Leonardo Pereira Medeiros

Caixeiro Viajante (Multi-Core-GPU)

Introdução

Este projeto consiste no estudo de técnicas mult-core na área de otimização discreta, que se aplica a problemas em que há uma sequência de escolhas, com a característica de que a solução ótima só pode ser encontrada ao enumerar todas as escolhas possíveis. Isto significa que todo algoritmo para sua solução é O(2n) ou pior.

Neste projeto serão implementados dois métodos de otimização discreta, com processamento mult-core em GPU estes são: solução aleatória e 2-opt.

Solução Aleatória: Consiste em sortear um número de soluções aleatoriamente e selecionar a que tiver menor custo, esta é uma solução simplória, com resultados não ideais, porém trata-se de uma solução inicial satisfatória.

2-opt (local search) : Consiste em aplicar a heurística 2-opt sobre o método de solução aleatória, aplicando operações em cada iteração para melhorar o parâmetro de critério. Embora este seja um método bastante rápido, seu valor final,em algumas execuções, será apenas próximo do valor ideal, pois esse método é afetado por mínimos locais, logo o processo de inicialização aleatória e melhoria e executado diversas vezes, para reduzir esse efeito colateral.

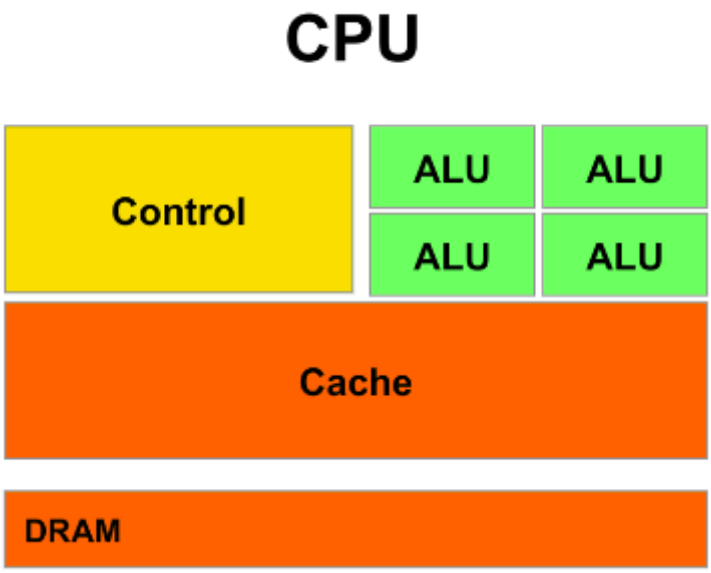
Esse estudo de otimização já foi realizado em outro projeto porém aplicando técnicas multicore em CPU, A fim de tornar a análise de otimização mais rica, os resultados obtidos no projeto em CPU serão adicionados à seção de análises para serem comparados com os resultados gerados pela implementação em GPU. A fim de concretizar possíveis vantagens e desvantagens de cada aplicação. definindo a que possui melhor desempenho sobre tal tema.

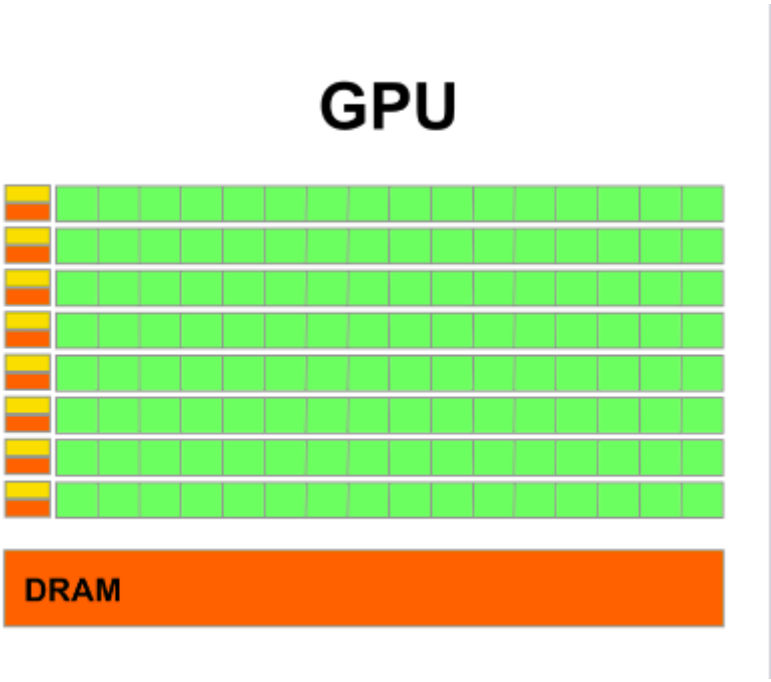
GPU vs CPU

Enquanto a CPU possui um design orientado a minimizar a latência, unidades lógicas aritméticas potentes, caches grandes, controle sofisticado e uma quantidade reduzida de cores. As Gpus possuem um design orientado a minimizar o throughput, com unidades aritméticas simples, caches reduzidos e uma quantidade massiva de cores.

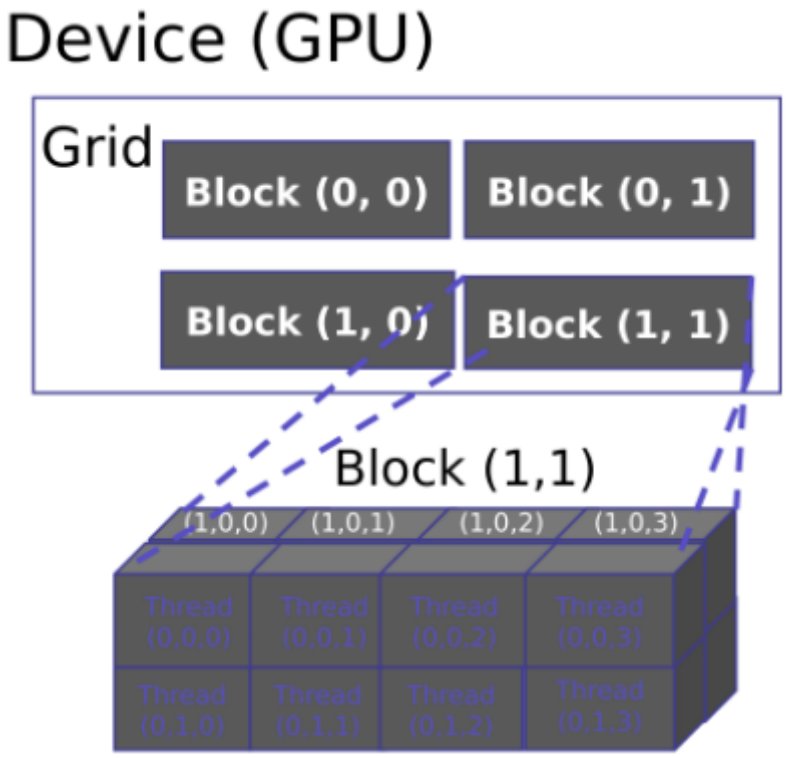
Dada essa diferença as CPUs podem ser até 10x mais rápidas que as GPUs para código sequencial, enquanto a GPUs podem ser 10x mais rápidas que as CPUs para códigos

paralelos.





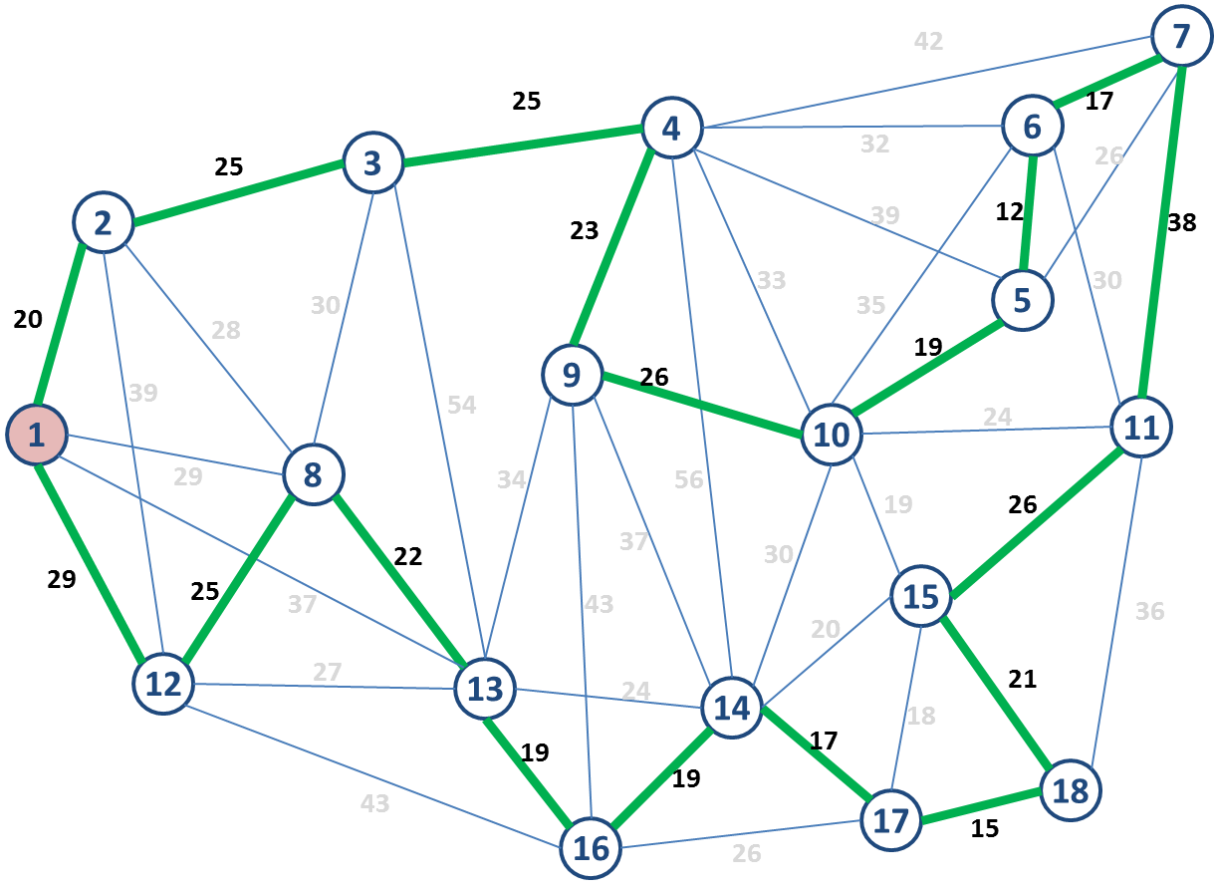
Para utilizar todo o poder de processamento da GPU utiliza-se o CUDA(Compute Unified Device Architecture), que é uma engine de computação desenvolvida pela Nvidia, que trabalha sobre a estrutura de divisão de memória da GPU em que a thread é a menor unidade de tarefa, essas threads são agrupadas em blocos, em que as threads de um mesmo bloco podem se comunicar e se sincronizar. Os blocos por sua vez são organizados em grids, onde deverão ter a mesma dimensão, executando a mesma função de kernel, podendo a grid ser bidimensional ou tridimensional.



O Problema

Para aplicar os conceitos de otimização discreta foi implementado o problema do Caixeiro Viajante, que consiste em determinar a menor rota para percorrer uma série de cidades(sem repetições) e retornar a cidade de origem.

Este problema foi inspirado na necessidade dos vendedores em realizar entregas em diversos locais, percorrendo o menor caminho possível, e consequentemente reduzindo o tempo necessário para a viagem e os possíveis custos com transporte e combustível.



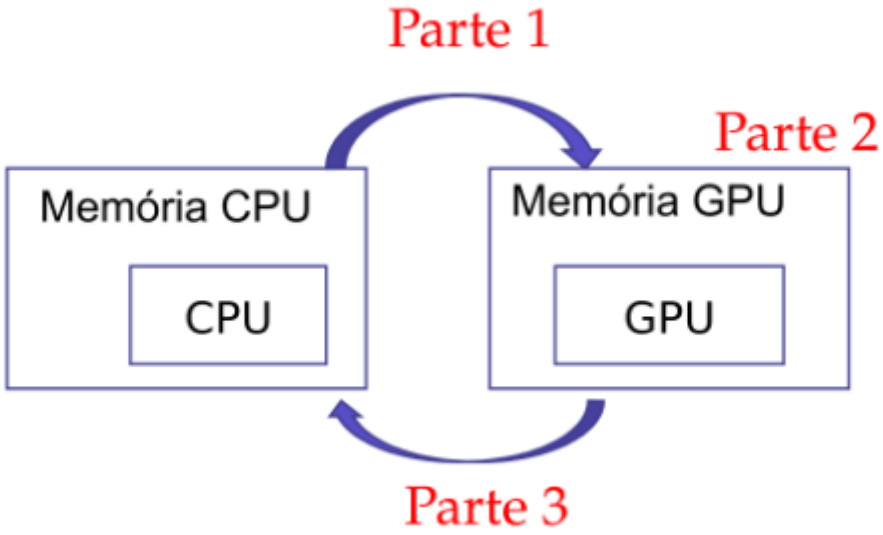
Estrutura Do Projeto

O projeto está estruturado de forma que cada método implementado possui seu respectivo arquivo.cpp, esses arquivos possuem uma função main, responsável por definir o fluxo de dados entre CPU e GPU. Para aplicar o método de otimização discreta de forma paralela sobre o conjunto de dados, há dois kernels, definidos pela flag **global**:

- **global** points_distance(1 dimensão): Pré calcula a distâncias entre todos os pontos
- **global** path_dist(2 dimenções): Calcula a distância de um caminho gerado aleatoriamente.

Também há duas funções auxiliares em GPU, definidas pela flag **device** :

- **device** dist: Calcula a distância entre dois pontos
- **device** path dist: Calcula a distância total de uma sequência de pontos.



- Parte1: Copia dados CPU -> GPU
- Parte2: Processa dados na GPU
- Parte3: Copia resultados GPU -> CPU

Testes

Os inputs selecionados para testar o desempenho da aplicação de técnicas de otimização discreta foram formulados de forma a tentar levar ao máximo o processamento da GPU. Como o programa exige mais processamento quando há muitos pontos que o caixeiro deve percorrer, esse valor foi testado de forma crescente, e com grandes intervalos de iterações máximas, a fim de tentar tornar mais visível o ganho de desempenho ao aplicar essa técnica para diferentes demandas de processamento.

Os seguintes inputs foram testados:

- Input1: N = 200
- Input2: N = 300
- Input3: N = 400
- Input4: N = 500

Em que N representa o número de pontos no sistema.

Definido como variável de desempenho o tempo de execução do programa, foi utilizada a estrutura cudaEvent_t para mensurar esta informação, estrutura esta que permite obter este parâmetro no contexto da GPU.

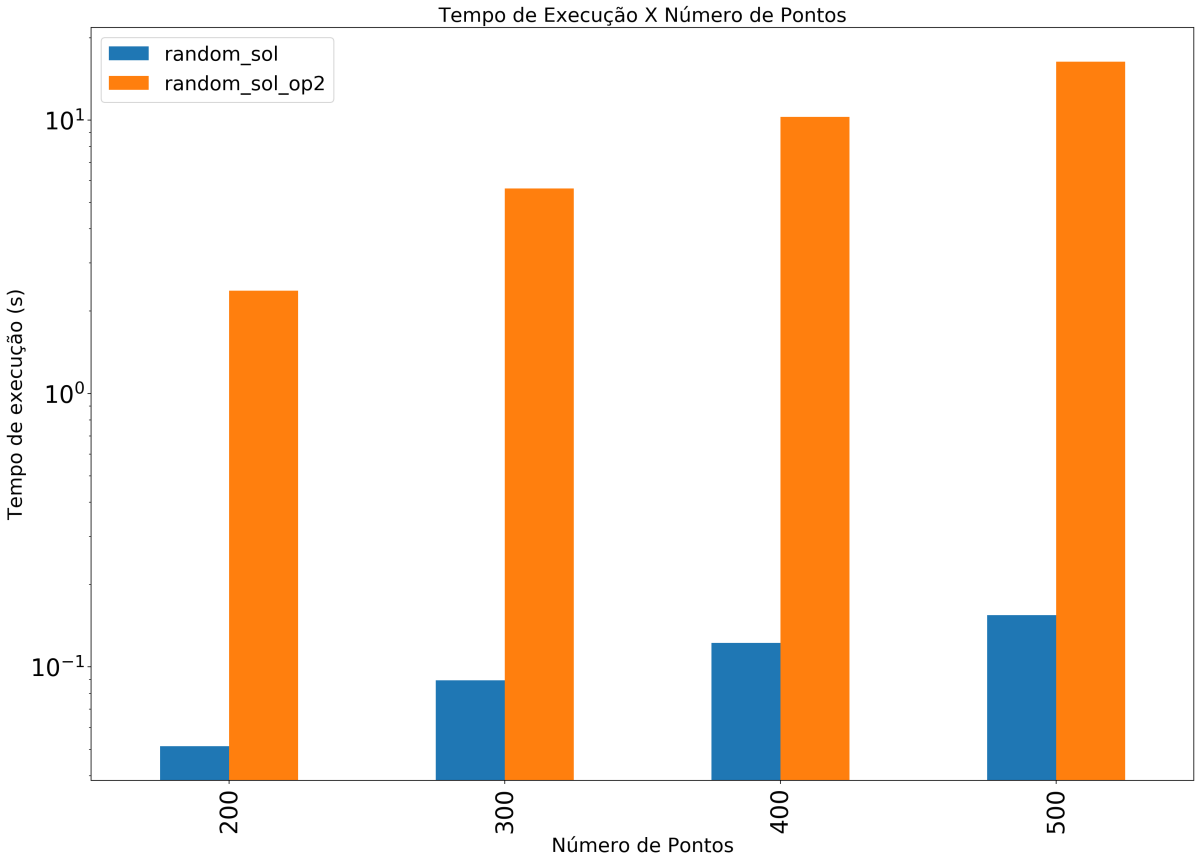
Como deseja-se medir apenas o desempenho do algoritmo do caixeiro viajante a medição não leva em conta o tempo gasto na leitura do arquivo de input e outros processos similares.

Resultados

```
##import dependences
import matplotlib.pyplot as plt
import pandas as pd
import subprocess
import sys
from IPython.display import display

##files
dir_fil = "../build/"
files    = ["random_sol", "random_sol_op2"]

dir_inp = "../inputs/"
inputs   = ["input1", "input2", "input3", "input4"]

n_sizes   = []
dic       = {}

## Run files and storage output
for input in inputs:
    dic[input] = {}

    for f in files:
        command = dir_fil + f + " < " + dir_inp + input
        stout   = subprocess.check_output(command, stderr=subprocess.STDOUT,
        dic[input][f]  = float(stout.split("\n")[-2].split(":")[1])

    with open(dir_inp + input, 'r') as f:
        nrect = f.read().split("\n")[0]

    n_sizes.append(int(nrect))

## Generate Dataframe
s0 = [dic["input1"]["random_sol"], dic["input2"]["random_sol"], dic["input3"]
s1 = [dic["input1"]["random_sol_op2"], dic["input2"]["random_sol_op2"], dic["


df = pd.DataFrame({"random_sol":s0, "random_sol_op2":s1,}, index = n_sizes)

display(df)

df.plot.bar(figsize = (20,14), fontsize=24, logy = True)
plt.legend(prop={'size': 20})
plt.title("Tempo de Execução X Número de Pontos", fontsize = 20)
plt.xlabel("Número de Pontos", fontsize = 20)
plt.ylabel("Tempo de execução (s)", fontsize = 20)
```

	random_sol	random_sol_op2
200	0.051296	2.37384
300	0.089388	5.61267
400	0.122595	10.26540
500	0.154617	16.32280

```
Text(0, 0.5, 'Tempo de execução (s)')
```

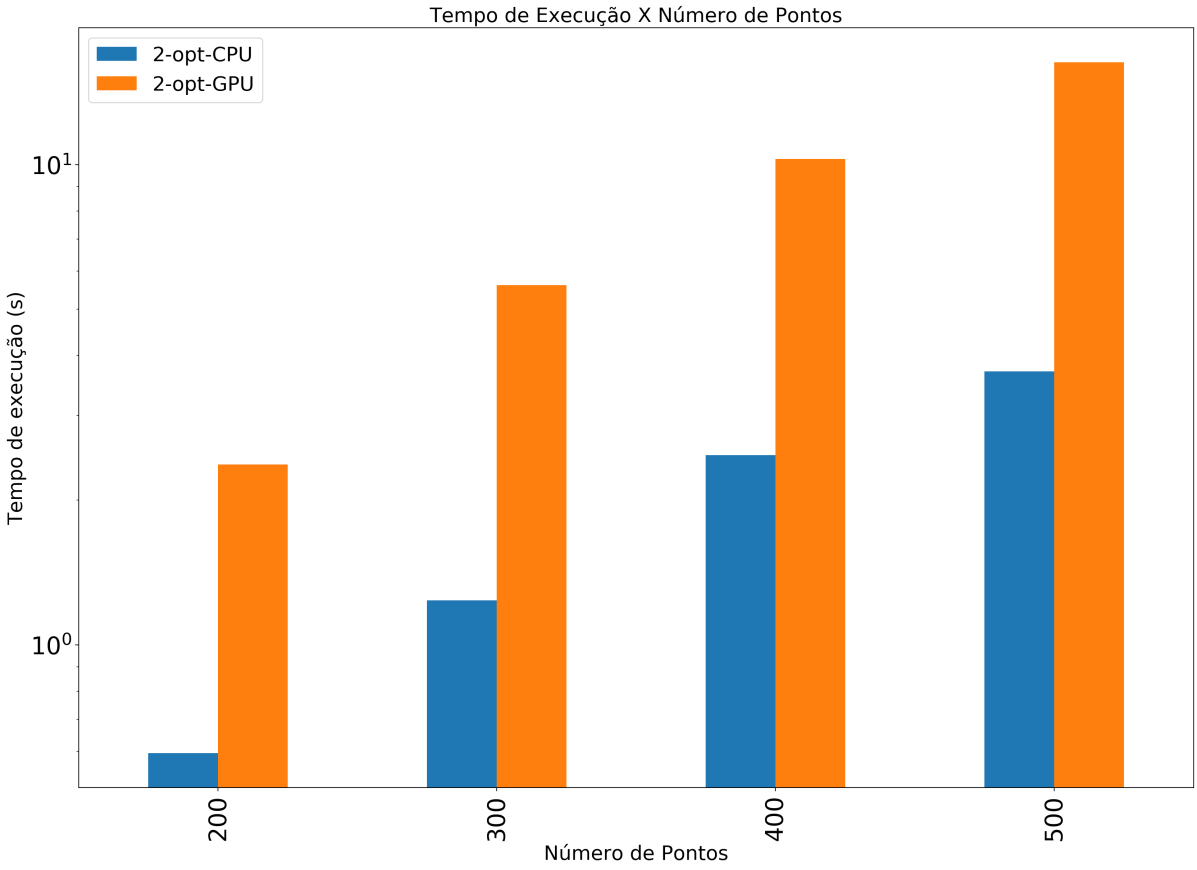


CPU vs GPU

```
s0 = [0.594745, 1.23814, 2.48137, 3.70665] ## Valores obtidos com o projeto e
s1 = [dic["input1"]["random_sol_op2"], dic["input2"]["random_sol_op2"], dic["

df = pd.DataFrame({"2-opt-CPU":s0, "2-opt-GPU":s1,}, index = n_sizes)

display(df)

df.plot.bar(figsize = (20,14), fontsize=24, logy = True)
plt.legend(prop={'size': 20})
plt.title("Tempo de Execução X Número de Pontos", fontsize = 20)
plt.xlabel("Número de Pontos", fontsize = 20)
plt.ylabel("Tempo de execução (s)", fontsize = 20)
```

	2-opt-CPU	2-opt-GPU
200	0.594745	2.37384
300	1.238140	5.61267
400	2.481370	10.26540
500	3.706650	16.32280

```
Text(0, 0.5, 'Tempo de execução (s)')
```



Conclusão

Como pode-se observar nos resultados obtidos ao comparar os métodos de GPU, tem-se que a solução com 2-opt possui um tempo de execução maior, o que já era esperado, pois este método consiste em acrescentar diversas operações ao método de solução aleatória, porém a solução 2-opt obtém valores melhores de custo, aproximando-se do valor ideal.

Ao comparar o método com 2-opt, considerado como um local search, implementado em GPU com o mesmo método implementado em CPU esperava-se que o código em GPU gera-se um desempenho muito melhor.

Porém como o obtido a implementação em CPU possui melhor desempenho, isso pode ter sido causado pela forma como o código está implementado, cada thread da GPU está acessando a memoria global(acesso lento), para corrigir este ocorrido pode-se utilizar a memória compartilhada de cada bloco(acesso rápido), e balancear a carga que cada thread realiza, esta é uma próxima etapa para o projeto.

Especificações da Máquina utilizada

Arquitetura: x86_64
Modo(s) operacional da CPU: 32-bit, 64-bit
Ordem dos bytes: Little Endian
CPU(s): 4
Lista de CPU(s) on-line: 0-3
Thread(s) per núcleo: 2
Núcleo(s) por soquete: 2
Soquete(s): 1
Nó(s) de NUMA: 1
ID de fornecedor: GenuineIntel
Família da CPU: 6
Modelo: 61
Nome do modelo: Intel(R) Core(TM) i7-5500U CPU @ 2.40GHz
Step: 4
CPU MHz: 2430.779
CPU MHz máx.: 3000,0000
CPU MHz mín.: 500,0000
BogoMIPS: 4788.89
Virtualização: VT-x
cache de L1d: 32K
cache de L1i: 32K
cache de L2: 256K
cache de L3: 4096K
CPU(s) de nó0 NUMA: 0-3

Published from doc.pmd using Pweave 0.30.3 on 04-11-2019.