Parallel-Programming Task0

> [!CAUTION]
> *21307289 刘森元*
>
> Code on https://github.com/Myocardial-infarction-Jerry/Parallel-Programming/tree/main/Task0.
>
> Project built by CMake.
>
> ```
> > cd Task0
> > cmake . && make
> ```

0. Environment

11th Gen Intel(R) Core(TM) i7-11700KF @ 3.60GHz

NVIDIA GeForce RTX 3080 Ti 012G

Windows Subsystem for Linux @ Ubuntu 22.04 LTS

1. Task

根据定义使用 C/C++/Python 语言实现一个串行矩阵乘法，并同归对比试验分析其性能.

$$C_{i,j}=\sum_{p=1}^{n}A_{i,p}B_{p,j}$$

> [!IMPORTANT]
> **输入：** m,n,k 三个整数，每个整数的取值范围均为 $[512,2048]$.
>
> **问题描述：** 随机生成 $m\times n$ 的矩阵 A 以及 $n\times k$ 的矩阵 B，并对这两个矩阵进行矩阵乘法运算，得到矩阵 C.
>
> **输出：** A,B,C 三个矩阵，及矩阵计算所消耗的时间 t.
>
> **要求：** 实现多个版本的串行矩阵乘法（可考虑多种语言/编译选项/实现方法/算法/库）. 并比对分析不同因素对最终性能的影响.

2. Theory

由于多个版本之间的矩阵乘法存在较大差异性，本次试验我将使用 C++ STD 标准库来封装一个矩阵乘法的运算，以此保证对比试验不受其他因素干扰.

其中封装具体代码如下：

```cpp
std::vector<std::vector<float>> operator*(const std::vector<std::vector<float>> &A, const
std::vector<std::vector<float>> &B) {
    int m = A.size(), n = A[0].size(), k = B[0].size();
```

```cpp
    std::vector<std::vector<float>> C(m, std::vector<float>(k, 0));

    // Calculation Here

    return C;
}

// Output matrix using std::iostream
std::ostream &operator<<(std::ostream &out, const std::vector<std::vector<float>> &mat) {
    for (auto &row : mat) {
        for (auto &val : row)
            out << val << " ";
        out << std::endl;
    }

    return out;
}
```

使用 `std::chrono` 来计算矩阵乘法运行时间，其提供了高精度的时间戳，能够较为准确的计算相关数据.

```cpp
// Multiply
std::cerr << "Calculating " << m << "*" << n << "*" << k << std::endl;
auto start = std::chrono::high_resolution_clock::now();
C = A * B;
auto end = std::chrono::high_resolution_clock::now();

// Calculate the duration
auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(end - start);
std::cerr << "Multiplication time: " << duration.count() << " ms" << std::endl;
float flops = (2.0 * m * n * k) / (duration.count() * 1e6);
std::cerr << "Performance: " << flops << " GFLOPS" << std::endl;
```

通过查询可知，加速比计算公式如下：

$$S_p = \frac{T_1}{T_p}$$

浮点性能计算公式如下：

$$\begin{aligned}\text{FLOPS} &= \frac{\text{Float point calculation}}{\text{Calculating time}} \\ &= \frac{m \times n \times k \times 2}{T}\end{aligned}$$

查询可知 11th Gen Intel(R) Core(TM) i7-11700KF @ 3.60GHz 的单精度峰值浮点性能为 40 GFLOPS.

3. Code

> **Caution**
>
> 由于程序运行部分大同小异，此处仅贴出部分代码，详细代码请见超链接.

3.1. Python

由于 Numpy 对矩阵乘法计算进行了特殊优化，不能作为朴素矩阵乘法 (Naive Matrix Multiply) 运行效率的比较样本，故于此重新实现了朴素的矩阵乘法.

> ⓘ **Note**
>
> *MatMul.py*
>
> ```python
> def matmul(A, B):
>     m, n, k = len(A), len(A[0]), len(B[0])
>     C = [[0 for _ in range(k)] for _ in range(m)]
>     for i in range(m):
>         for j in range(k):
>             for l in range(n):
>                 C[i][j] += A[i][l] * B[l][j]
>     return C
> ```

3.2. C/C++

统一使用 `std::vector<std::vector<float>>` 进行矩阵封装并进行运算，采用重载运算符方式.

实现了朴素的矩阵乘法.

> ⓘ **Note**
>
> *MatMul.cpp*
>
> ```cpp
> std::vector<std::vector<float>> operator*(const std::vector<std::vector<float>> &A, const
> std::vector<std::vector<float>> &B) {
>     int m = A.size(), n = A[0].size(), k = B[0].size();
>     std::vector<std::vector<float>> C(m, std::vector<float>(k, 0));
>
>     for (int i = 0; i < m; i++)
>         for (int j = 0; j < k; j++)
>             for (int l = 0; l < n; l++)
>                 C[i][j] += A[i][l] * B[l][j];
>
>     return C;
> }
> ```

3.3. C/C++ 调整循环顺序

通过调整循环顺序，减少部分寻址时间.

> ⓘ **Note**
>
> *MatMul_Loop.cpp*

```
std::vector<std::vector<float>> operator*(const std::vector<std::vector<float>> &A, const
std::vector<std::vector<float>> &B) {
 int m = A.size(), n = A[0].size(), k = B[0].size();
 std::vector<std::vector<float>> C(m, std::vector<float>(k, 0));

 for (int i = 0; i < m; i++)
     for (int l = 0; l < n; l++) {
         int a = A[i][l];
         for (int j = 0; j < k; j++)
             C[i][j] += a * B[l][j];
     }

 return C;
}
```

3.4. C/C++ 编译优化

在 CMakeLists.txt 中启用 O3 编译优化. 源码使用朴素矩阵乘法 *MatMul.cpp*.

> ⓘ **Note**
>
> *CMakeLists.txt*
>
> ```
> # Compile MatMul_CompileOptimized with optimization flags
> add_executable(MatMul_CompileOptimized MatMul.cpp)
> target_compile_options(MatMul_CompileOptimized PRIVATE -O3)
> ```

3.5. C/C++ 循环展开

通过手动压缩计算展开循环.

> ⓘ **Note**
>
> *MatMul_LoopExtended.cpp*
>
> ```
> std::vector<std::vector<float>> operator*(const std::vector<std::vector<float>> &A, const
> std::vector<std::vector<float>> &B) {
>  int m = A.size(), n = A[0].size(), k = B[0].size();
>  std::vector<std::vector<float>> C(m, std::vector<float>(k, 0));
>
>  for (int i = 0; i < m; i++) {
>      for (int j = 0; j < k; j++) {
>          int l = 0;
>          for (; l < n - 3; l += 4) {
>              C[i][j] += A[i][l] * B[l][j];
>              C[i][j] += A[i][l + 1] * B[l + 1][j];
>              C[i][j] += A[i][l + 2] * B[l + 2][j];
>              C[i][j] += A[i][l + 3] * B[l + 3][j];
>          }
>          for (; l < n; l++) {
>              C[i][j] += A[i][l] * B[l][j];
>          }
> ```

```
    }
  }

  return C;
}
```

3.6. C/C++ Using Intel MKL

使用 Intel 提供的 oneAPI© 进行矩阵乘法运算. - [Tutorial: Using oneMKL for Matrix Multiplication]()

> ⓘ **Note**
>
> *[MatMul_MKL.cpp]()*

```
std::vector<std::vector<float>> operator*(const std::vector<std::vector<float>> &A, const
std::vector<std::vector<float>> &B) {
  // Get the dimensions of the matrices
  int m = A.size();
  int n = A[0].size();
  int k = B[0].size();

  std::vector<std::vector<float>> C(m, std::vector<float>(k, 0));

  float *a = (float *)mkl_malloc(m * n * sizeof(float), 64);
  float *b = (float *)mkl_malloc(n * k * sizeof(float), 64);
  float *c = (float *)mkl_malloc(m * k * sizeof(float), 64);

  // Copy the matrices to the arrays
  for (int i = 0; i < m; i++)
      for (int j = 0; j < n; j++)
          a[i * n + j] = A[i][j];

  for (int i = 0; i < n; i++)
      for (int j = 0; j < k; j++)
          b[i * k + j] = B[i][j];

  // Perform matrix multiplication using Intel MKL
  cblas_sgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, m, k, n, 1.0, a, n, b, k, 0.0, c,
k);

  // Copy the result back to the matrix
  for (int i = 0; i < m; i++)
      for (int j = 0; j < k; j++)
          C[i][j] = c[i * k + j];

  mkl_free(a);
  mkl_free(b);
  mkl_free(c);
  return C;
}
```

4. Result

> ⚠ Warning
>
> Using `m = n = k = 512`

Version	Running time (ms)	Relative acceleration ratio	Absolute acceleration ratio	GFLOPS	Peak performance percentage
Python	11346	/	/	0.023658	0.05 %
C/C++	894	12.691	12.691	0.300263	0.70 %
C/C++ 调整循环顺序	647	1.3817	17.536	0.414893	1.03 %
C/C++ 编译优化	109	5.9357	104.09	2.46271	6.15 %
C/C++ 循环展开	866	0.1258	13.101	0.309972	0.77 %
C/C++ Using Intel MKL	14	61.857	810.42	19.174	47.9 %

4.1. Python

```
chef@ChefMichelin-PC ▶ python MatMul.py
Enter m, n, k:
512 512 512
Calculating 512*512*512
Multiplication time: 11346 ms
Performance: 0.023658 GFLOPS
```

4.2. C/C++

```
chef@ChefMichelin-PC ▶ ./MatMul
Enter m, n, k: 512 512 512
Calculating 512*512*512
Multiplication time: 894 ms
Performance: 0.300263 GFLOPS
Result in output.txt
```

4.3. C/C++ 调整循环顺序

```
chef@ChefMichelin-PC ▶ ./MatMul_Loop
Enter m, n, k: 512 512 512
Calculating 512*512*512
Multiplication time: 647 ms
Performance: 0.414893 GFLOPS
Result in output.txt
```

4.4. C/C++ 编译优化

```
chef@ChefMichelin-PC ▶ ./MatMul_CompileOptimized
Enter m, n, k: 512 512 512
Calculating 512*512*512
Multiplication time: 109 ms
Performance: 2.46271 GFLOPS
Result in output.txt
```

4.5. C/C++ 循环展开

```
chef@ChefMichelin-PC ▶ ./MatMul_LoopExtended
Enter m, n, k: 512 512 512
Calculating 512*512*512
Multiplication time: 866 ms
Performance: 0.309972 GFLOPS
Result in output.txt
```

4.6. C/C++ Using Intel MKL

```
chef@ChefMichelin-PC ▶ ./MatMul_MKL
Enter m, n, k: 512 512 512
Calculating 512*512*512
Multiplication time: 14 ms
Performance: 19.174 GFLOPS
Result in output.txt
```