Universidade de São Paulo

Instituto de Ciências Matemáticas e de Computação

Desenvolvimento de Código Otimizado 2023.2

Atividade 5: Otimização em Python

Gabriel da Cunha Dertoni	11795717
Matheus Ventura de Sousa	11345541
Pedro Lucas de Moliner de Castro	11795784
Vitor Caetano Brustolin	11795589

Profa. Dr. Sarita Mazzini Bruschi

São Carlos, Dezembro de 2023

1 Introdução

O objetivo da presente atividade é aplicar algumas técnicas de otimização na linguagem de programação *Python* e analisar o impacto causado por elas no tempo de execução da aplicação final usando o módulo de *profiling* nativo. Para isso, foi desenvolvido um novo código e adaptado um código anterior para a linguagem *Python*.

2 Ambiente de execução

O experimento foi executado usando interpretador *Python* padrão na versão $3.10.12$ e utilizando a biblioteca *numpy* na versão $1.24.2$. A máquina utilizado no experimento tem as especificações descritas na tabela a seguir:

Nome do modelo	Intel(R) Core(TM) i5-9400F CPU @ 2.90GHz
Arquitetura	x86_64
Modos operacionais da CPU	32-bit, 64-bit
Ordem dos bytes	Little Endian
CPUs	6
Threads por núcleo	1
Núcleos por soquete	6
Soquetes	1
Núcleos de NUMA	1
Frequência máxima	4100 MHz
Frequência mínima	800 MHz
Cache de L1d	192 KiB (6 instâncias)
Cache de L1i	192 KiB (6 instâncias)
Cache de L2	1.5 MiB (6 instâncias)
Cache de L3	9 MiB (1 instância)

3 Códigos

3.1 Multiplicação de matrizes

Da atividade 1, adaptou-se em `src/matmul.py` o código de multiplicação de matrizes em suas três versões: sem otimização, aplicando *loop interchange* e aplicando *loop unrolling*. Além dessas três implementações, foram testadas três funções da biblioteca *numpy* que fazem essa mesma operação: `numpy.dot`, `numpy.matmul` e o operador @ com parâmetros do tipo `numpy.matrix`.

3.2 Somas dos valores positivos e negativos

Para tentar explorar as diferentes formas de se iterar sobre múltiplos valores em *Python*, as funções do novo código desenvolvido `src/accumulated.py` recebem uma lista de valores `float` e deve retornar as somas dos valores negativos e positivos.

Foram desenvolvidas seis versões desse algoritmo: interando com índices numéricos e fazendo um *if* dentro do *loop*, utilizando o iterador nativo e fazendo o mesmo *if*, iterando separadamente entre os resultados da função `filter` e acumulando os valores com o operador $+=$, utilizando a função `functools.reduce`, utilizando duas vezes a função `sum` filtrando as entradas com *comprehensions* e chamando duas vezes a função `numpy.sum` filtrando os valores passando uma máscara para o parâmetro `where`.

4 Execução

Para executar os códigos, primeiro instale suas dependências com o auxílio do arquivo `src/requirements.txt` e, depois, utilize o interpretador *Python* passando, opcionalmente, um argumento para determinar o tamanho dos dados teste:

```
python3 -m pip install -r src/requirements.txt
python3 src/accumulated.py
python3 src/matmul.py 150
```

5 Análise de Desempenho

Para determinar o tempo de execução, cada função foi executada 10 e monitorada através da função `timeit.timeit`.

5.1 Multiplicação de matrizes

Multiplicação de matrizes 100×100

Função	Tempo médio de execução (s)
Sem otimização	0.123369
Loop unrolling	0.145130
Loop interchange	0.122631
`numpy.dot`	0.000041
`numpy.matmul`	0.000023
Operador `@`	0.000021

Primeiramente, é possível notar a diferença abismal de performance entre as funções do *numpy* e as implementações diretamente em *Python*. Essa diferença se dá pois as funcionalidades principais do *numpy* são implementadas em *C* e, portanto, compiladas para linguagem de máquina e tendo muito mais controle sobre fatores de baixo nível como a gerência de memória, já as implementações em *Python* devem passar pelo interpretador a cada operação e sofrem com penalidades de performance de algumas abstrações como o *garbage collector*.

Comparando as versões feitas em *Python*, nota-se que a aplicação do *loop unrolling*, na prática, diminui a performance. Uma explicação possível para esse resultado é que, quando iterando em um `range` na versão normal, os incrementos dos índices são feitos internamente na plataforma *CPython*, porém, na versão com *unrolling*, parte desses incrementos é feita explicitamente em código *Python* e, por isso, devem passar de novo pelo interpretador. A aplicação do *loop interchange* resultou em uma diminuição de tempo muito pequena para ser considerada relevante.

Finalmente, fazendo a comparação dos resultados das funções do *numpy*, o procedimento `dot` foi o que apresentou a pior performance, sendo quase duas vezes mais lento que os outros, possivelmente por ser uma função mais genérica que pode realizar outras operações como multiplicação escalar entre vetores. O operador `@` se mostrou ainda um pouco mais eficiente que a função `matmul` provavelmente por apresentar menos indireções ou por ter menos validações dos parâmetros.

5.2 Somas dos valores positivos e negativos

Somas dos valores positivos e negativos com 1000000 de elementos

Função	Tempo médio de execução (s)
Iteração com índices	0.072292
Iteração com iterador	0.048164
Iteração com `filter`	0.134408
`functools.reduce`	0.108753
`sum` com *comprehensions*	0.077751
`numpy.sum`	0.015024

De primeira, é possível ver que a utilização do iterador nativo é significativamente mais rápida que fazer a iteração por meio de um índice numérico. Mais uma vez, isso provavelmente se deve à menor carga de trabalho feita em *Python* e a realização das operações em *CPython*.

As versões utilizando `filter` e `sum` com *comprehensions* foram mais lentas que as versões com somente um *for*, ao que tudo indica, por fazerem duas iterações ao invés de uma. Apesar disso, é notável o ganho de performance ao utilizar a função *builtin* `sum` ao invés de escrever a acumulação de forma iterativa.

O baixo desempenho da função `functools.reduce` pode ser explicado pela necessidade de fazer chamadas de volta para a função de acumulação definida em *Python* que, além de ser implementada em um linguagem mais lenta, precisa fazer operações de destruição e construção de tuplas.

Novamente, a implementação usando *numpy* foi a mais rápida, mesmo que nesse caso ela tenha que fazer quatro iterações sobre os dados, duas para calcular as máscaras e duas para fazer as somas.

6 Conclusão

Em conclusão, a aplicação de métodos mais tradicionais de otimização como *loop interchange* e *loop unrolling* não se mostrou muito efetiva, isso porque o gargalo de performance em *Python* está ligado a outros fatores como a necessidade de utilização de um interpretador e a falta de controle de recursos de baixo nível como memória ou vetores de instruções SIMD. A melhor forma de otimização, então, é a utilização ao máximo das funcionalidades *builtin* como iteradores nativos e a função `sum` ou, melhor ainda, o uso de bibliotecas otimizadas como *numpy* que fazem a ponte entre a API em *Python* e as implementações vetorizadas feitas em linguagens como *C* ou *C++*.