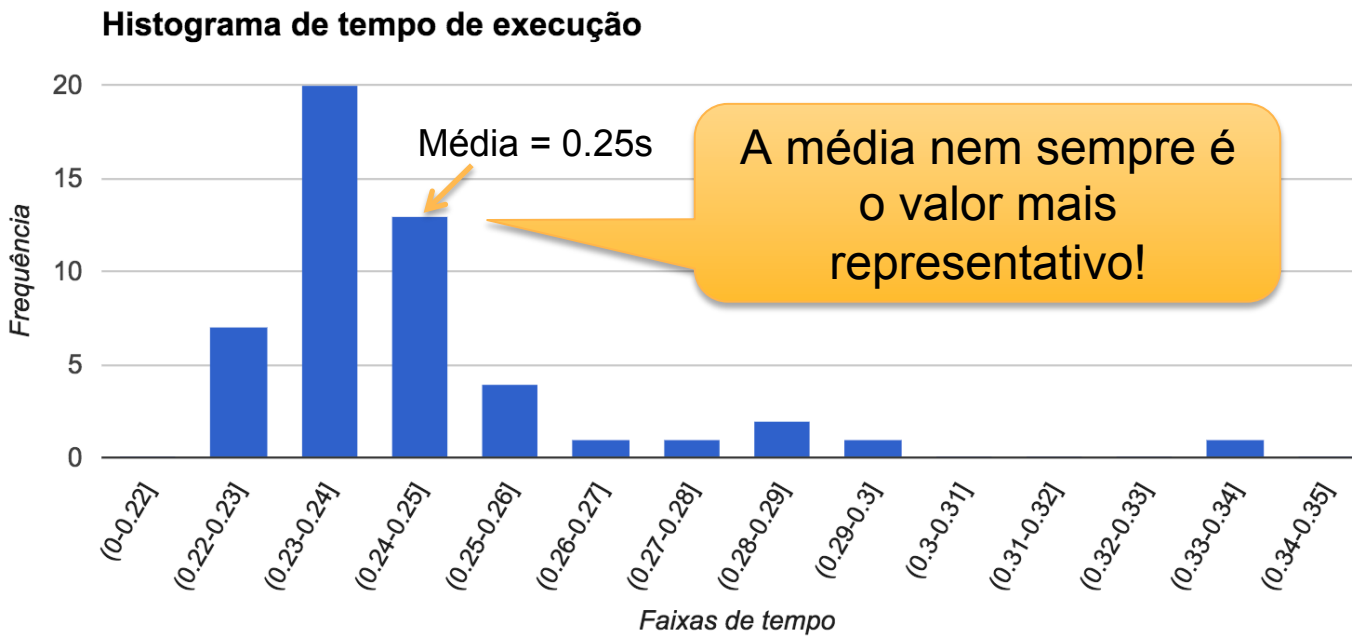
- Aplicação test.c
- 50 amostras
- Média = 0.25s



Cuidados com variabilidade e média

Experimento:

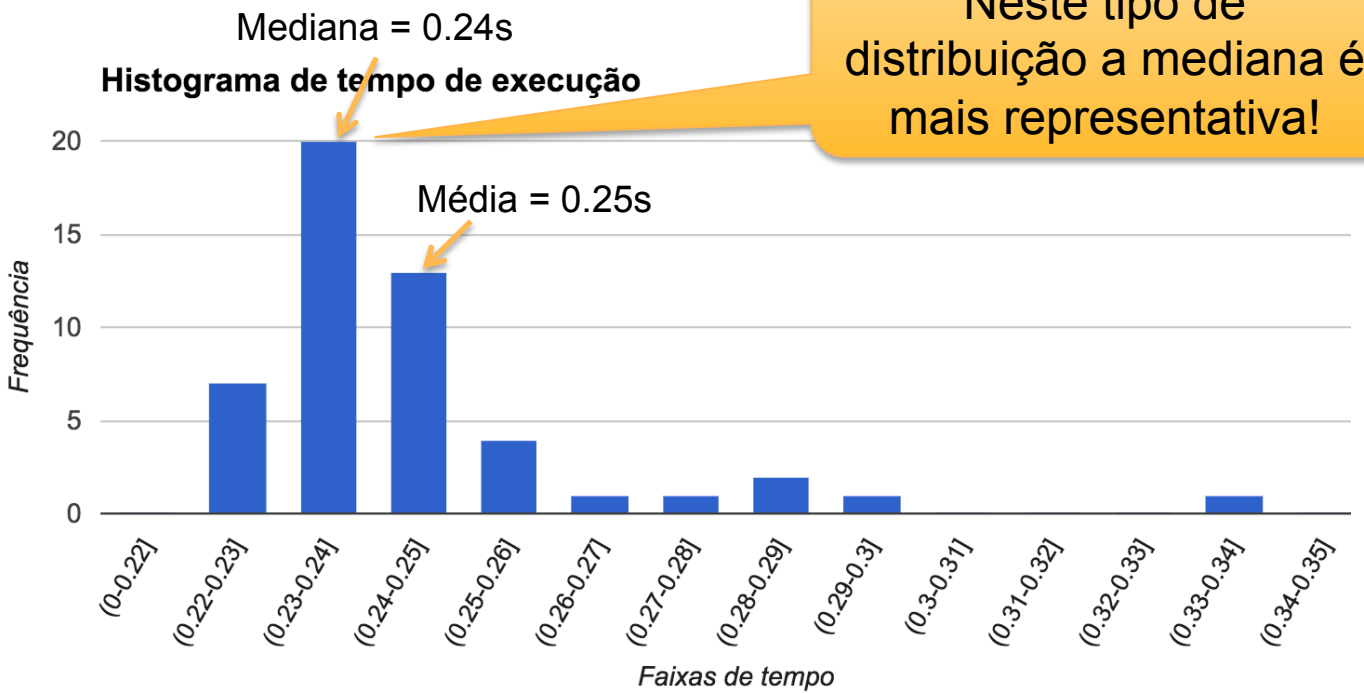
- Aplicação test.c
- 50 amostras
- Média = 0.25s



Cuidados com variabilidade e média

Experimento:

- Aplicação test.c
- 50 amostras
- Média = 0.25s
- Mediana = 0.24s



Neste tipo de distribuição a mediana é mais representativa!

Cuidados com variabilidade e média

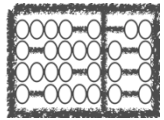
Experimento:

- Aplicação test.c
- 50 amostras
- Média = 0.25s
- Mediana = 0.24s

Cuidados com variabilidade e média

Dicas para realizar e reportar resultados experimentais de desempenho!

- Execute o experimento múltiplas vezes, se possível em diferentes sistemas!
- Analise a distribuição das amostras!
- Se for representativa, use a mediana ou a média.
- Medidas de dispersão (desvio padrão, intervalo de confiança, ...) também podem ser úteis => cuidado com a distribuição

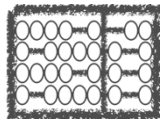


Agenda

Análise de desempenho

Detecção de código quente

Otimização de Código



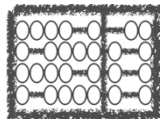
Detectando código quente

Regra 90/10: 90% do tempo de execução é gasto em 10% do código.

- Laços, funções recursivas, etc.

Código quente = código frequentemente executado

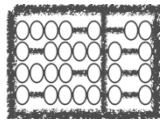
Estratégia de otimização: identificar, analisar e melhorar código quente



Detectando código quente

Amostragem do PC (*Program Counter*) com base no tempo (Abordagem Manual)

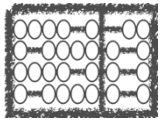
```
Abra seu programa com o depurador GDB;
enquanto (o programa não terminar)
{
  interrompa a execução após um tempo (ctrl+c)
  guarde o valor do PC
  continue a execução
}
construa um histograma usando as amostras de PC
```



Detectando código quente

Amostragem do PC (*Program Counter*) com base no tempo (Abordagem Manual)

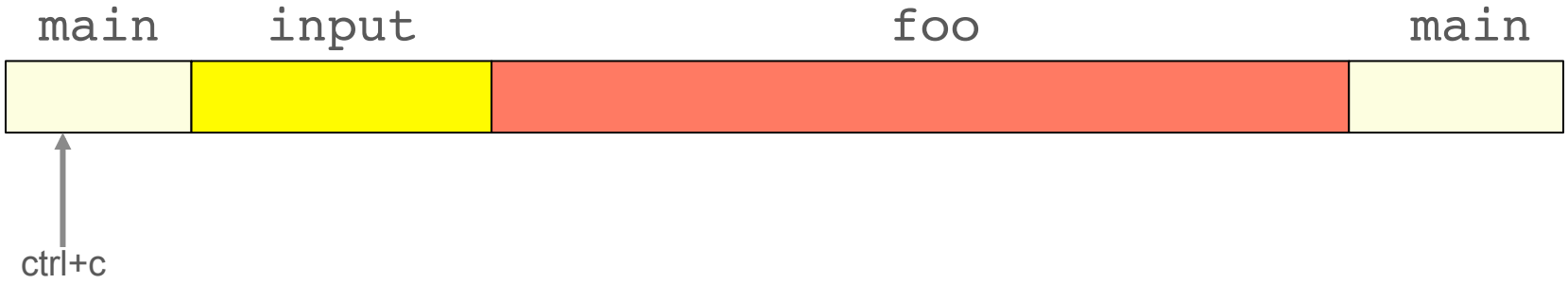
Exemplo:

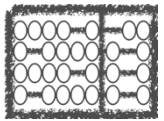


Detectando código quente

Amostragem do PC (*Program Counter*) com base no tempo (Abordagem Manual)

Exemplo:

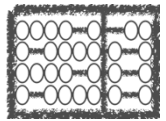




Detectando código quente

Amostragem do PC (*Program Counter*) com base no tempo (Abordagem Manual)

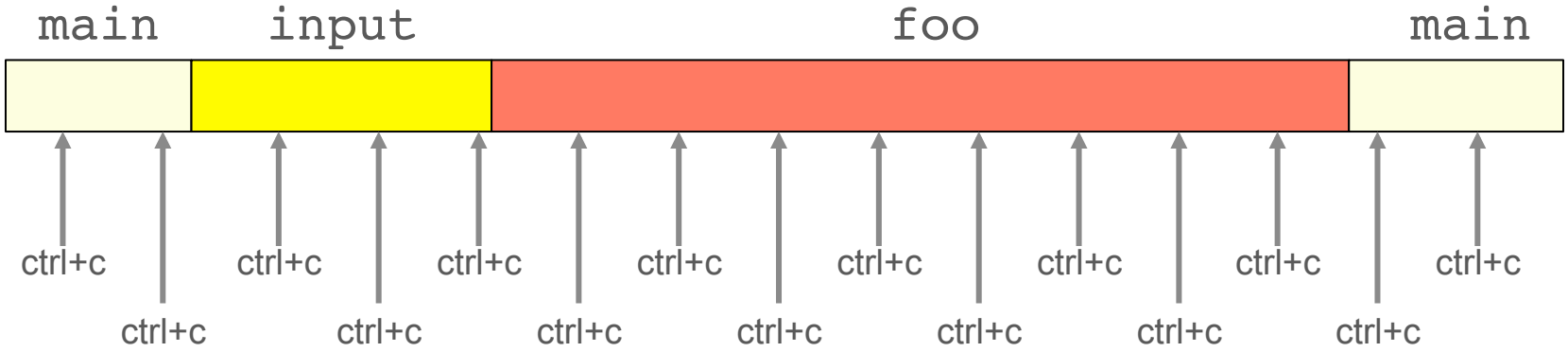
Exemplo:

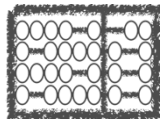


Detectando código quente

Amostragem do PC (*Program Counter*) com base no tempo (Abordagem Manual)

Exemplo:



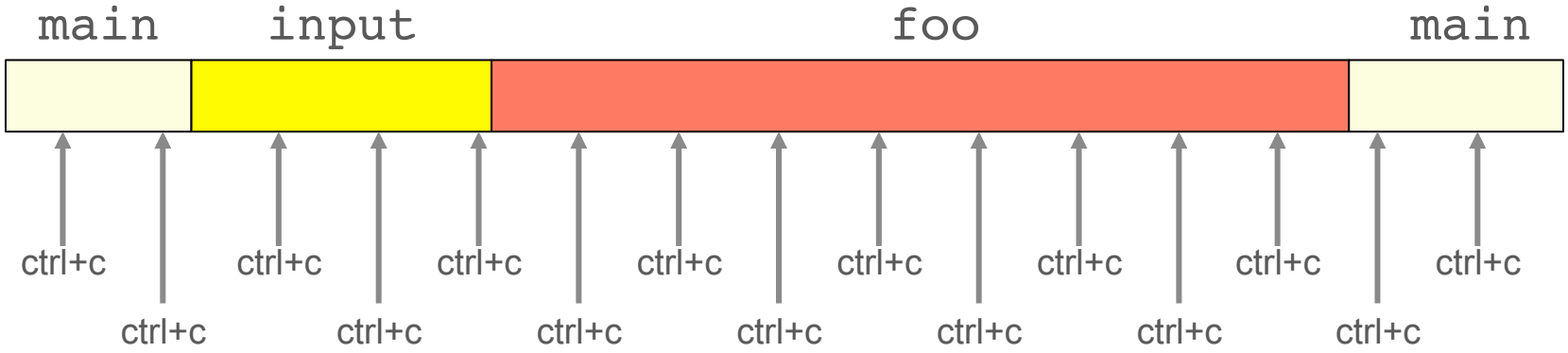


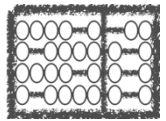
Detectando código quente

Amostragem do PC (*Program Counter*) com base

Exemplo:

main: 4 (27%)
input: 3 (20%)
foo: 8 (53%)





Detectando código quente

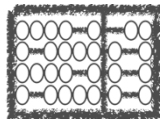
Amostragem do PC (*Program Counter*) com base no tempo

A abordagem manual não é muito prática

Perfilador (*Profiler*): ferramenta que nos ajuda a traçar o perfil de execução de aplicações

Diversas ferramentas: VTune, CodeAnalyst, perf, ...

```
perf record ./my_prog.bin
```

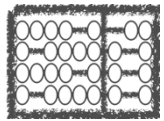


Detectando código quente

Amostragem do PC (*Program Counter*) com base no tempo (perf)

perf coleta amostras do PC de tempos em tempos e grava no arquivo perf.data

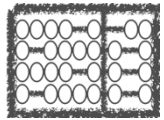
Para inspecionar amostras com o comando `perf script`



Detectando código quente

Amostragem do PC (*Program Counter*) com base no tempo (perf)

```
$ perf script
  test.x  3559  1766.975400: 250000 cpu-clock:            400500 main (/home ... test.x
  test.x  3559  1766.975698: 250000 cpu-clock:            400505 main (/home ... test.x
  test.x  3559  1766.976159: 250000 cpu-clock:            40050b main (/home ... test.x
  test.x  3559  1766.983328: 250000 cpu-clock:            400505 main (/home ... test.x
  test.x  3559  1766.992301: 250000 cpu-clock:            4004fa main (/home ... test.x
  test.x  3559  1766.998935: 250000 cpu-clock:            400500 main (/home ... test.x
  test.x  3559  1767.001108: 250000 cpu-clock:            4004fa main (/home ... test.x
  test.x  3559  1767.006936: 250000 cpu-clock:  ffffffff810a4c4a [unknown] ([unknown])
  test.x  3559  1767.009763: 250000 cpu-clock:            400500 main (/home ... test.x
  test.x  3559  1767.011107: 250000 cpu-clock:            40050b main (/home ... test.x
  ...
```



Detectando código quente

Amostragem do PC (*Program Counter*) com base no tempo (perf)

```
$ perf script
  test.x  3559  1766.975400: 250000 cpu-clock:          400500 main (/home ... test.x
  test.x  3559  1766.975698: 250000 cpu-clock:          400505 main (/home ... test.x
  ...
```

```
$ perf script | cut -d: -f3 | sort -n | uniq –c

      1   ffffffff81081e21 [unknown] ([unknown])
      2   ffffffff810a4c4a [unknown] ([unknown])
     47            4004fa main (/home/mc404/Desktop/ERAD17-Tutorial/test.x)
     48            40050b main (/home/mc404/Desktop/ERAD17-Tutorial/test.x)
    141            400500 main (/home/mc404/Desktop/ERAD17-Tutorial/test.x)
      2            400503 main (/home/mc404/Desktop/ERAD17-Tutorial/test.x)
     69            400505 main (/home/mc404/Desktop/ERAD17-Tutorial/test.x)
      9            400516 main (/home/mc404/Desktop/ERAD17-Tutorial/test.x)
```

histograma

Detectando código quente

`perf report` mostra esta informação de forma organizada!

Amostragem do PC (*Program Counter*) com base no tempo (perf)

```
mc404@mc404-VirtualBox: ~/Desktop/ERAD17-Tutorial
Samples: 319  of event 'cpu-clock', Event count (approx.): 79750000
Overhead  Command  Shared Object      Symbol
  99,06%  test.x   test.x            [.] main
   0,63%  test.x   [unknown]         [k] 0xffffffff810a4c4a
   0,31%  test.x   [unknown]         [k] 0xffffffff81081e21

For a higher level overview, try: perf report --sort comm,dso
```

```
$ perf s
  test.x
  test.x
  ...
```

```
$ perf s

     1
     2
    47
    48
   141
     2
    69
     9
```

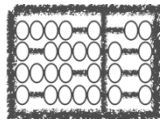
Detectando código quente

Amostragem do PC (*Program Counter*) com base no tempo (perf)

Muitas vezes, identificar a função mais quente já é o suficiente!

Em outros casos, precisamos identificar qual trecho dentro de uma função é o código mais quente.

- O `perf report` pode nos informar quais trechos da função são os mais quentes!



Detectando código quente

Amostra

Muitas v

Em outr é o código m

- O pe ais quent

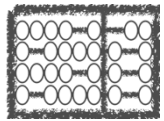
Detectando código quente

Amostragem do PC (*Program Counter*) com base no tempo (perf)

E se quisermos identificar trechos de programa que:

- Causam muitas faltas na *cache* do processador (*cache misses*)?
- Causam muitos erros de predição de desvio (*branch mispredictions*)?
- Causam muitas faltas de páginas (*page faults*)?

Podemos utilizar o `perf` também!



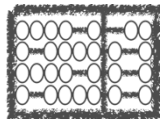
Detectando código quente

Amostragem do PC (*Program Counter*) com base em X, onde X é:

- *cache misses*, *branch mispredictions*, *page faults*, ...
- perf list : lista todos os eventos disponíveis

```
perf record -e page-faults ./my_prog.bin
```

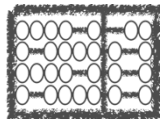
- Grava amostras do PC em função dos eventos *page-faults*!



Detectando código quente

Outras ferramentas

- Instruments: Apple OSX
- VTune: Intel (Linux, Windows)
- CodeAnalyst: AMD
- gprof
- ...

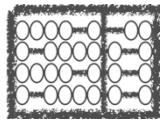


Agenda

Análise de desempenho

Detecção de código quente

Otimização de Código



Aceleração de Aplicações

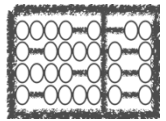
Otimizações Simples

Otimizações na Compilação

Vetorização de Código

Bibliotecas Otimizadas

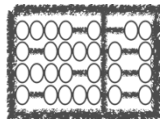
Otimizações de Acesso a Dados



Otimizações Simples

Faça menos trabalho!

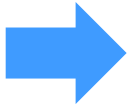
```
bool flag = false;
for (i=0; i<N; i++)
{
  if (A[i] > 5.0)
     flag = true;
}
return flag;
```



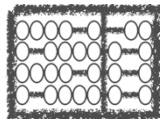
Otimizações Simples

Faça menos trabalho!

```
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;
```

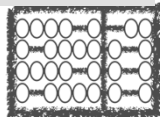


Otimizações Simples

Evite operações custosas!

```
int ang;

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

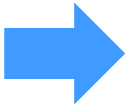
```
int ang;

for (...) {
        ang = calcula_angulo();
        res += tan_table[ang%360];
}
double tan_table[] = {
        0.0,
        ....
}
```



Otimizações Simples

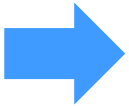
Quando possível, diminua o tamanho dos dados!

```
double a[N];
double b[N];
double c[N];
```



```
float a[N];
float b[N];
float c[N];
```

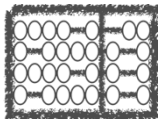
```
int IDs[N];
```



```
short IDs[N];
```

Cabem mais dados nas *caches*!

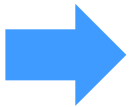
Mais operações por ciclo em instruções SIMD.



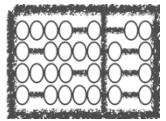
Otimizações Simples

Elimine sub-expressões comuns.

```
int r, s;
...

for (i=0; i<N; i++)
{
  A[i] = A[i] + r + s;
}
```



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

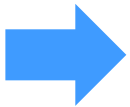


Otimizações Simples

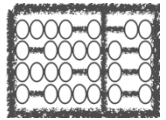
Elimine sub-expressões comuns.

Compiladores são capazes de realizar esta otimização!

```
int r, s;
...

for (i=0; i<N; i++)
{
  A[i] = A[i] + r + s;
}
```



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

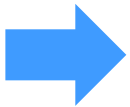


Otimizações Simples

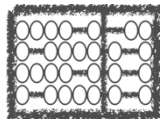
Elimine sub-expressões comuns.

Será que o compilador é capaz de realizar esta otimização?

```
int r, s;
...

for (i=0; i<N; i++)
{
  A[i] = A[i] + r + foo();
}
```



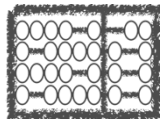
```
int r, s;
...
int tmp = r+foo();
for (i=0; i<N; i++)
{
  A[i] = A[i] + tmp;
}
```



Otimizações Simples

Evite saltos condicionais.

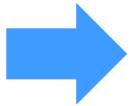
```
float sinal;
for (i=0; i<N; i++)
  for (j=0; j<N; j++) {
    if(i>j)
      sinal = 1.0;
    else if(i<j)
      sinal = -1.0;
    else
      sinal = 0.0;
    acc += sinal x A[i][j];
  }
```



Otimizações Simples

Evite saltos condicionais.

```
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];
```

Aceleração de Aplicações

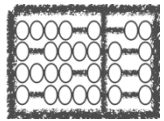
Otimizações Simples

Otimizações na Compilação

Vetorização de Código

Bibliotecas Otimizadas

Otimizações de Acesso a Dados



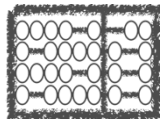
Otimizações na Compilação

Use as otimizações de seu compilador.

Muitas otimizações => difícil selecionar o melhor conjunto e a melhor ordem...

Dica: comece com um conjunto padrão: -O3

- gcc –O3 ...
- clang –O3 ...



Otimizações na Compilação

Use as otimizações de seu compilador.

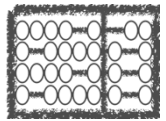
Muitas otimizações => difícil selecionar o melhor conjunto e a melhor ordem...

Dica: comece com um conjunto padrão: -O3

Talvez seja necessário habilitar **mais opções**! Ex:

gcc4.7 -O3 -march=corei7-avx -mtune=corei7-avx ...

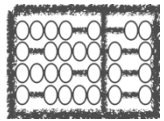
- Gera instruções **AVX**! (SIMD/256 bits)



Otimizações na Compilação

Use os relatórios (*logs*) de compilação!

- gcc4.7 -O3 -**ftree-vectorizer-verbose**=1 test.c -c

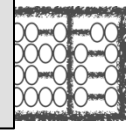


Otimizações na Compilação

Use os relatórios (*logs*) de compilação!

- gcc4.7 -O3 -**ftree-vectorizer-verbose**=1 test.c -c

```
Analyzing loop at test.c:9
Vectorizing loop at test.c:9
9: created 2 versioning for alias checks.
9: LOOP VECTORIZED.
test.c:6: note: vectorized 1 loops in function.

Analyzing loop at test.c:16
Vectorizing loop at test.c:16
16: created 1 versioning for alias checks.
16: LOOP VECTORIZED.
test.c:13: note: vectorized 1 loops in function.
```



Aceleração de Aplicações

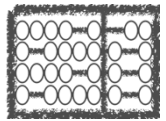
Otimizações Simples

Otimizações na Compilação

Vetorização de Código

Bibliotecas Otimizadas

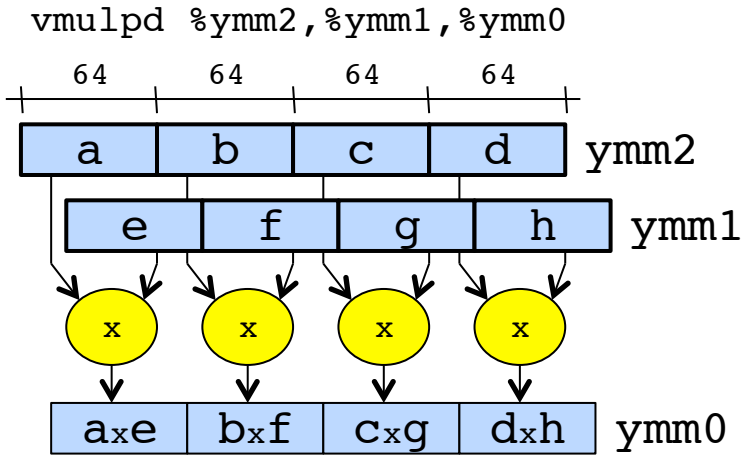
Otimizações de Acesso a Dados

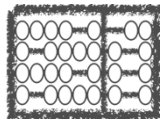


Vetorização de Código

Vetorização: usar instruções SIMD para realizar operações de forma paralela!

Exemplo:

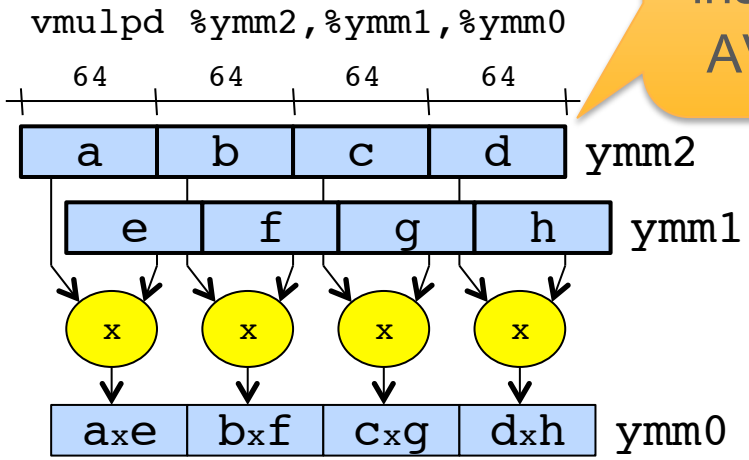




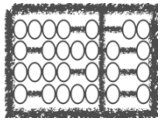
Vetorização de Código

Vetorização: usar instruções SIMD para realizar operações de forma paralela!

Exemplo:



Diversos conjuntos de instruções: SSE, SSE2, AVX, AVX2, NEON, ...

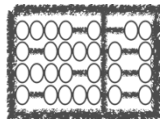


Vetorização de Código

Vetorização: geração de código manual: "*compiler intrinsics*"

Exemplo:

```
float produto_interno(float* a, float* b)
{
        int i;
        float total=0.0;
        for (i=0; i<SIZE; i++)
                total += a[i] * b[i];
        return total;
}
```

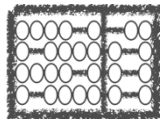


Vetorização de Código

Vetorização: geração de código manual: "*compiler intrinsics*"

```
for (i=0; i<SIZE; i++)
        total += a[i] * b[i];
```

```
for (i=0; i<SIZE; i+=4){
        total[0] += a[i+0] * b[i+0];
        total[1] += a[i+1] * b[i+1];
        total[2] += a[i+2] * b[i+2];
        total[3] += a[i+3] * b[i+3];
}
```

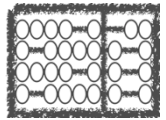


Vetorização de Código

Vetorização: geração de código manual: "*compiler intrinsics*"

```
for (i=0; i<SIZE; i+=4){
        total[0] += a[i+0] * b[i+0];
        total[1] += a[i+1] * b[i+1];
        total[2] += a[i+2] * b[i+2];
        total[3] += a[i+3] * b[i+3];
}
```

```
for (i=0; i<SIZE; i+=4){
        v1  = _mm_loadu_ps(a+i);
        v2  = _mm_loadu_ps(b+i);
        v3  = _mm_mul_ps(v1, v2);
        acc = _mm_add_ps(acc, v3);
}
```



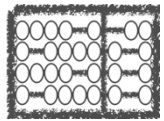
Vetorização de Código

Vetorização: geração de código automática pelo compilador!

- A maioria dos compiladores modernos já faz automaticamente.

- gcc4.7 -O3 -ftree-vectorizer-verbose=1 test.c –c

Dica: Remova expressões condicionais de dentro do laço!

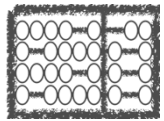


Vetorização de Código

Vetorização: geração de código automática pelo compilador!

Exemplo:

```
void soma_vetor(float* a, float* end_a, float* b, float* end_b)
{
  while( (a < end_a) && (b < end_b) )
    *(a++) += *b++;
}
```

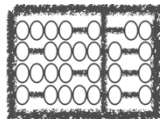


Vetorização de Código

```
gcc -O3 -ftree-vectorize -ftree-vectorizer-verbose=5 -Wall
soma_vetor_v1.c -c

Analyzing loop at soma_vetor_v1.c:2
soma_vetor_v1.c:1: note: vectorized 0 loops in function.
```

Exemplo:

```
void soma_vetor(float* a, float* end_a, float* b, float* end_b)
{
  while( (a < end_a) && (b < end_b) )
    *(a++) += *b++;
}
```

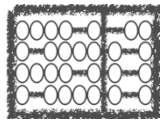


Vetorização de Código

Vetorização: geração de código automática pelo compilador!

```
while( (a < end_a) && (b < end_b) ) {
    *(a++) += *b++;
}
```

```
int n = MIN(end_a-a, end_b-b);
int i;
for (i=0; i<n; i++) {
    *(a++) += *b++;
}
```

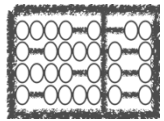


Vetorização de Código

Vetorização: geração de código automática pelo compilador!

```
gcc -O3 -ftree-vectorize -ftree-vectorizer-verbose=1 -
Wall soma_vetor_v2.c -c
Analyzing loop at soma_vetor_v2.c:6
Vectorizing loop at soma_vetor_v2.c:6
6: created 1 versioning for alias checks.
6: LOOP VECTORIZED.
soma_vetor_v2.c:3: note: vectorized 1 loops in function.
```

```
for (i=0; i<n; i++) {
    *(a++) += *b++;
}
```



Aceleração de Aplicações

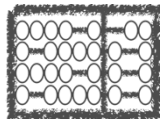
Otimizações Simples

Otimizações na Compilação

Vetorização de Código

Bibliotecas Otimizadas

Otimizações de Acesso a Dados



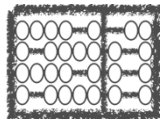
Bibliotecas Otimizadas

Em muitos casos, é muito difícil gerar código ótimo (ou bom) para processadores modernos!

- Compilador não consegue inferir informações!
- Ausência de informação da microarquitetura!
- etc...

Gerar código em ling. de montagem é um desafio!

Solução: usar bibliotecas otimizadas!



Bibliotecas Otimizadas

Exemplo: BLAS – *Basic Linear Algebra Subprograms*

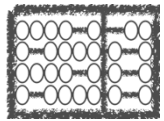
operações entre vetores e matrizes!

BLAS nível 1: operações vetor/vetor

BLAS nível 2: operações vetor/matriz

BLAS nível 3: operações matriz/matriz

- Exemplo: DGEMM (DP - General Matrix Multiply)



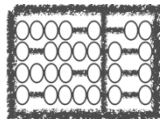
Bibliotecas Otimizadas

Exemplo: BLAS – *Basic Linear Algebra Subprograms*

Muito utilizada em computação de alto desempenho

Geralmente fornecida pelo fabricante do *Hardware*

- *Intel MKL, AMD ACML, cuBLAS ...*



Aceleração de Aplicações

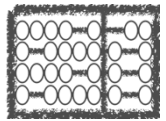
Otimizações Simples

Otimizações na Compilação

Vetorização de Código

Bibliotecas Otimizadas

Otimizações de Acesso a Dados



Otimizações de Acesso a Dados

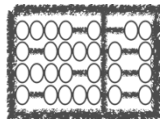
Largura de banda da memória do computador (Banda):

Memória: 2 x 4 GB (1333 MHz DDR3 SDRAM)

Banda = 1333 MT/s x 64 bits x 2 (canais)

= 10666 MB/s x 2

= 21.3 GB/s



Otimizações de Acesso a Dados

Largura de banda da memória d

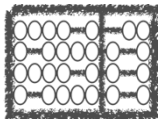
Memória: 2 x 4 GB (1333 MHz D

Banda = 1333 MT/s x 64 bits x nais)

= 10666 MB/s x 2

= 21.3 GB/s

Será que é o suficiente para manter o processador ocupado?



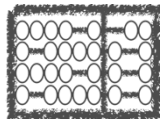
Otimizações de Acesso a Dados

Flops: # de operações de ponto-flutuante por segundo

Dados do computador: (i7-2760QM)

- 4 Cores
- 2.4 GHz
- ALUs: 8 operações FP (DP) por ciclo

  4 somas + 4 multiplicações
- Total = 4 x 2.4 x 8 = 76.8 GFlops

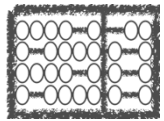


Otimizações de Acesso a Dados

ALUs: 19.4 GFlops x 4 cores = 76.8 GFlops

Memória: 21.3 GB/s

DP FP = 8 *bytes*, logo Memória = 2.66 G FP / s



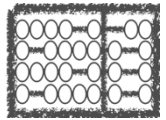
Otimizações de Acesso a Dados

ALUs: 19.4 GFlops x 4 cores = 76.8 GFlops

Memória: 21.3 GB/s

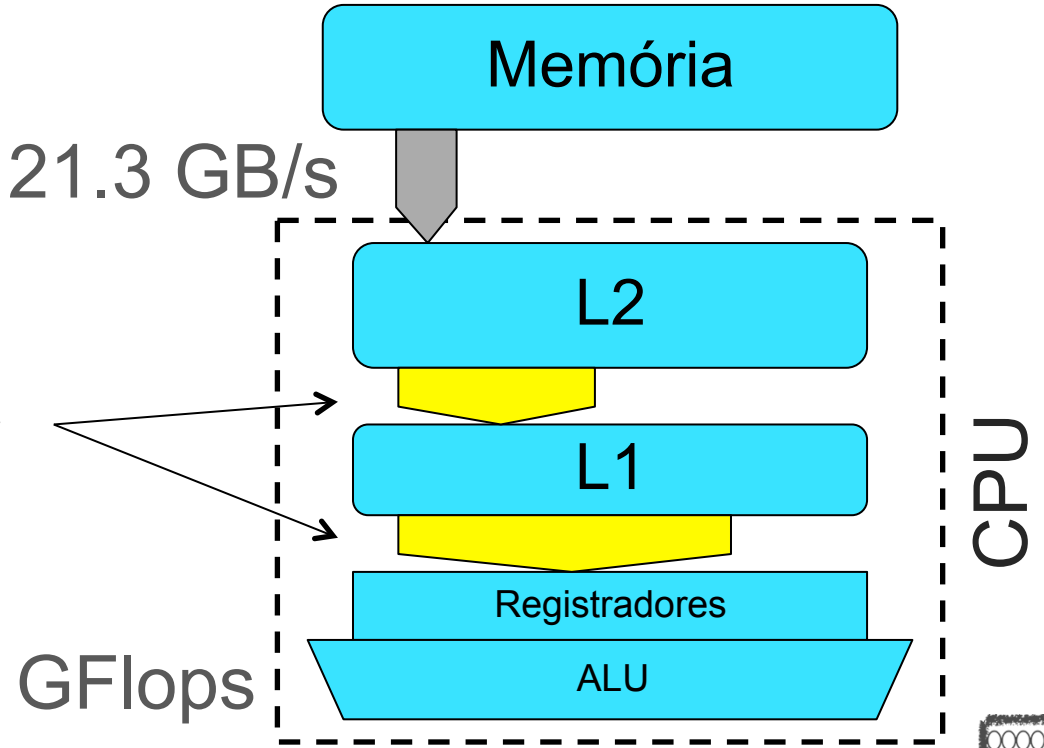
DP FP = 8 *bytes*, logo Memória = 2.66 G FP / s

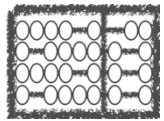
< 3.5% do # de operações nas ALUs



Otimizações de Acesso a Dados

A largura de banda das *caches* é maior!

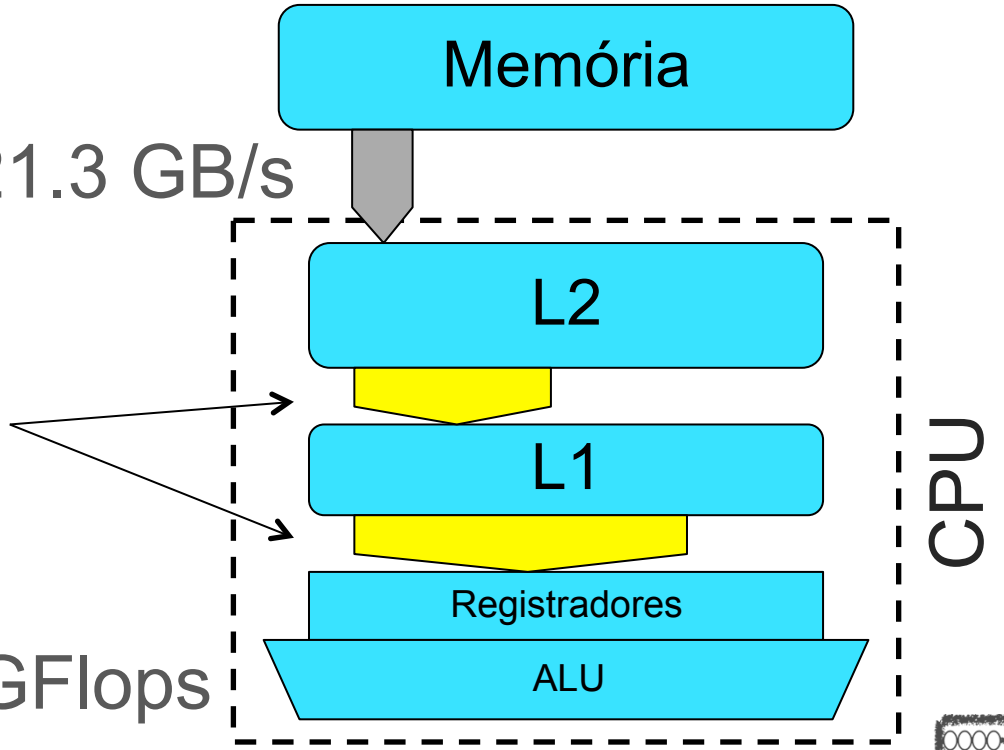




Otimizações de Acesso a Dados

Maximizar o uso das caches!

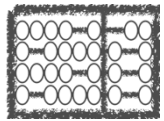
banda das *caches* é maior!



Otimizações de Acesso a Dados

Exemplo: produto interno

```
sum = 0.0;

for (i=0; i<N; i++)

        sum += A[i] x B[i];
```

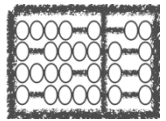


Otimizações de Acesso a Dados

Exemplo: produto interno

```
sum = 0.0;

for (i=0; i<N; i++)

        sum += A[i] x B[i];
```

Não há reuso de dados.



Otimizações de Acesso a Dados

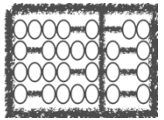
Exemplo: produto interno

```
sum = 0.0;

for (i=0; i<N; i++)

        sum += A[i] x B[i];
```

Não há reuso de dados.

Memória: Medido **~13 GB/s** vs pico = 21.3 GB/s

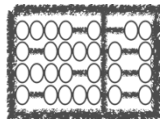
CPU: Medido = **~1.6 GFlop/**s vs pico = 19.4 GFlop/s



Otimizações de Acesso a Dados

Exemplo: transposição de matrizes

```
for (i=0; i<N; i++)
        for (j=0; j<N; j++)
                A[i][j] = B[j][i]
```

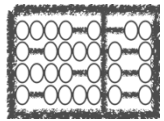


Otimizações de Acesso a Dados

Exemplo: transposição de matrizes

```
for (i=0; i<N; i++)
        for (j=0; j<N; j++)
                A[i][j] = B[j][i]
```

Memória: Medido **~0.93 GB/s** vs pico = 21.3 GB/s



Otimizações de Acesso a Dados

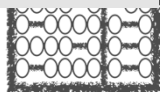
Exemplo: transposição de matrizes

Solução: Blocagem de laços!!!

Pico = 21.3 GB/s
Sem blocagem: **~0.93 GB/s**
Com blocagem: **~5.26 GB/s**

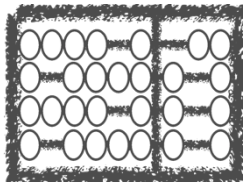
```
for(i=0;i<N;i++)
 for(j=0;j<N;j++)
  A[i][j]=B[j][i]
```

```
for(jk=0; jk<N; jk+=BLK)
 for(i=0; i<N; i++)
  for(j=jk; j<(jk+BLK); j++)
   A[i][j] = B[j][i]
```



Análise de Aplicações e Otimização de Desempenho

Prof. Edson Borin
edson@ic.unicamp.br



Institute of Computing

University of Campinas

Mais informações

Curso de aceleração de aplicações científicas

http://www.ic.unicamp.br/~edson/disciplinas/mo802/2016-1s/index.html