TAREFA 3 – AVALIAÇÃO DE DESEMPENHO

Gabriela Almeida

É possível tornar um programa mais eficiente usando opções do compilador para gerar instruções SIMD. O objetivo dessa tarefa é quantificar essa diferença de desempenho usando funções que consistem em manipulações matemáticas em vetores de tamanhos variados.

Essas diferentes funções estão presentes no arquivo **funcs.cpp**, o qual ao ser compilado e executado gera 21 vetores com conteúdos e tamanhos variados (de 100 a 100 milhões de elementos) e calcula o tempo que demorar para executar cada função em todos os elementos desses vetores. Por fim é gerado um arquivo **t3Tempos.txt** que contem um vetor chamado “tamanhos” o qual possui os tamanhos dos vetores usados para fazer os cálculos em cada iteração. Além disso, contem vetores para cada tipo de função, o qual possui o tempo gasto para fazer aquele cálculo naquela iteração.

In [21]:

```
from IPython.display import Image
Image("Figura1.png")
```

Out[21]:

t3TemposcomSIMDAVX..txt

```
tamanhos=[100, 200, 400, 800, 1600, 3200, 6400, 12800, 25600, 51200, 102400, 204800, 40960
tempo_inner_prod=[1.454e-06, 6.3e-07, 1.175e-06, 2.325e-06, 4.563e-06, 9.066e-06, 1.8048e-
tempo_sum_positive=[5.19e-07, 8.3e-07, 1.528e-06, 2.878e-06, 5.807e-06, 1.0799e-05, 2.2065
tempo_sqrt_element=[5.28e-07, 7.73e-07, 1.456e-06, 2.89e-06, 5.721e-06, 1.1356e-05, 2.2765
tempo_exp_element=[2.409e-06, 3.461e-06, 6.912e-06, 1.379e-05, 2.7594e-05, 5.6127e-05, 0.0
tempo_log_element=[1.7036e-05, 5.502e-06, 1.1418e-05, 2.2322e-05, 4.4698e-05, 8.9946e-05,
tempo_gauss=[6.017e-06, 4.167e-06, 8.19e-06, 1.6469e-05, 3.2822e-05, 6.6319e-05, 0.0001618
```

Analisando a imagem a cima é possível deduzir que na primeira iteração foi gerado um vetor de 100 elementos e que para calcular, por exemplo, a raiz de cada um deles (tempo_sqrt_element) demorou 5.19 e-07 segundos. Já na segunda iteração foi gerado um vetor com 200 elementos e para calcular o exponencial de cada um deles (tempo_exp_element) demorou 3.1461 e-06 segundos.

Para poder quantificar a diferença de desempenho é preciso compilar o arquivo **funcs.cpp** de diferentes formas. Primeiramente ele é compilado sem habilitar a auto vetorização:

```
g++ -o funcsSemSIMD funcs.cpp
```

e então esse programa é executado gerando o arquivo **t3Tempos.txt** o qual é aconselhável renomear pois quando compilar o arquivo **funcs.cpp** novamente, porém habilitando a auto vetorização, esse arquivo será sobrescrito. Portanto renomeia-se o arquivo para **t3TemposSemSIMD.txt**.

Em um segundo momento compila-se o programa habilitando a auto vetorização e utilizando uma arquitetura AVX:

```
g++ -ffast-math -ftree-vectorize -mavx -o funcscomSIMDAVX funcs.cpp
```

Após a execução do programa renomeia-se o arquivo de saída gerado para **t3TemposcomSIMDAVX.txt**. Por fim o programa é compilado habilitando a auto vetorização utilizando uma arquitetura AVX2:

```
g++ -ffast-math -ftree-vectorize -mavx2 -o funcscomSIMDAVX2 funcs.cpp
```

e após a execução do programa é aconselhável renomear o arquivo de saída para **t3TemposcomSIMDAVX2.txt.**

Esses arquivos de saída foram importados nesse notebook e para poder diferenciar os vetores de cada um deles seus nomes foram alterados. Nos vetores do arquivo **t3TemposSemSIMD.txt** foi adicionado "_sem_SIMD" ao final:

In [22]:

```
tamanhos=[100, 200, 400, 800, 1600, 3200, 6400, 12800, 25600, 51200, 102400, 204800,
tempo_inner_prod_sem_SIMD=[1.437e-06, 8.14e-07, 1.486e-06, 2.925e-06, 5.806e-06, 1.1
tempo_sum_positive_sem_SIMD=[5.92e-07, 1.044e-06, 1.893e-06, 3.988e-06, 7.539e-06, 1
tempo_sqrt_element_sem_SIMD=[7.42e-07, 1.336e-06, 2.637e-06, 5.25e-06, 1.0366e-05, 2
tempo_exp_element_sem_SIMD=[3.407e-06, 5.189e-06, 1.0481e-05, 2.0601e-05, 4.1071e-05
tempo_log_element_sem_SIMD=[2.7263e-05, 7.402e-06, 1.4605e-05, 2.918e-05, 5.8114e-05
tempo_gauss_sem_SIMD=[7.676e-06, 7.019e-06, 1.4013e-05, 2.7847e-05, 5.5811e-05, 0.00
```

Nos vetores do arquivo **t3TemposcomSIMDAVX.txt** foi adicionado "_com_SIMDAVX" ao final:

In [23]:

```
tempo_inner_prod_com_SIMDAVX=[1.454e-06, 6.3e-07, 1.175e-06, 2.325e-06, 4.563e-06, 9
tempo_sum_positive_com_SIMDAVX=[5.19e-07, 8.3e-07, 1.528e-06, 2.878e-06, 5.807e-06,
tempo_sqrt_element_com_SIMDAVX=[5.28e-07, 7.73e-07, 1.456e-06, 2.89e-06, 5.721e-06,
tempo_exp_element_com_SIMDAVX=[2.409e-06, 3.461e-06, 6.912e-06, 1.379e-05, 2.7594e-0
tempo_log_element_com_SIMDAVX=[1.7036e-05, 5.502e-06, 1.1418e-05, 2.2322e-05, 4.4698
tempo_gauss_com_SIMDAVX=[6.017e-06, 4.167e-06, 8.19e-06, 1.6469e-05, 3.2822e-05, 6.6
```

E nos vetores do arquivo **t3TemposcomSIMDAVX2.txt** foi adicionado "_com_SIMDAVX2" ao final

In [24]:

```
tempo_inner_prod_com_SIMDAVX2=[1.25e-06, 7.27e-07, 1.759e-06, 3.045e-06, 4.619e-06,
tempo_sum_positive_com_SIMDAVX2=[5.06e-07, 1.025e-06, 2.483e-06, 3.874e-06, 1.9684e-
tempo_sqrt_element_com_SIMDAVX2=[5.38e-07, 8.74e-07, 2.754e-06, 4.659e-06, 6.663e-06
tempo_exp_element_com_SIMDAVX2=[2.53e-06, 5.821e-06, 1.0161e-05, 1.3879e-05, 3.1756e
tempo_log_element_com_SIMDAVX2=[2.0165e-05, 8.407e-06, 1.4657e-05, 2.8779e-05, 5.194
tempo_gauss_com_SIMDAVX2=[4.398e-06, 6.135e-06, 1.1255e-05, 1.6783e-05, 3.7993e-05,
```

Foi entao criada a função "plotar" abaixo para poder analisar visualmente a melhora do desempenho quando

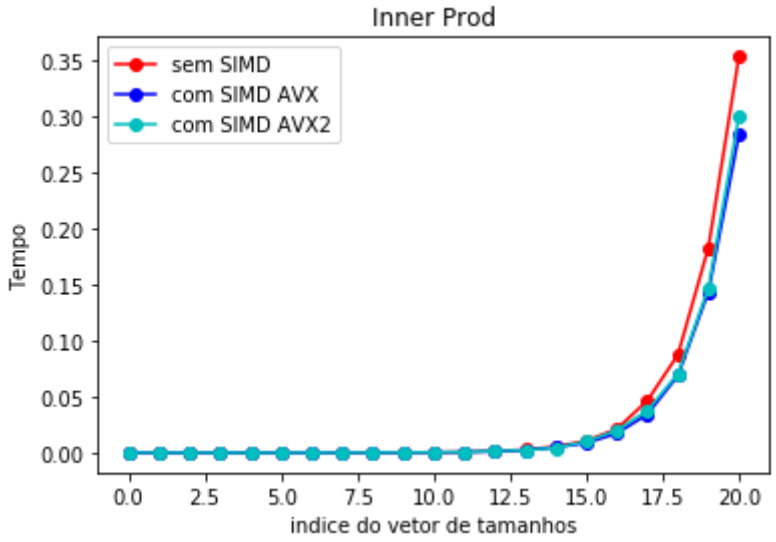
se compila habilitando a auto vetorização. Para facilitar a análise, ao invés do eixo x obter os tamanhos efetivos dos vetores usados na análise, o que atrapalharia devido a grande diferença entre o menor e o maior tamanho de vetor, usou-se o índice desses valores no vetor "Tamanhos":

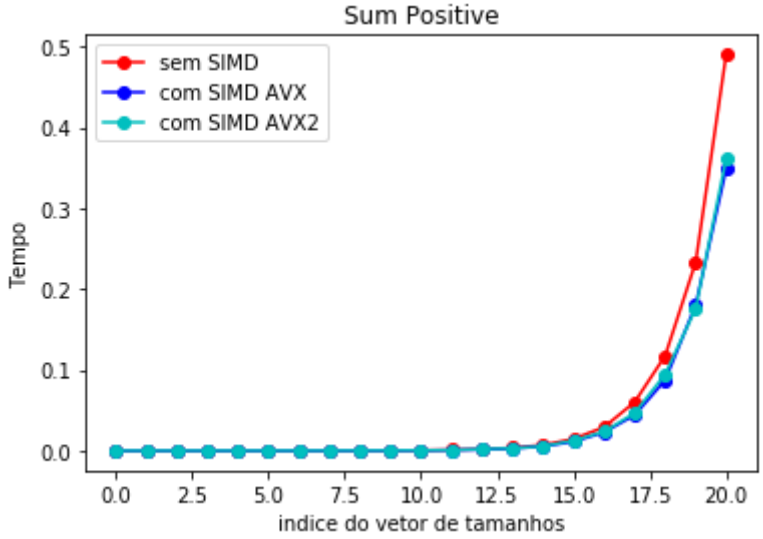
In [25]:

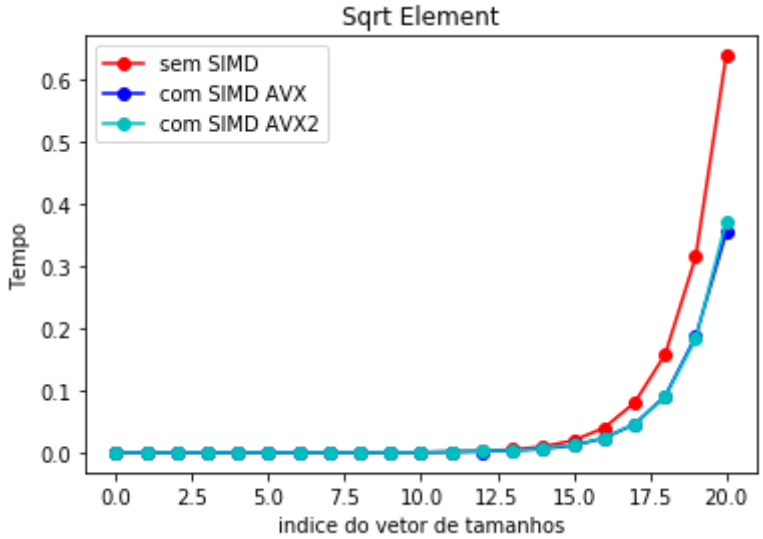
```
import matplotlib.pyplot as plt
t = []
for i in range (0, len(tamanhos)):
    t.append(i)
def plotar(vetor_sem_SIMD, vetor_com_com_SIMDAVX, vetor_com_SIMDAVX2, titulo):
    sem_SIMD, = plt.plot(t, vetor_sem_SIMD, 'ro-', label='sem SIMD')
    com_SIMDAVX, = plt.plot(t, vetor_com_com_SIMDAVX, 'bo-', label='com SIMD AVX')
    com_SIMDAVX2, = plt.plot(t, vetor_com_SIMDAVX2, 'co-', label= 'com SIMD AVX2')
    plt.xlabel('indice do vetor de tamanhos')
    plt.ylabel('Tempo')
    plt.title(titulo)
    plt.legend(handles=[sem_SIMD, com_SIMDAVX,com_SIMDAVX2])
    plt.show()
```

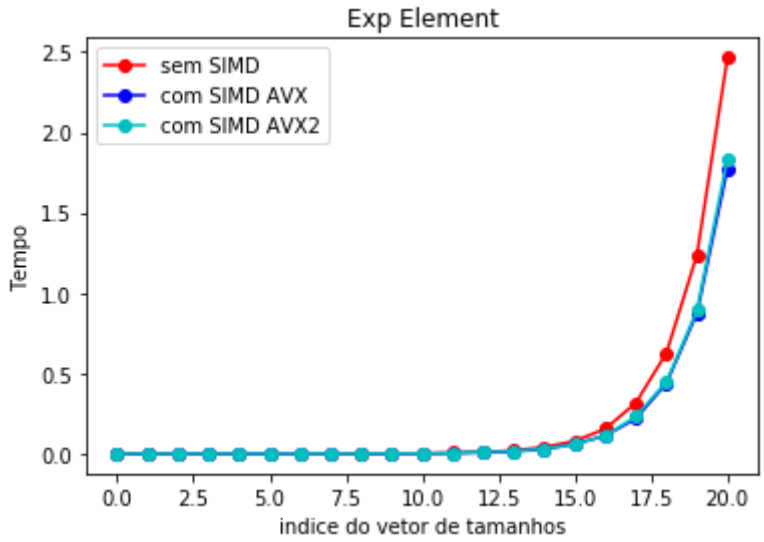
In [26]:

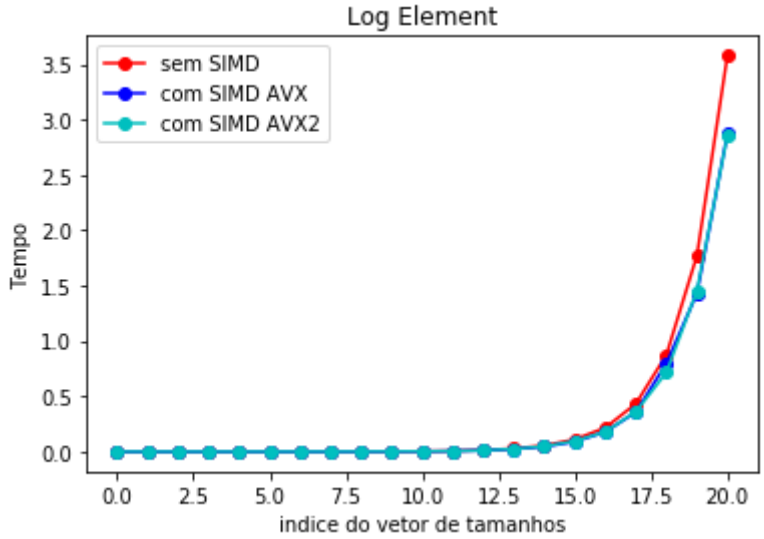
```
plotar(tempo_inner_prod_sem_SIMD,tempo_inner_prod_com_SIMDAVX,tempo_inner_prod_com_S
plotar(tempo_sum_positive_sem_SIMD,tempo_sum_positive_com_SIMDAVX,tempo_sum_positive
plotar(tempo_sqrt_element_sem_SIMD,tempo_sqrt_element_com_SIMDAVX,tempo_sqrt_element
plotar(tempo_exp_element_sem_SIMD,tempo_exp_element_com_SIMDAVX,tempo_exp_element_co
plotar(tempo_log_element_sem_SIMD,tempo_log_element_com_SIMDAVX,tempo_log_element_co
plotar(tempo_gauss_sem_SIMD,tempo_gauss_com_SIMDAVX,tempo_gauss_com_SIMDAVX2, 'Gauss
```

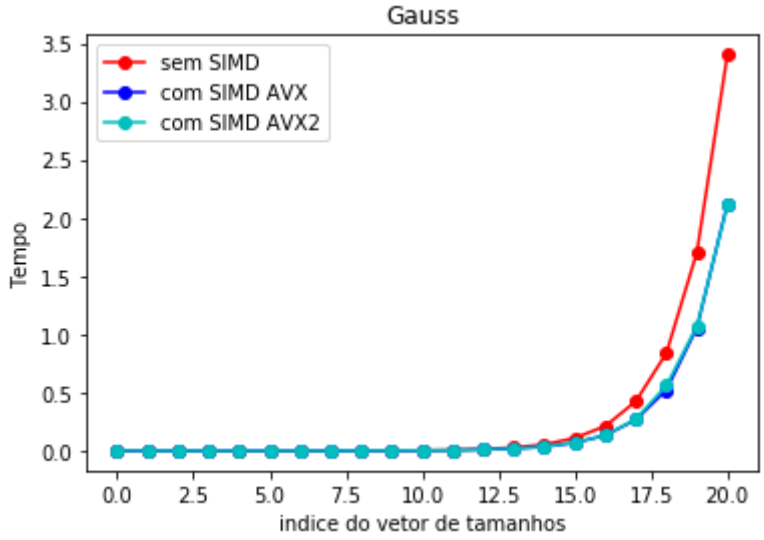












Analisando os gráficos a cima é possível perceber que só existem ganhos de desempenhos expressivos a partir do 15o tamanho de array representado no vetor "Tamanhos":

In [27]:

```
tamanhos[15]
```

Out[27]:

```
3276800
```

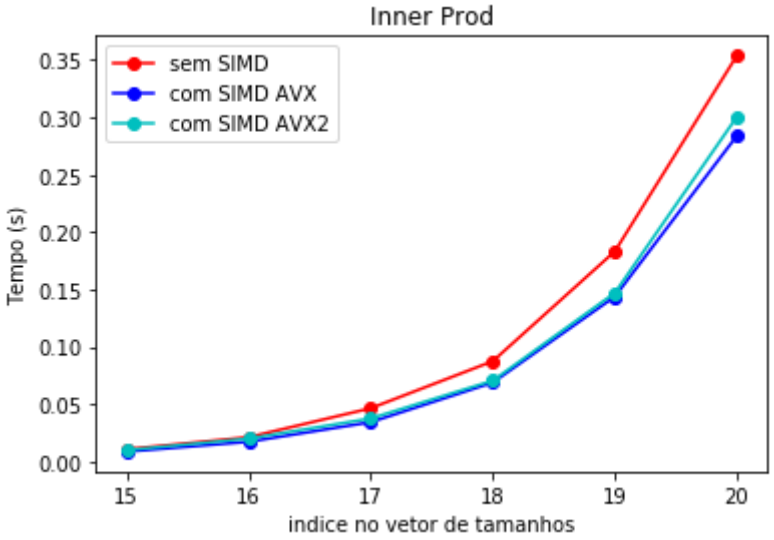
Portanto a partir do tamanho de array 3 milhões o código vetorizado mostra ganhos de desempenho expressivos. Por isso foi criada outra função (plota_apartir15) a qual plota gráficos os quais é possível analisar melhor a diferença de desempenho apenas dos vetores cujo tamanho é maior que 3 milhões:

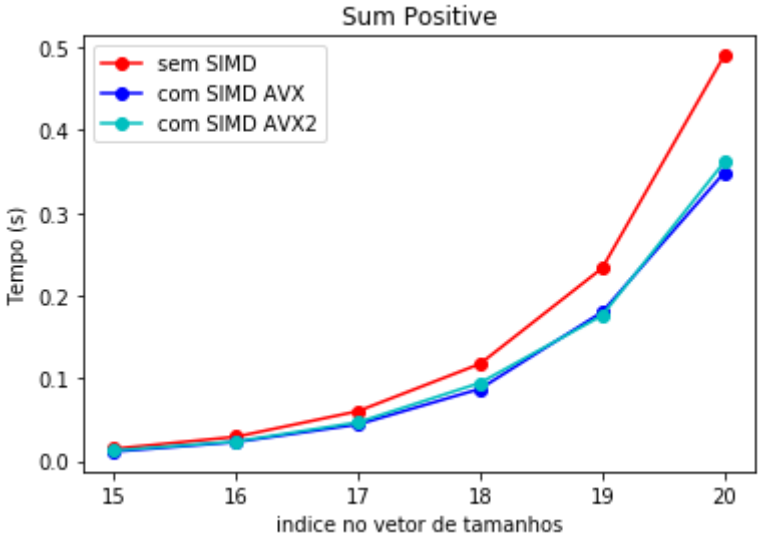
In [28]:

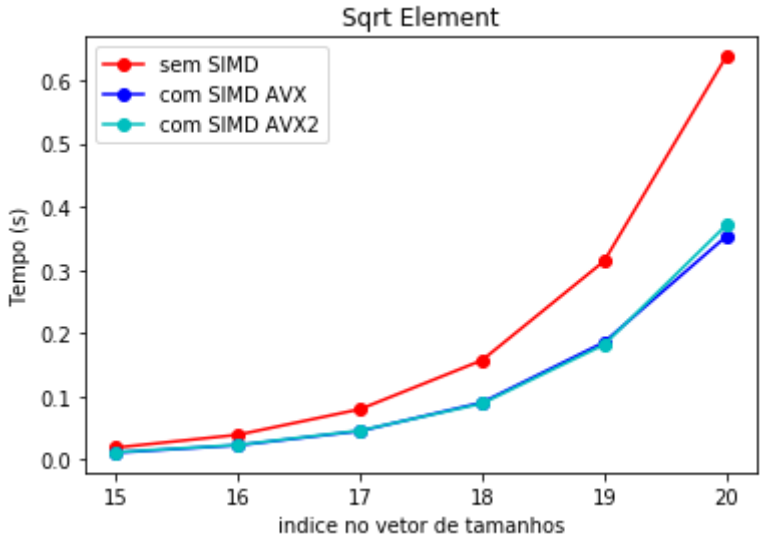
```
for i in range (0, len(tempo_inner_prod_sem_SIMD)):
    t[i] = i
def plotar_apartir15(vetor_sem_SIMD, vetor_com_com_SIMDAVX, vetor_com_SIMDAVX2, titu
    sem_SIMD, = plt.plot(t[15:len(tamanhos)], vetor_sem_SIMD[15:len(t)], 'ro-', labe
    com_SIMDAVX, = plt.plot(t[15:len(tamanhos)], vetor_com_com_SIMDAVX[15:len(t)],
    com_SIMDAVX2, = plt.plot(t[15:len(tamanhos)], vetor_com_SIMDAVX2[15:len(t)], 'co
    plt.xlabel('indice no vetor de tamanhos')
    plt.ylabel('Tempo (s)')
    plt.title(titulo)
    plt.legend(handles=[sem_SIMD, com_SIMDAVX,com_SIMDAVX2])
    plt.show()
```

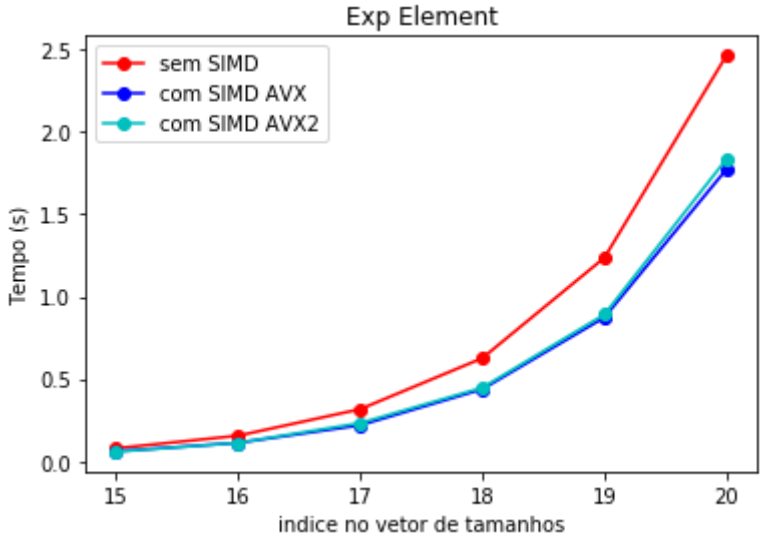
In [29]:

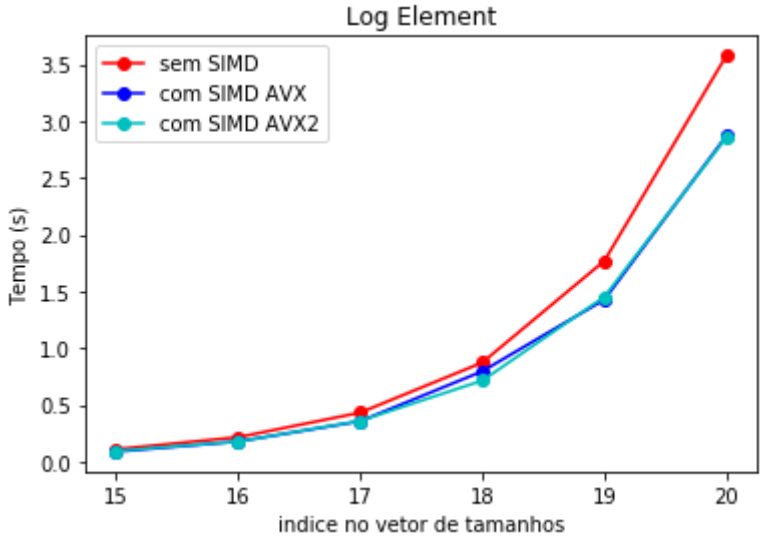
```
plotar_apartir15(tempo_inner_prod_sem_SIMD,tempo_inner_prod_com_SIMDAVX,tempo_inner_
plotar_apartir15(tempo_sum_positive_sem_SIMD,tempo_sum_positive_com_SIMDAVX,tempo_su
plotar_apartir15(tempo_sqrt_element_sem_SIMD,tempo_sqrt_element_com_SIMDAVX,tempo_sq
plotar_apartir15(tempo_exp_element_sem_SIMD,tempo_exp_element_com_SIMDAVX,tempo_exp_
plotar_apartir15(tempo_log_element_sem_SIMD,tempo_log_element_com_SIMDAVX,tempo_log_
plotar_apartir15(tempo_gauss_sem_SIMD,tempo_gauss_com_SIMDAVX,tempo_gauss_com_SIMDAV
```

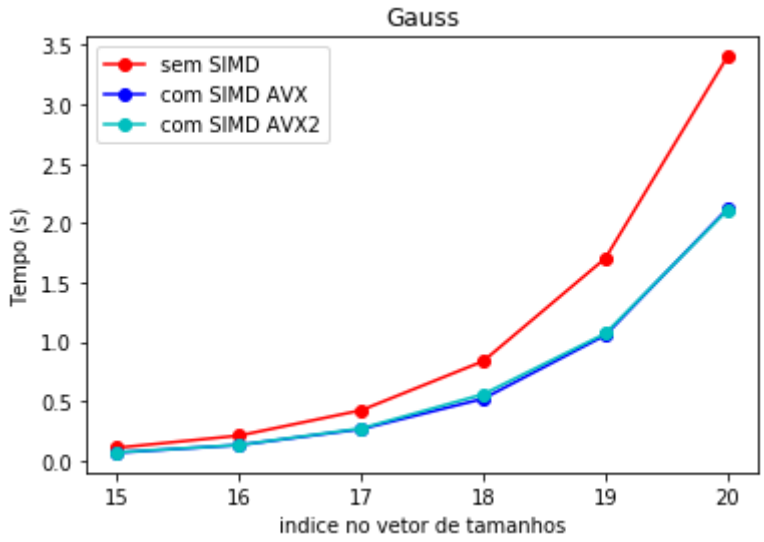





Sqrt Element
sem SIMD
com SIMD AVX
com SIMD AVX2
Tempo (s)
0.0
0.1
0.2
0.3
0.4
0.5
0.6
15
16
17
18
19
20
indice no vetor de tamanhos


Exp Element
sem SIMD
com SIMD AVX
com SIMD AVX2
Tempo (s)
0.0
0.5
1.0
1.5
2.0
2.5
15
16
17
18
19
20
indice no vetor de tamanhos


Log Element
sem SIMD
com SIMD AVX
com SIMD AVX2
Tempo (s)
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
15
16
17
18
19
20
indice no vetor de tamanhos



Como é possível perceber analisando os gráficos a cima, em todos os tipos de função, em todos os tamanhos de vetores, as compilações que a auto vetorização estava habilitada tiveram um desempenho melhor do que quando não estava desabilitada. Além disso é possível perceber que o desempenho da arquitetura AVX é, no geral, melhor que a da arquitetura AVX2.
Um ponto importante a se destacar é que a arquitetura AVX usa registradores de 128 bits, ou seja, por registrador é possível armazenar 2 variáveis double (que é o tipo de variável usada nas funções). Portanto com a vetorização o ganho de desempenho deveria ser por volta de 2 vezes.

Para provar a afirmação acima, para descobrir os ganhos de desempenho e quais tipos de operações resultam em maior ganho de desempenho fez-se as médias dos ganhos de cada operação para cada tipo de arquitetura de auto vetorização:

In [30]:

```
inner_prod_media1=0
inner_prod_media2=0

sum_positive_media1=0
sum_positive_media2=0

sqrt_element_media1=0
sqrt_element_media2=0

exp_element_media1=0
exp_element_media2=0

log_element_media1=0
log_element_media2=0

gauss_media1=0
gauss_media2=0

cont = 0
for i in range (14 , len(tamanhos)):
    inner_prod_media1 = inner_prod_media1 + (tempo_inner_prod_sem_SIMD[i]/tempo_inne
    inner_prod_media2 = inner_prod_media2 + (tempo_inner_prod_sem_SIMD[i]/tempo_inne

    sum_positive_media1 = sum_positive_media1 + (tempo_sum_positive_sem_SIMD[i]/temp
    sum_positive_media2 = sum_positive_media2 + (tempo_sum_positive_sem_SIMD[i]/temp

    sqrt_element_media1 = sqrt_element_media1 + (tempo_sqrt_element_sem_SIMD[i]/temp
    sqrt_element_media2 = sqrt_element_media2 + (tempo_sqrt_element_sem_SIMD[i]/temp

    exp_element_media1 = exp_element_media1 + (tempo_exp_element_sem_SIMD[i]/tempo_e
    exp_element_media2 = exp_element_media2 + (tempo_exp_element_sem_SIMD[i]/tempo_e

    log_element_media1 = log_element_media1 + (tempo_log_element_sem_SIMD[i]/tempo_l
    log_element_media2 = log_element_media2 + (tempo_log_element_sem_SIMD[i]/tempo_l

    gauss_media1 = gauss_media1 + (tempo_gauss_sem_SIMD[i]/tempo_gauss_com_SIMDAVX[i
    gauss_media2 = gauss_media2 + (tempo_gauss_sem_SIMD[i]/tempo_gauss_com_SIMDAVX2[

    cont = cont +1

print("Média de ganho AVX para inner_prod= " + str(inner_prod_media1/cont))
print("Média de ganho AVX2 para inner_prod= " + str(inner_prod_media2/cont)+'\n')

print("Média de ganho AVX para sum_positive= " + str(sum_positive_media1/cont))
print("Média de ganho AVX2 para sum_positive= " + str(sum_positive_media2/cont)+'\n

print("Média de ganho AVX para sqrt_element= " + str(sqrt_element_media1/cont))
print("Média de ganho AVX2 para sqrt_element= " + str(sqrt_element_media2/cont)+'\n

print("Média de ganho AVX para exp_element= " + str(exp_element_media1/cont))
print("Média de ganho AVX2 para exp_element= " + str(exp_element_media2/cont)+'\n')

print("Média de ganho AVX para log_element= " + str(log_element_media1/cont))
print("Média de ganho AVX2 para log_element= " + str(log_element_media2/cont)+'\n')

print("Média de ganho AVX gauss= " + str(gauss_media1/cont))
print("Média de ganho AVX2 gauss= " + str(gauss_media2/cont))
```

```
Média de ganho AVX para inner_prod= 1.2495002775526947
Média de ganho AVX2 para inner_prod= 1.1824885101186011

Média de ganho AVX para sum_positive= 1.3316190822700975
Média de ganho AVX2 para sum_positive= 1.2729372167273636

Média de ganho AVX para sqrt_element= 1.729382291685068
Média de ganho AVX2 para sqrt_element= 1.656855290548903

Média de ganho AVX para exp_element= 1.4071008664282052
Média de ganho AVX2 para exp_element= 1.3710847159997448

Média de ganho AVX para log_element= 1.2095756245519615
Média de ganho AVX2 para log_element= 1.2027758751315136

Média de ganho AVX gauss= 1.602992514072008
Média de ganho AVX2 gauss= 1.557619576014062
```

Analisando os resultados é possível comprovar que a arquitetura AVX2 possui um melhor desempenho quando comparada com a arquitetura AVX. No entanto, o ganho, no geral, é um pouco distante do esperado (que era de 2 vezes). Além disso é possivel perceber que a operação "gauss" e "sqrt_element" são as que resultaram no maior ganho de desempenho.