# 《高性能计算机系统结构》实验

学号：22721576 姓名：王雪飞

## OpenMP实现矩阵乘法

### 实验要求：

绘制图表：

1. x轴线程数，y轴加速比。 以不同计算量（存储占用不变）作为系列，画出多条曲线。观察计算量和线程数之间的关系。
2. x轴线程数，y轴加速比。以不同矩阵规模作为系列，画出多条曲线。观察计算量、存储量和线程数之间的关系。

### 实验设计

1. 矩阵规模为512，计算A\*A， A\*A\*A，A\*A\*A\*A，

线程数：{1，2，4，6，8，10，12，14，16，18，20，22，24，26，28，30，32}；

1. 计算A\*A，矩阵规模：{128，256，512，1024，2048}，

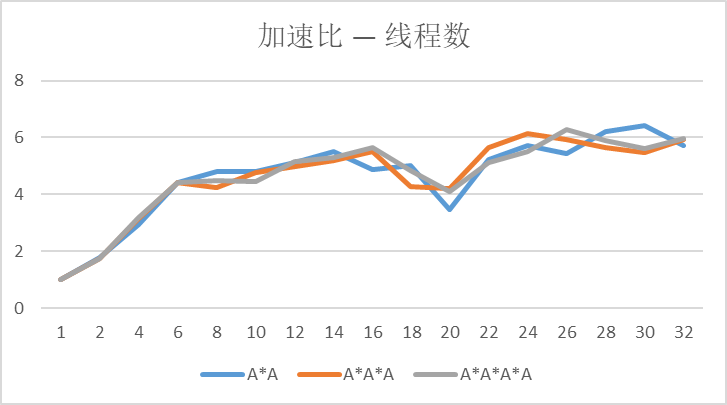
线程数：{1，2，4，6，8，10，12，14，16，18，20，22，24，26，28，30，32}；

### 实验结果

1. 如图表1-1所示，对于矩阵规模为512的方阵A，计算量分别为A\*A， A\*A\*A，A\*A\*A\*A时，随着线程数的增加，加速比总体呈现上升趋势。随着线程数的不断增多，多线程带来的加速优势逐渐减弱，甚至出现因线程数的增多加速比减小的情况（如线程数从18增至20）。

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **线程数/计算量** | **1** | **2** | **4** | **6** | **8** | **10** | **12** | **14** | **16** | **18** | **20** | **22** | **24** | **26** | **28** | **30** | **32** |
| A\*A | 1.00 | 1.78 | 2.94 | 4.40 | 4.82 | 4.82 | 5.12 | 5.51 | 4.86 | 5.01 | 3.47 | 5.23 | 5.72 | 5.44 | 6.21 | 6.40 | 5.72 |
| A\*A\*A | 1.00 | 1.76 | 3.14 | 4.42 | 4.26 | 4.75 | 4.98 | 5.20 | 5.51 | 4.26 | 4.19 | 5.66 | 6.12 | 5.93 | 5.66 | 5.46 | 5.94 |
| A\*A\*A\*A | 1.00 | 1.76 | 3.20 | 4.43 | 4.48 | 4.44 | 5.14 | 5.30 | 5.63 | 4.85 | 4.11 | 5.10 | 5.49 | 6.27 | 5.89 | 5.62 | 5.98 |

表格 1-1 不同计算量下，线程数对加速比的影响



图表 1-1 不同计算量下，线程数对加速比的影响

1. 如图表1-2所示，对不同规模的矩阵作矩阵乘法运算时，增加线程数对加速比的影响效果不同。当矩阵规模较小时（如128，256），较少的线程数对运算加速的效果较明显，随着线程数的进一步增加，加速效果不如线程数少时，甚至可能会因为线程间切换的代价增大而使运算变慢（如矩阵规模为128，线程数为12，14，16，18，20，22）。而当矩阵规模较大时（如1024，2048），加速比基本随着线程数的增加而变大。

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **线程数/矩阵规模** | **1** | **2** | **4** | **6** | **8** | **10** | **12** | **14** | **16** | **18** | **20** | **22** | **24** | **26** | **28** | **30** | **32** |
| 128 | 1.00 | 1.87 | 2.41 | 2.58 | 3.37 | 1.76 | 0.39 | 0.78 | 0.26 | 0.28 | 0.74 | 0.69 | 3.95 | 4.26 | 4.94 | 2.09 | 2.82 |
| 256 | 1.00 | 1.50 | 2.68 | 3.34 | 3.67 | 3.85 | 4.50 | 2.90 | 4.09 | 3.00 | 2.51 | 2.50 | 4.21 | 3.53 | 3.77 | 4.20 | 5.80 |
| 512 | 1.00 | 1.80 | 3.17 | 3.98 | 5.56 | 4.94 | 5.23 | 4.65 | 6.09 | 6.37 | 5.41 | 5.19 | 6.67 | 6.55 | 6.08 | 5.79 | 6.26 |
| 1024 | 1.00 | 1.82 | 3.19 | 3.88 | 6.14 | 5.87 | 6.53 | 6.90 | 7.06 | 7.46 | 6.36 | 7.09 | 7.52 | 7.68 | 7.27 | 7.43 | 7.42 |
| 2048 | 1.00 | 1.65 | 2.90 | 4.52 | 5.69 | 6.41 | 7.30 | 7.82 | 8.69 | 8.63 | 9.05 | 8.45 | 8.18 | 8.69 | 7.94 | 8.47 | 8.70 |

表格 1-2 不同矩阵规模下，线程数对加速比的影响

图表 1-2 不同矩阵规模下，线程数对加速比的影响

## CUDA实现任意一个算法

### 实验要求：

CUDA实现矩阵乘法。

### 实验设计：

分别计算

A[1024][1024] \* B[1024][1024]

A[2048][2048] \* B[2048][2048]

A[4096][4096] \* B[4096][4096]

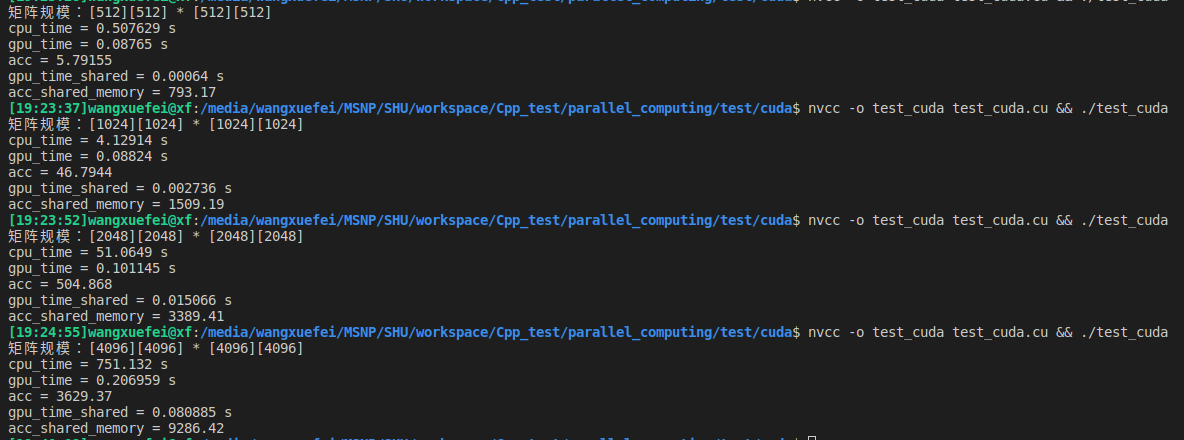
查看GPU对CPU进行矩阵乘法的加速比。

### 实验结果：

如下图所示，对于较大规模的矩阵乘法计算，GPU的加速效果非常明显，并且随着矩阵规模的增大，加速效果更好。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **矩阵规模** | **Cpu\_time(s)** | **Gpu\_time(s)** | **Gpu\_time(s)**  **Shared mem** | **Acc**  **Global mem** | **Acc**  **Shared mem** |
| [1024][1024] \* [1024][1024] | 4.12914 | 0.08824 | 0.002736 | 46.7944 | 1509.19 |
| [2048][2048] \* [2048][2048] | 51.0649 | 0.101145 | 0.015066 | 504.868 | 3389.41 |
| [4096][4096] \* [4096][4096] | 751.132 | 0.206959 | 0.080885 | 3629.37 | 9286.42 |

表格 2-1 GPU对CPU进行矩阵乘法的加速效果



图表 2-1 GPU对CPU进行矩阵乘法的加速效果

## MPI实现矩阵乘法

### 实验要求：

MPI实现矩阵乘法，要求结合OpenMP实现分层次的并行。

### 实验设计：

计算A[2048][2048] \* B[2048][2048]；

使用MPI\_Scatter();将A矩阵按行分发至各进程，MPI\_Bcast();广播B矩阵，各进程计算完成后由MPI\_Gather();收集结果。

### 实验结果：

如以下图表可知，计算矩阵A[2048][2048] \* B[2048][2048]，8进程1线程对运算的加速效果最好。

1. 如图表3-1，加速比随着进程数的增加现增大后减小。线程较少时，多进程能带来更好的加速比；线程数较多时，少进程数能带来更好的加速比。
2. 如图表3-2，当进程数较少时（如1、2、4个进程），加速比随着线程数的增加现增大后减小；当进程数较多时（如8、16、32个进程），加速比随着线程数的增加而减小。

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **线程数/进程数** | **1thrd** | **2thrd** | **4thrd** | **8thrd** | **16thrd** | **32thrd** |
| 1proc | 1.0000 | 2.8021 | 4.0002 | 4.5545 | 4.2028 | 3.7358 |
| 2proc | 1.6958 | 3.7148 | 7.1477 | 5.8195 | 4.9960 | 4.2718 |
| 4proc | 8.4575 | 8.6692 | 7.6651 | 6.6841 | 6.5876 | 6.2787 |
| 8proc | 13.9590 | 8.9606 | 9.0855 | 8.5425 | 8.5471 | 8.4282 |
| 16proc | 12.2733 | 10.7699 | 10.2282 | 10.2327 | 9.7859 | 10.0748 |
| 32proc | 11.9012 | 9.8471 | 10.0967 | 9.6751 | 9.2278 | 10.0519 |

表格 3-1 不同进程/线程数对加速比的影响

图表 3-1 不同线程数下，进程数对加速比的影响

图表 3-2 不同进程数下，线程数对加速比的影响

## Tensor core

### 实验要求：

调研Tensor core，并应用相关指令集进行计算加速，分析并讨论这些技术的趋势和不同。

### Tensor Core：

Tensor Core（张量计算核心）是由NVIDIA研发的新型处理核心，可实现混合精度计算，并能根据精度的降低动态调整算力，在保持准确性的同时提高吞吐量。Tensor Core在NVIDIA的Volta、Turing和Ampere GPU架构中均有配备，其中Ampere架构中的Tensor Core为第三代Tensor Core。NVIDIA Tensor Core技术助力AI实现了大幅加速，将训练时间从数周缩短到几小时，显著加快了推理速度。其发展历史如下：

第一代Tensor Core——NVIDIA Volta架构：

2017年5月发布的NVIDIA Volta架构中的第一代Tensor Core专为深度学习而设计，通过FP16和FP32下的混合精度矩阵乘法提供了突破性的性能——与NVIDIA Pascal相比，用于训练的峰值teraFLOPS(TFLOPS)性能提升了高达12倍，用于推理的峰值TFLOPS性能提升了高达6倍。这项关键功能使Volta提供了比Pascal高3倍的训练和推理性能。

第二代Tensor Core——NVIDIA Turing架构：

2018年8月发布的NVIDIA Turing Tensor Core技术能进行多精度计算，可实现高效的AI推理。Turing Tensor Core提供了一系列用于深度学习训练和推理的精度（从FP32到FP16再到INT8和INT4），性能远超NVIDIA Pascal GPU。Turing架构也首次在消费级GeForce产品线上配备Tensor Core，GeForce RTX20系列显卡具备了全新的AI硬件加速计算能力。

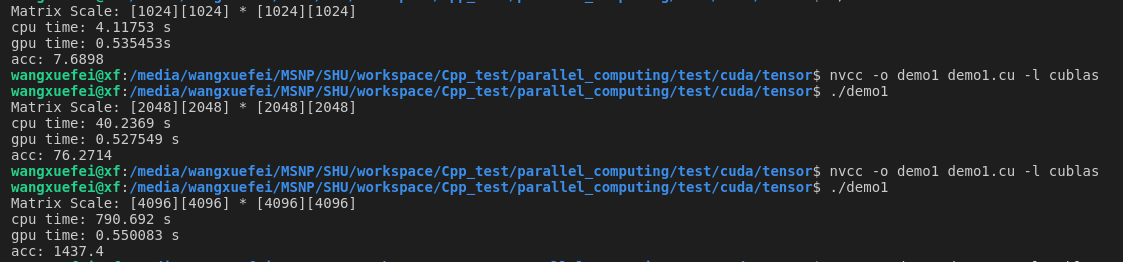
第三代Tensor Core——NVIDIA Ampere架构：

2020年5月发布的NVIDIA Ampere架构带来性能的提升，提供了新的精度，可覆盖研究人员需要的全系列精度（TF32、FP64、FP16、INT8和INT4），从而加速和简化AI的采用，并将NVIDIA Tensor Core的强大功能扩展到HPC领域。

有几种使用Tensor Core的方法，从底层到高层依次为：

* 使用CUDA中的wmma API；
* 使用诸如cuBLAS，cuDNN这样的运算库；
* 通过框架内打开Tensor Core的开关使用，包括pyTorch和TensorFlow；
* 使用框架上的高层库，例如pytorch的Apex矩阵运算库；

### 使用cuBLAS运算库加速矩阵乘法运算：



图表 4-1 使用Tensor Core加速矩阵乘法运算