华创证券 HUA CHUANG SECURITIES

证券研究报告

专题报告 2024 年 03 月 22 日

【专题报告】

并行计算在金融上的应用

❖ **并行计算**

根据菲利(Flynn)的分类体系，系统可分为四种类型：单指令单数据(SISD)、单指令多数据(SIMD)、多指令单数据(MISD)和多指令多数据(MIMD)。随着金融数据量的持续增长以及对分析结果时效性的迫切需求，加速计算任务成为当务之急。充分利用计算机的计算资源，将计算任务从“单指令单数据”向“多指令多数据”转变，已成为一项重要课题。

本篇报告主要介绍了 **CPU 和 GPU 加速计算的原理**；**提供了利用 python 和 C 语言（CUDA）对常见计算进行加速的案例**；**比较了影响计算效率的影响因素**，例如进程数量、数据量大小等。

❖ **CPU 并行**

现代 CPU 通常集成多个核心，每个核心都可独立执行指令。多核计算利用这些核心同时执行不同任务，从而提高计算性能。Python 和 C 语言都支持多进程和多线程编程，使用户能够充分利用多核并行计算。相对于串行计算，合理利用并行计算可将特定任务的计算时间缩短 80%以上。

本报告**提供了 CPU 并行计算案例以及重要参数对计算效率的影响**。测试结果表明，多进程计算随着进程数量的增加，存在“快速提高计算效率”、“效率不再提升”和“效率逐渐下降”三个阶段。

❖ **GPU 并行**

从并行计算的视角来看，英伟达于 2001 年发布的 GeForce 3 代表着 GPU 并行计算的重要突破。随后推出的一系列计算型显卡推动了 GPU 并行计算的发展。利用 GPU 的并行计算能力结合 CUDA 技术，可以实现复杂任务的加速，例如利用 GPU 加速常见的机器学习框架 TensorFlow、PyTorch 等。实验结果显示，相对于串行计算，利用 GPU 计算特定任务可使计算时间缩短 90%以上。

测试结果表明对**大量数据的合并、聚合统计和用户自定义函数 GPU 的加速效果明显，处理数据量较小的任务直接使用 CPU 更好**。

❖ **投资建议：**

用户合理选择加速方式可以大幅提高投研效率，提高投资时效性。

❖ **风险提示：**

本报告中所有统计结果和模型方法均基于历史数据，不代表未来趋势。不同用户硬件差异对加速效果有影响。

华创证券研究所

证券分析师：王小川

电话：021-20572528

邮箱：wangxiaochuan@hcyjs.com

执业编号：S0360517100001

联系人：黄河

邮箱：huanghe@hcyjs.com

相关研究报告

《排序学习选股模型之沪深 300 精选》 2024-03-21

《AI+HI 系列（3）：人工智能在选股与 ETF 轮动上应用》 2024-03-15

《AI+HI 系列（2）：PatchTST、TSMixer、ModernTCN 时序深度网络构建量价因子》 2024-03-11

《2023 年四季报公募基金十大重仓股持仓分析》 2024-01-24

《短期信号回暖，但市场或仍处于底部筑底过程——2023 年策略总结与 2024 年初行情预判》 2024-01-09

投资主题

报告亮点

本报告介绍了如何充分利用计算机的计算资源对计算任务进行加速，提供了 CPU 和 GPU 加速案例，旨在帮助投资者及时完成计算任务。

投资逻辑

更快速的分析结果可以增加投资的时效性。

目 录

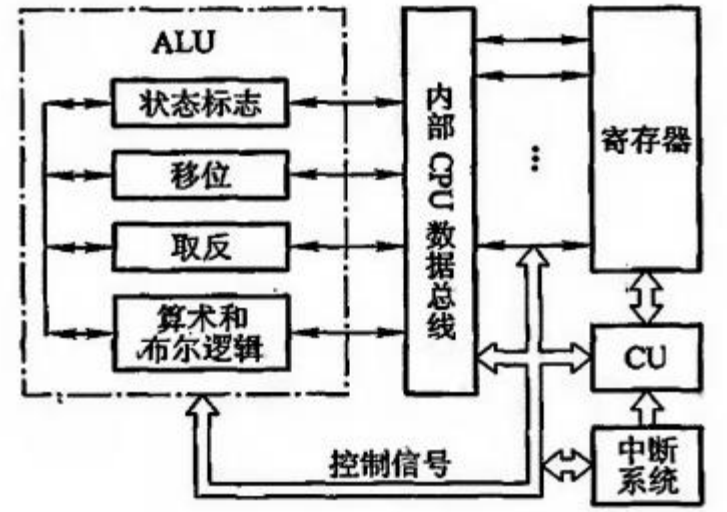
图表目录

一、CPU 并行计算

（一）原理

CPU（Central Processing Unit，中央处理器）逻辑上由三个部分组成，分别是控制单元（下图中 CU 和中断系统）、运算单元（Arithmetic and Logic Unit，ALU）和存储单元（缓存和寄存器），三部分通过内部总线连接起来。

图表 1 CPU 的内部构成



资料来源：唐朔飞《计算机组成原理》第二版

CPU 每取出来并执行一条指令所需的全部时间称为指令周期，也就是 CPU 完成一条指令的时间。在大多数情况下，CPU 就是按照“取指令-执行指令-再取指令-再执行指令…”这个流程进行的。其对应于 CPU 主频频率，在同系列的 CPU 中，**主频越高运行速度越快**。

CPU 的存储单元是 CPU 中暂时保存数据的地方，保存着待处理或者已经处理好的数据，利用寄存器（cache）和缓存，**增加寄存器容量可以减少 CPU 访问内存的次数**，从而提高 CPU 的运行速度。缓存包括 CPU 的一级缓存和二级缓存。

以上为单个 CPU 内核的主要参数，但是对于单个 CPU 核心而言，同一时刻只能运行一个进程。为了提高计算机的运行效率，CPU 逐渐向着单块 CPU 上集成多个 CPU 核心和单个核心可以执行多个进程发展。多核 CPU 指的是将多个 CPU 核心放在一块 CPU 上。多线程技术指的是一个 CPU 核心能够运行多个线程的技术。

CPU 的并行主要有两种方式，多进程和多线程。计算机程序并不能单独执行，只有将程序加载到内存，系统分配资源后才能执行。进程指的是系统中正在运行的程序，是系统分配资源的基本单位，在内存中有完备的数据空间和代码空间。

而线程，是进程的一个实体，是 CPU 调度的基本单位。一个 CPU 核心在同一个时刻只能运行一个线程。一个进程至少拥有一个线程，也可以拥有多个线程。由于同一个核只能运行一个线程，因此当线程数量超过 CPU 核数的时候反而会影响计算速度。

多进程适用于，对于资源管理和保护要求高，不限制开销和效率的任务。因为多进程的每个进程互相独立，都会在内存中开辟计算空间，因此子进程崩溃并不会影响主进程的

稳定，但是对内存的占用比较高。多进程的编程和调试相对于多线程也更加简单。对于有全局锁的编程语言，只能使用多进程。

多线程适用于，需要频繁创建和销毁的任务场景，例如 Web 服务器连接、爬虫等。多线程占用内存少、切换简单、速度很快。但多线程中，子线程报错会影响全部线程。线程之间同步复杂，例如不同线程执行任务快慢不同、某些变量不能同时被两个线程修改。

图表 2 多进程和多线程比较

对比维度	多进程	多线程	总结
c 语言是否支持	支持	支持	相似
python 是否支持	支持	不支持	多进程占优
编程难易程度	简单	复杂	多进程占优
可靠性	进程之间不会互相影响	子线程报错影响全部	多进程占优
扩展性	扩展简单，增加 CPU 即可	扩展复杂	多进程占优
内存占用	占用多	占用少	多线程占优
创建销毁难易程度	难	易	多线程占优

资料来源：华创证券整理

（二）不同 CPU 产品比较

上一节中，我们介绍了影响 CPU 性能的核心因素是频率和核数。本节我们列举了两块常见的 CPU 产品，以产品性能参数具体解释其对计算效率的影响。

以英特尔系列产品为例：

图表 3 酷睿系列产品性能比较

型号	Intel® Core™ i9-12900 Processor	Intel® Core™ i9-13900 Processor
名称	第 12 代智能英特尔® 酷睿™ i9 处理器	第 13 代英特尔® 酷睿™ i9 处理器
内核数	16	24
Performance-core（性能核）数	8	8
Efficient-core（能效核）数	8	16
线程数	24	32
最大睿频频率	5.10 GHz	5.60 GHz
缓存	30 MB Intel® Smart Cache	36 MB Intel® Smart Cache

资料来源：华创证券整理

对于以酷睿 i9 为例，其性能核可以同时运行两个线程，而效能核只能运行一个线程，因此 13 代 i9 总共可以运行 32 个线程，12 代 i9 可以运行 24 个线程。13 代酷睿 i9 在睿频（主频）和存续大小上较 12 代酷睿 i9 也有一定提高。高频率意味着处理器能够更快地执行单个任务，而多核心设计则使其能够同时处理多个任务或多线程任务，从而提高整体计算效率。

在选购 CPU 时，商家通常都会在显著位置标注上述核心指标。从品牌来看，英特尔系列

产品和 AMD 系列产品是目前主流产品，在同等价位下，两者性能差异较小。需要注意的是会有部分机器学习的库依赖 CPU 型号。

此外还有另一种针对服务器类型的 CPU，例如英特尔至强（Xeon）系列。这类 CPU 侧重于性能稳定、支持多路互联等等。在相同的性能下，服务器 CPU 价格通常高于普通 CPU。

（三）Python 实现并行计算

1、Pandas 调用多进程 —— pandarallel

Pandas 是 python 中数据分析常用的一个库，pandarallel 对于 pandas 常见的 API（Application Programming Interface）都有比较好的支持。目前 pandarallel 支持的 pandas API 有：

图表 4 Pandas 原生 API 和 并行 API

Pandas API	并行 Pandas API
df.apply(func)	df.parallel_apply(func)
df.applymap(func)	df.parallel_applymap(func)
df.groupby(args).apply(func)	df.groupby(args).parallel_apply(func)
df.groupby(args1).col_name.rolling(args2).apply(func)	df.groupby(args1).col_name.rolling(args2).parallel_apply(func)
df.groupby(args1).col_name.expanding(args2).apply(func)	df.groupby(args1).col_name.expanding(args2).parallel_apply(func)
series.map(func)	series.parallel_map(func)
series.apply(func)	series.parallel_apply(func)
series.rolling(args).apply(func)	series.rolling(args).parallel_apply(func)

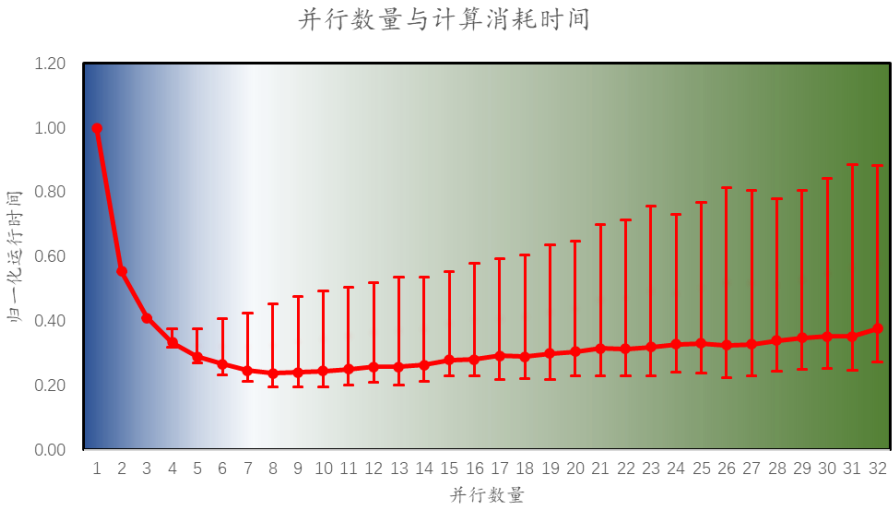
资料来源：华创证券

以下面并行程序为示例：

```
import pandas as pd
import numpy as np
from pandarallel import pandarallel
pandarallel.initialize(progress_bar=True,verbose = 1, nb_workers=8 )
if __name__=='__main__':
    df_size = int(1e7)
    df = pd.DataFrame(dict(rm=np.random.rand(df_size),
                           beta=np.random.rand(df_size),
                           rf=np.random.rand(df_size)))
    def func(x):
        return x.beta*(x.rm-x.rf)+x.rf
    res_parallel = df.parallel_apply(func, axis=1)
    res = df.apply(func, axis=1)
```

这里我们使用了 8 个核去计算资本资产定价模型下资产的收益。其中 pandarallel 为 Pandas 支持的并行库，pandarallel 在使用之前需要进行初始化。progress_bar 为是否显示进度条，nb_workers 为所使用的进程数，进程数不能超过 CPU 所支持的最大线程数。测试运行时间显示如下：

图表 5 进程数量与计算耗时



资料来源：华创证券

为了使得结果可比，我们对每个进程重复了 100 次独立样本，并采用串行计算耗时对并行计算耗时进行归一化处理。测试样本所用的 CPU 为 12 代英特尔酷睿 i7。结果表明，多进程计算随着进程数量的增加，存在“快速提高计算效率”、“效率不再提升”和“效率逐渐下降”三个阶段。其原因在于，一开始，计算机的内存和 I/O 资源是充分的，多进程会迅速提升计算效率，随着进程数量的增加，计算机除线程外的资源会互相竞争导致计算效率下降。

上图也表明，并行计算将给定计算任务时间缩短 80%（归一化计算时间在 0.2）。

2、Python 基于进程并行的库 —— multiprocessing

Multiprocessing 库是一个用来产生进程的包，Multiprocessing 包同时提供了本地和远程并发操作，通过使用子进程而非线程有效地绕过了全局解释器锁。Multiprocessing 提供了多种方法来进行进程间的通信，如队列、管道、共享内存等。这些通信方式使得不同进程之间能够安全地交换数据和信息。Multiprocessing 模块允许程序员充分利用计算机上的多个处理器。

下面是使用 Multiprocessing 的一个示例:

```
import multiprocessing
import target_function
if __name__ == '__main__':
    Process_jobs = []
    for i in range(multiprocessing.cpu_count()):
        p = multiprocessing.Process(target=target_function.func,args=(i,))
        Process_jobs.append(p)
```

```
    for p in Process_jobs:
        p.start()
    for p in Process_jobs:
        p.join()
```

其中 target_function.py 的内容为：

```
import time
def func(i):
    print(f'Starting function {i}')
    time.sleep(20)
print(f'Finished function {i}')
```

运行上述代码可以从任务管理器中观察到计算机打开了多个 python 程序，如下图所示，其中 python 程序的数量取决于计算机的线程数量。

图表 6 多进程下的打开的 python 程序

PyCharm (27)	10.2%	2,161.5 MB	0.7 MB/秒	0.1 Mbps
Filesystem events process...	0%	0.4 MB	0 MB/秒	0 Mbps
PyCharm	0.4%	1,547.4 MB	0.1 MB/秒	0.1 Mbps
Python	0%	60.5 MB	0 MB/秒	0 Mbps
Python	9.8%	11.6 MB	0.7 MB/秒	0 Mbps
Python	0%	26.4 MB	0 MB/秒	0 Mbps
Python	0%	26.2 MB	0 MB/秒	0 Mbps
Python	0%	26.3 MB	0 MB/秒	0 Mbps
Python	0%	26.3 MB	0 MB/秒	0 Mbps
Python	0%	26.3 MB	0 MB/秒	0 Mbps
Python	0%	26.4 MB	0 MB/秒	0 Mbps
Python	0%	26.4 MB	0 MB/秒	0 Mbps
Python	0%	26.2 MB	0 MB/秒	0 Mbps
Python	0%	26.3 MB	0 MB/秒	0 Mbps
Python	0%	26.4 MB	0 MB/秒	0 Mbps
Python	0%	26.2 MB	0 MB/秒	0 Mbps
Python	0%	26.4 MB	0 MB/秒	0 Mbps
Python	0%	26.4 MB	0 MB/秒	0 Mbps
Python	0%	26.3 MB	0 MB/秒	0 Mbps
Python	0%	26.4 MB	0 MB/秒	0 Mbps
Python	0%	26.3 MB	0 MB/秒	0 Mbps
Python	0%	26.4 MB	0 MB/秒	0 Mbps
Python	0%	26.4 MB	0 MB/秒	0 Mbps
Python	0%	26.4 MB	0 MB/秒	0 Mbps
Python	0%	26.4 MB	0 MB/秒	0 Mbps
控制台窗口主机	0%	4.9 MB	0 MB/秒	0 Mbps

资料来源：华创证券

在实际的使用过程中用户还应该结合 quota 和 pipe 等不同进程间交换数据和通信的部分。

3、Python 基于线程并行的库 —— threading

在 CPython 中，由于存在全局解释器锁，同一时刻只有一个线程可以执行 Python 代码，但针对 I/O 密集型任务，多线程仍然是一个合适的选项。I/O 密集型任务指的是需求频

繁进行数据进出操作的任务。因为在 I/O 密集型任务中，大部分时间都花费在等待 I/O 操作完成上，而不是在执行计算密集型操作上。在这种情况下，多线程能够利用等待 I/O 操作的时间来执行其他任务，从而提高了系统的整体效率。

以下为示例:

```
import threading
import time
def function(i):
    print ("function called by thread %i\n" %i)
    time.sleep(10)
    print("function end by thread %i\n" % i)
    return
threads = []
for i in range(5):
    t = threading.Thread(target=function , args=(i,))
    threads.append(t)
    t.start()
    t.join()
```

上述示例程序利用 threading 打印了线程的编码。

（四）C 语言实现并行计算

随着 python 作为一种“胶水语言”的发展，目前将 C 语言作为生产力工具的使用人员越来越少。但是 Cpython 作为广泛使用的一种 python 解释器，背后离不开 C 语言的优良性质。

在 C 语言中，实现并行有两种方法，单进程多线程，或多进程实现，每个进程都有一个或多个线程。多进程适用于线程管理互斥的任务，例如同时提供用户界面和执行后台计算。多线程适合异步输入和输出 (I/O) 、I/O 完成端口、异步过程调用 (APC)以及等待多个事件的能力。

但由于 C 语言受众比较小，我们仅仅提供示例，不做详细介绍。

1、C 语言实现多进程计算

C 语言中，主要是通过 CreateProcess 函数创建独立于创建进程运行的新进程。以下为一个简单的示例程序，其中 cmdLine[]也可以是用户自己的执行文件。

```
#define NUM_PROCESSES 5
int main() {
```

```
    STARTUPINFO si[NUM_PROCESSES];
    PROCESS_INFORMATION pi[NUM_PROCESSES];
    wchar_t cmdLine[] = L"cmd.exe /C echo Hello from child process && pause";
    for (int i = 0; i < NUM_PROCESSES; i++) {
        ZeroMemory(&si[i], sizeof(STARTUPINFO));
        si[i].cb = sizeof(STARTUPINFO);
        if (!CreateProcess(NULL, cmdLine, NULL, NULL, FALSE, 0, NULL, NULL, &si[i],
&pi[i])) {
            fprintf(stderr, "CreateProcess failed\n");
            return 1;
        }
        wprintf(L"Child process %d created with PID: %d\n", i, pi[i].dwProcessId);
    }
    for (int i = 0; i < NUM_PROCESSES; i++) {
        WaitForSingleObject(pi[i].hProcess, INFINITE);
        CloseHandle(pi[i].hProcess);
        CloseHandle(pi[i].hThread);
    }
    return 0;
}
```

上述示例程序打印了不同进程的编号。

2、C 语言实现多线程计算

在 C 语言中使用多线程适合于如下任务：高度并行且可以异步 (例如分布式索引搜索或网络 I/O)、创建和销毁大量线程的应用程序（例如 web 访问）、可以在后台并行处理独立工作项的应用程序（例如加载多个选项卡）。在 C 语言中，主要是通过 CreateThread 函数为进程创建新线程。以下为示例程序：

```
DWORD WINAPI threadFunction(LPVOID lpParam) {
    int threadNumber = *((int*)lpParam);
    printf("Thread %d is running...\n", threadNumber);
    Sleep(3000);
    printf("Thread %d finished.\n", threadNumber);
```

```
    return 0;
}
int main() {
    const int numThreads = 5;
    HANDLE threads[numThreads];
    DWORD threadIds[numThreads];
    printf("Creating %d threads...\n", numThreads);
    for (int i = 0; i < numThreads; i++) {
        int* threadNumber = (int*)malloc(sizeof(int));
        *threadNumber = i + 1;
        threads[i] = CreateThread(NULL, 0, threadFunction, threadNumber, 0, &threadIds[i]);
        if (threads[i] == NULL) {
            fprintf(stderr, "Error creating thread %d\n", i + 1);
            return 1;
        }
    }
    WaitForMultipleObjects(numThreads, threads, TRUE, INFINITE);
    for (int i = 0; i < numThreads; i++) {
        CloseHandle(threads[i]);
    }
    printf("All threads finished.\n");
    return 0;
}
```

上述示例程序打印了不同线程的编号。

二、GPU 并行计算

（一）原理

GPU 的发展离不开图形显示技术的发展。20 世纪 80 年代，用户开始购买 2D 显示加速卡的个人计算机。这些显卡提供了基于硬件的位图运算功能，能够在图形操作系统的显示和可用性起到辅助作用。20 世纪 90 年代，消费者应用程序对显卡的需求快速增长，其中第一人称射击游戏，例如 Doom 等，都要求 PC 游戏创建更加真实的 3D 场景，可以说，第一人称射击游戏在初期为 3D 图形技术在消费者应用中的普及起到了极大的推动作用。

从并行计算角度，2001 年英伟达发布的 GeForce3 代表最重要的突破，GeForce3 是计算工业第一块实现 Direct 8.0 标准的芯片。该标准要求硬件中包含可编程的像素点着色功能。开发人员正是从 GeForce3 系列第一次开始能够对 GPU 中的计算实现某种精度的控制。

在 GeForce3 发布的 5 年后，英伟达发布了 GeForce 8800 GTX。GeForce 8800 GTX 是第一块基于 CUDA（Compute Unified Device Architecture）架构的 GPU。CUDA 架构使得程序可以对芯片上每个数学逻辑单元进行排列，GPU 上的执行单元不仅能任意读写内存，同时可以访问共享内存。CUDA 架构的这些性质使得 GPU 不仅能够执行传统的图形显示，还能高效实现通用计算。换句话说，CUDA 提供了在 GPU 上编程的功能。

CUDA 架构最初被用于物理上的流体力学计算和分子动力学模拟，如今被广泛应用于深度学习等领域。

GPU 于 CPU 最大的不同在于其更多的晶体管被用于数据处理，而不是数据缓存和流量控制，如下图所示：

图表 7　CPU 和 GPU 的架构比较

资料来源：NVIDIA CUDA 编程指南

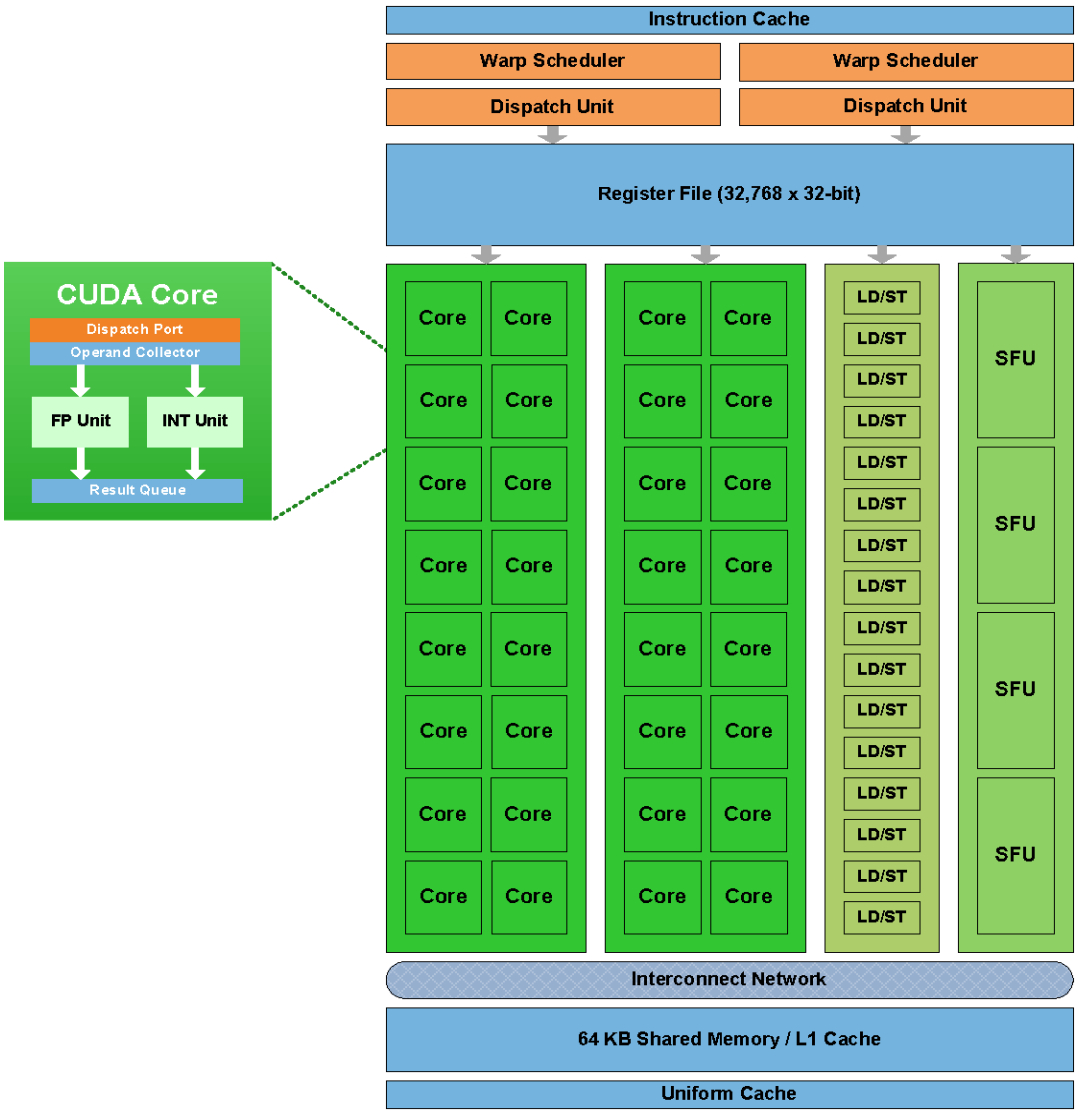
一块 4 核 CPU 可以同时执行 4 个运算，而一块 GPU 可以同时执行成千上万个运算。

2010 年英伟达发布的费米（Fermi）架构是第一个完整的 GPU 架构，后续英伟达陆续发布了开普勒（Kepler）和麦克斯韦（Maxwell）架构。从 2016 年开始英伟达开始向着深度学习发展，推出了 Pascal、Volta、Turing 和 Ampere 等架构。

为了帮助用户理解 GPU 是如何工作的。我们以 2010 年英伟达发布的费米（Fermi）架构为例说明 GPU 能够加速的核心逻辑及与 CPU 的差异。Fermi 计算核心由 16 个 SM（Stream Multiprocesser）组成，每个 SM 包含 2 个线程束（Warp），16 组加载存储单元（LD/ST）和 4 个特殊函数单元（SFU）组成。每个线程束包含 16 个 Cuda Core 组成，每一个 Cuda Core 由 1 个浮点数单元 FPU 和 1 个逻辑运算单元 ALU 组成。

SM 又可以拆分成一组流处理器（Stream Processors），每个流处理器包含一个真正可以用于计算的核心（Core），每个核心都可以执行一个线程。核心是 GPU 计算的最小单元。所有的核心可以在同一个时刻在不同数据上执行同一个程序，这就是 GPU 可以加速计算的关键。每个 SM 包含一定的量的寄存器。这部分寄存器可以存放一些 GPU 上的变量（local variable）。

图表 8 SM 结构



资料来源：Nvidia Fermi 架构白皮书

费米架构共计包含 512 个 CUDA 核心，每个核心包含一个运算单元。在不考虑其他条件情况下，一块费米架构的 GPU 可以同时进行 512 个线程计算，而目前较先进的 CPU 含有 32 个进程，只能同时完成 32 线程的计算。

由于 GPU 在计算的时候，CUDA 核心只能通过 GPU 的内存取数据，费米架构提供了最多 6GB 的 GDDR5 DRAM 内存。GPU 内存中的数据通常会被缓存在高速缓存中，以加快访问速度。缓存可以减少对主存或全局内存的访问次数，从而提高程序的性能和效率。在 CUDA 编程中，需要将数据从主机内存复制到 GPU 内存，以及将计算结果从 GPU 内存复制回主机内存，更大的内存有助于减少数据传输所用的时间。

GPU 带宽用于在主机（CPU）和设备（GPU）之间传输数据。在进行 GPU 计算时，通常需要将数据从主机内存复制到 GPU 内存，以及将计算结果从 GPU 内存复制回主机内存。GPU 带宽决定了数据传输的速度，高带宽意味着可以更快地完成数据传输操作。

（二） 不同 GPU 产品比较

1、计算能力（Compute Capability）

在上一章中，我们提到影响 GPU 性能的核心因素包括了，CUDA 核心、缓存、带宽和内存大小。Nvidia 公司为了比较不同系列的 GPU 在计算方面的能力，给出了一个通用指标，计算能力（Compute Capability）。计算能力由设备的版本号表示，有时也称为其“SM 版本”。该版本号标识了 GPU 硬件支持的特性，并且在运行时由应用程序使用，以确定当前 GPU 上可用的硬件特性和指令。

计算能力包括主要修订号 X 和次要修订号 Y，并以 X.Y 表示。现在的部分机器学习模式会对 GPU 的计算力提出要求，例如 Tensorflow 2.0 要求计算能力不能低于 3.5。

具有相同主要修订号的设备具有相同的核心架构。基于 NVIDIA Hopper GPU 架构的设备的主要修订号为 9，基于 NVIDIA Ampere GPU 架构的设备的主要修订号为 8，基于 Volta 架构的设备的主要修订号为 7，基于 Pascal 架构的设备的主要修订号为 6，基于 Maxwell 架构的设备的主要修订号为 5，基于 Kepler 架构的设备的主要修订号为 3。以下为部分 Nvidia GPU 产品的计算能力。

图表 9 不同产品的计算能力

数据中心产品和工作站产品	计算能力	GeForce 和 TITAN 产品	计算能力	Quadro 和 RTX 桌面 GPU 产品	计算能力
NVIDIA H100	9	GeForce RTX 4090	89	RTX 6000	89
NVIDIA L4	89	GeForce RTX 4080	89	RTX A6000	86
NVIDIA L40	89	GeForce RTX 4070 Ti	89	RTX A5000	86
NVIDIA A100	8	GeForce RTX 3090 Ti	86	RTX A4000	86
NVIDIA A40	86	GeForce RTX 3090	86	T1000	75
NVIDIA A30	8	GeForce RTX 3080 Ti	86	T600	75
NVIDIA A10	86	GeForce RTX 3080	86	T400	75
NVIDIA A16	86	GeForce RTX 3070 Ti	86	Quadro RTX 8000	75
NVIDIA A2	86	GeForce RTX 3070	86	Quadro RTX 6000	75
NVIDIA T4	75	Geforce RTX 3060 Ti	86	Quadro RTX 5000	75
NVIDIA V100	7	Geforce RTX 3060	86	Quadro RTX 4000	75
Tesla P100	6	GeForce GTX 1650 Ti	75	Quadro GV100	7
Tesla P40	61	NVIDIA TITAN RTX	75	Quadro GP100	6

Tesla P4	61	Geforce RTX 2080 Ti	75	Quadro P6000	61
Tesla M60	52	Geforce RTX 2080	75	Quadro P5000	61
Tesla M40	52	Geforce RTX 2070	75	Quadro P4000	61
Tesla K80	37	Geforce RTX 2060	75	Quadro P2200	61
Tesla K40	35	NVIDIA TITAN V	7	Quadro P2000	61
Tesla K20	35	NVIDIA TITAN Xp	61	Quadro P1000	61
Tesla K10	3	NVIDIA TITAN X	61	Quadro P620	61

资料来源：英伟达官网，华创证券

在不同的计算力版本下，GPU 所能完成的任务有差异，主要差异在于 GPU 的 SM 设计差异，下表列出了 SM 的差异。目前，很多利用 GPU 的计算任务都会对算力提出要求。

图表 10　不同计算能力 GPU 的差异

计算能力	5	5.2	5.3	6	6.1	6.2	7	7.2	7.5	8	8.6	8.7	8.9	9
每个设备的最大常驻网格数量（并发核函数执行）	32		16	128	32	16	128	16	128					
每个 SM 的最大常驻块数量	32								16	32	16		24	32
每个 SM 的最大常驻线程束数量	64								32	64	48			64
每个 SM 的最大常驻线程数量	2048								1024	2048	1536			2048
每个线程块的 32 位寄存器的最大数量	64 K		32 K	64 K		32 K	64 K							
每个 SM 的最大共享内存量	64 KB	96 KB	64 KB		96 KB	64 KB	96 KB		64 KB	164 KB	100 KB	164 KB	100 KB	228 KB
每个线程块的最大共享内存量（32 位）	48 KB						96 KB	96 KB	64 KB	163 KB	99 KB	163 KB	99 KB	227 KB

每个 SM 的缓存	Between 12 KB and 48 KB	Between 24 KB and 48 KB	32 ~ 128 KB	32 or 64 KB	28 KB ~ 192 KB	28 KB ~ 128 KB	28 KB ~ 192 KB	28 KB ~ 128 KB	28 KB ~ 256 KB

资料来源：英伟达官网，华创证券

2、其他影响因素

GPU 的 SM 计算能力相同的情况下，不同 GPU 主要是由其架构、CUDA 核心数量、内存大小和内存带宽影响。例如在相同算力的情况下，面向计算的显卡 NVIDIA L4 其功耗方面有优势，面向消费者的 GeForce RTX 4090 在核心数量方面有优势。

面向不同任务的时候，所需要考虑的因素不同，例如进行大规模计算的时候，显卡功耗越低，其电力需求成本越小，而面向消费者的显卡为了追求性能，并不会考虑功耗。因此，如果是个人消费者，主要目的是打游戏，把深度学习作为研究兴趣，建议配置 GeForce 高端卡，如果是深度学习或并行计算研究员，建议配置对应的专业卡。

（三）Python 利用 GPU 进行并行计算

1、pyCUDA

Python 利用 GPU 并行计算的一个重要包是 PyCUDA。CUDA 是通过对 C 语言的扩展实现的。CUDA 的一个核心思想是“层次”编程，其将程序分解为两部分，一部分通过 CPU（host）执行，另一部分通过 GPU（device）执行。

通常在使用 pyCUDA 的时候分为如下几个步骤:

首先在 GPU 上分配内存;

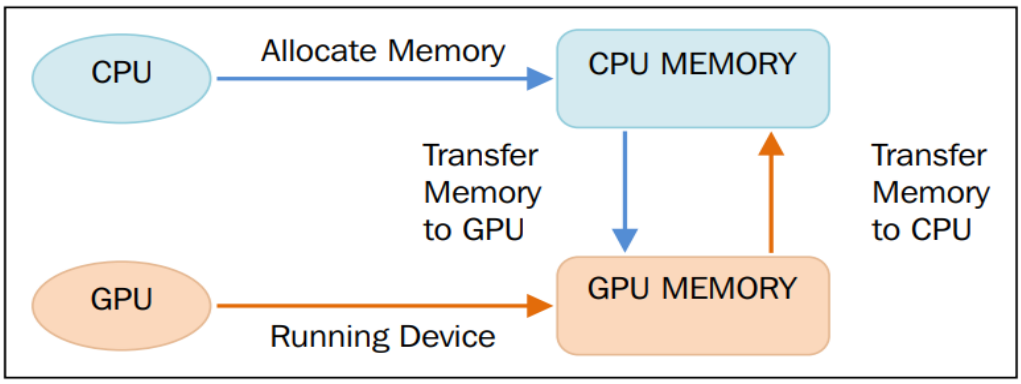
复制 host 上的内容进入 device;

编译并运行在 GPU 上的函数（kernel function）;

将结果复制进入 host;

释放 GPU 上的内存。

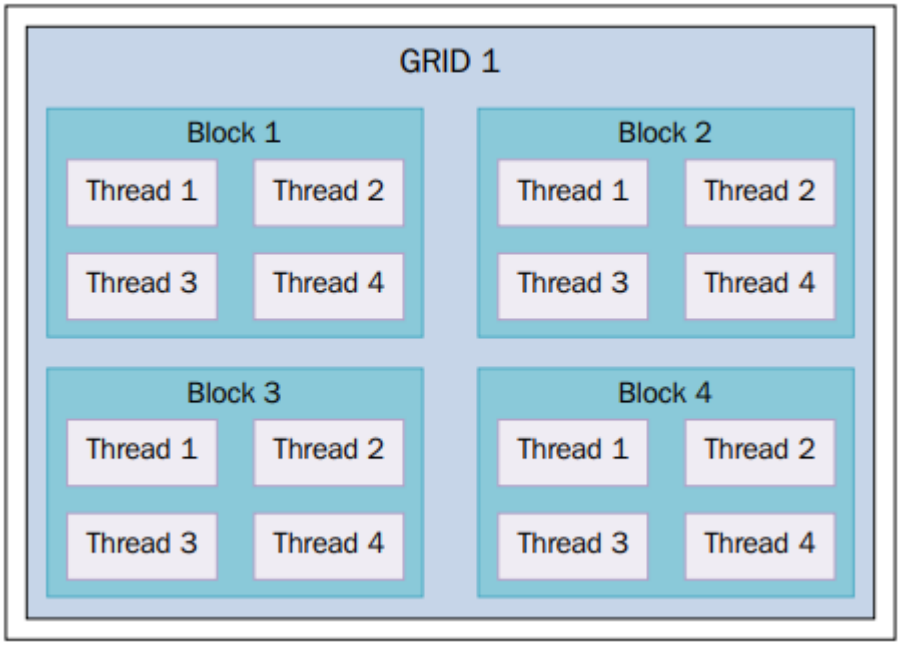
图表 11　GPU 计算步骤



资料来源：Giancarlo Zaccone, “python parallel programming cookbook.”

上图我们简单描述了，pyCUDA 是如何执行程序的。在具体执行的时候，需要将程序指定到不同的须（threads）上。CUDA 通过两个层级定义须，分别是 grid 和 block，通过这样一个结构可以指定不同的须去控制 GPU 执行程序。

图表 12　GPU 块结构



资料来源：华创证券

实际在使用时，面对不同的程序任务，设置不同的 grid 和 block 大小，甚至是 block 的维度都会对程序产生影响。另外数据调度的时间也不能忽略，从 CPU 调度数据进入 GPU、从 GPU 拷贝结果到 CPU 的时间都会对计算时间造成影响。

由于本报告旨在提供给投资者一个可实践可复现的加速手段。因此如何利用 pyCUDA 计算并不作为本报告的主要内容。如果投资者希望提供定制化加速服务，欢迎联系华创金工。

这里，我们主要介绍一些对不同模块进行加速的包。

2、可以支持 DaraFrame 的加速库——RAPIDS

RAPIDS 是一个开源软件库，旨在加速数据科学和机器学习工作流程。它由 NVIDIA 推出，利用 GPU 的并行计算能力来加速数据处理和机器学习任务。RAPIDS 提供了一系列用于数据预处理、特征工程、机器学习和深度学习的库和工具，旨在使数据科学家和机器学习工程师能够更快地分析和处理大规模数据集。RAPIDS 库的核心组件包括:

cuDF: 基于 GPU 的数据框架，类似于 Pandas，用于高效地处理和分析结构化数据。cuDF 提供了 Pandas API 的 GPU 实现，可以在 GPU 上直接执行数据操作，从而实现更快的数据处理速度。

cuML：基于 GPU 的机器学习库，提供了各种常见的机器学习算法的 GPU 实现，包括线性回归、逻辑回归、决策树、随机森林、聚类、降维等。cuML 可以利用 GPU 的并行计算能力来加速模型训练和推理过程，从而缩短训练时间并提高模型性能。

cuGraph：基于 GPU 的图分析库，用于在大规模图数据上执行图算法。cuGraph 提供了各种图算法的 GPU 实现，包括最短路径、连通性、图聚类、图分区等，可以加速图数据的分析和处理过程。

cuSpatial：基于 GPU 的空间数据分析库，用于处理和分析地理空间数据。cuSpatial 提供了各种空间数据算法的 GPU 实现，包括距离计算、空间索引、空间连接等，可以加速地理空间数据的处理和分析过程。

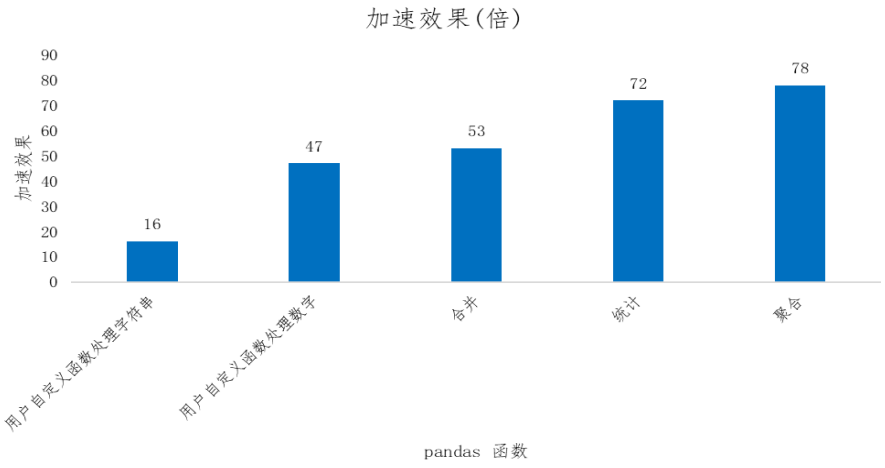
由于我们的服务群体主要使用 DataFrame，我们着重介绍 cuDF。cuDF 是一个使用 GPU 对 DataFrame 进行加速的库，可以被用于数据聚合、连接等操作。用户使用 cuDF 对原有程序进行加速时，不需要改动任何原有程序，只需要指明利用 GPU 加速运行 Pandas 即可。例如以下面这段程序为例：

```
df_size = int(1000000)
df = pd.DataFrame(dict(rm=np.random.rand(df_size),
                       beta=np.random.rand(df_size),
                       rf=np.random.rand(df_size)))
def func(x):
    return x.beta * (x.rm - x.rf) + x.rf
df.apply(func, axis=1)
```

在 Intel(R) Core(TM) i7-12700 上的运行时间为 11s，利用 cuDF 加速后在 NVIDIA T1200 Laptop GPU 上 1s 内完成计算，加速效率高达 11 倍以上。

对比来看，对 Pandas 的常见函数有至少 16 倍加速效果，最高 78 倍加速效果：

图表 13　GPU 对 Pandas 的加速效果



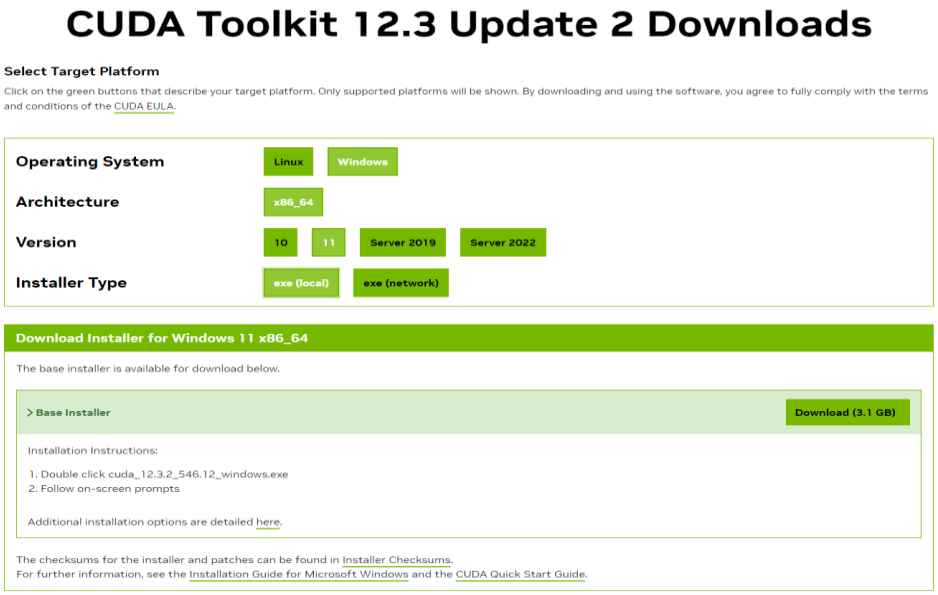
资料来源：华创证券

我们以 cuDF 为例，介绍如何对常规 pandas 进行替换和加速，系统为 window11 家庭中文版，有条件的开发者建议直接选择 linux 系统。

用户首先需要确保：CUDA 版本在 11.2 以上、NVIDIA 驱动在 450.80.02 以上、显卡的计算力在 7.0 以上、系统为 Ubuntu 20.04 或 22.04、CentOS 7、Rocky Linux 8、WSL2 on Windows 11。

如果用户需要从零开始安装，首先需要在官网下载适合自己的 CUDA TOOLKIT（图 14 所示）。在 window 系统下直接安装即可，在 linux 系统下，建议做好备份，以命令行模式安装，因为 CUDA 自带驱动，可能会与 linux 自带驱动冲突。

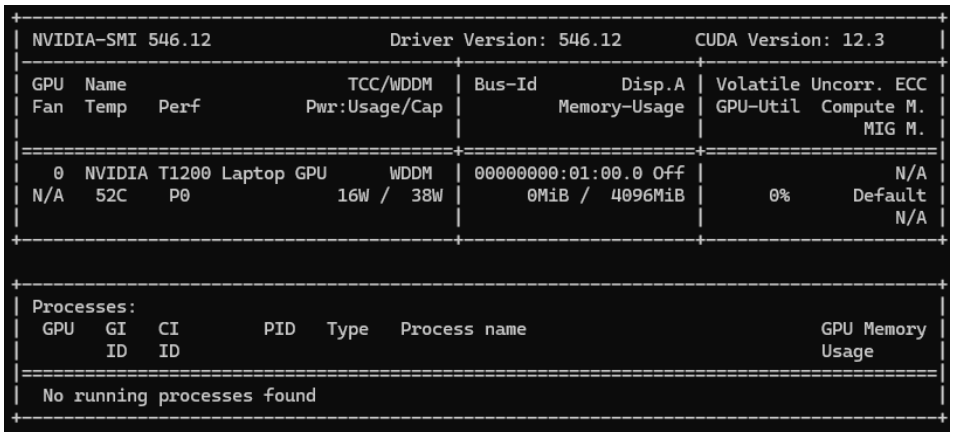
图表 14 CUDA Toolkit



资料来源：英伟达官网

Window 和 linux 下的查看方式是打开命令行，输入 nvidia-smi 获得如下结果（图 15 所示）：

图表 15 查看显卡驱动和 CUDA 版本



资料来源：华创证券

上图中，DriverVerision 为 546.12 ，CUDA version 为 12.3。接下来，用户可以打开 Cuda toolkit 的安装包，并运行 deviceQuery 获得计算的计算力（图 16 所示）:

图表 16 显卡算力查询

```
deviceQuery.exe Starting...

 CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: "NVIDIA T1200 Laptop GPU"
  CUDA Driver Version / Runtime Version          12.3 / 12.3
  CUDA Capability Major/Minor version number:    7.5
  Total amount of global memory:                 4096 MBytes (4294705152 bytes)
  (16) Multiprocessors, ( 64) CUDA Cores/MP:     1024 CUDA Cores
  GPU Max Clock rate:                            1425 MHz (1.42 GHz)
  Memory Clock rate:                             5001 Mhz
  Memory Bus Width:                              128-bit
  L2 Cache Size:                                 1048576 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(131072), 2D=(131072, 65536), 3D=(16384, 16384, 16384)
  Maximum Layered 1D Texture Size, (num) layers  1D=(32768), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(32768, 32768), 2048 layers
  Total amount of constant memory:               zu bytes
  Total amount of shared memory per block:       zu bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  1024
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          zu bytes
```

资料来源：华创证券

其中，CUDA Capability Major 显示算力为 7.5。用户也可以在 NVIDIA 官网（CUDA GPUs - Compute Capability | NVIDIA Developer）直接查看对应显卡的计算力，如下图所示（图 17 所示）:

图表 17 英伟达官网的 GPU 算力表

CUDA-Enabled Datacenter Products

Tesla Workstation Products

GPU	Compute Capability
Tesla K80	3.7
Tesla K40	3.5
Tesla K20	3.5
Tesla C2075	2.0
Tesla C2050/C2070	2.0

NVIDIA Data Center Products

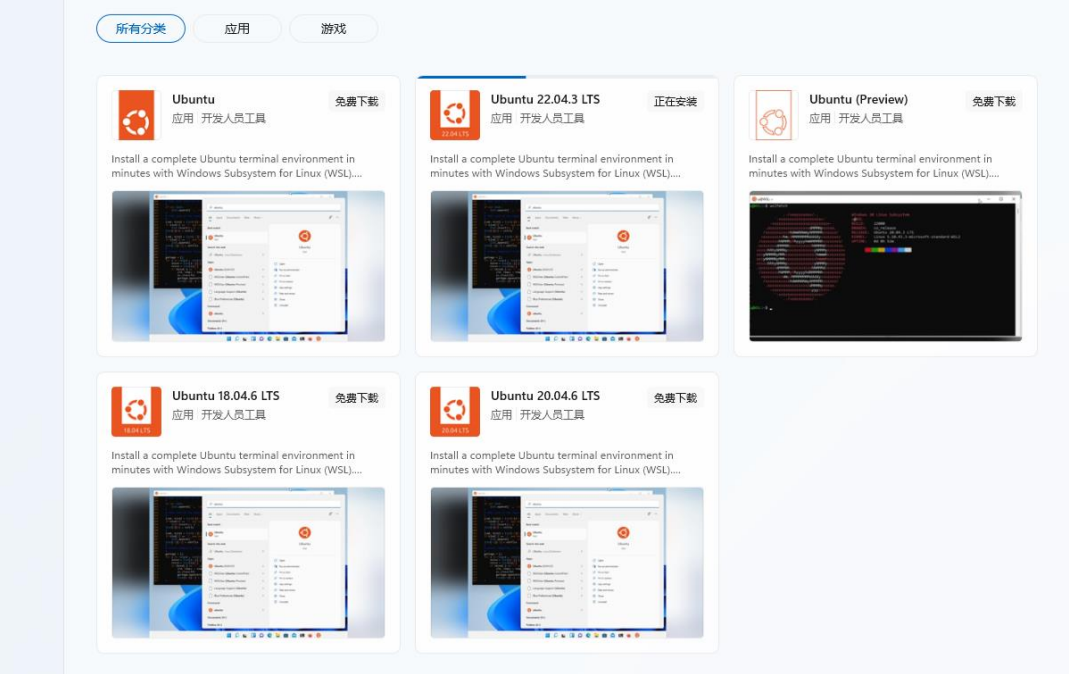
GPU	Compute Capability
NVIDIA H100	9.0
NVIDIA L4	8.9
NVIDIA L40	8.9
NVIDIA A100	8.0
NVIDIA A40	8.6
NVIDIA A30	8.0
NVIDIA A10	8.6
NVIDIA A16	8.6
NVIDIA A2	8.6
NVIDIA T4	7.5
NVIDIA V100	7.0

资料来源：英伟达官网

常见的 RAPIDS 支持的消费显卡需要在 Geforce RTX 2060 或 T500 以上。

由于 cuDF 只能在 Linux 系统下运行，因此我们需要在 Window 系统下安装 Linux 虚拟机。有两种方法，一、在 Windows 系统下，通过 wsl 安装 Linux 系统，默认为安装 Ubuntu 22.04。二、在 Windows 应用商店搜索 Ubuntu 进行安装。

图表 18 window 应用商店



资料来源：Microsoft Store，华创证券

打开 wsl 后，有三种方式安装 cuDF 库：

通过 Conda 安装：这种方式是通过 Conda 建立了 RAPIDS 的环境。用户可以在这个环境下进行开发。缺点是用户使用时需要先指定所使用的库。

通过 pip 安装：直接通过 pip 安装的方式简单直接，用户使用时可以直接导入。这种方式的缺点是不同库之间可能会存在依赖，导致部分库无法使用。

源码安装：这种方式针对不希望引入过多的依赖包的用户。安装和使用都较为不便。

图表 19 不同安装 RAPIDS 方式对比

方式	过程	优点	缺点	推荐与否
conda	conda create --solver=libmamba -n RAPIDS-24.02 -c RAPIDSai -c conda-forge -c nvidia RAPIDS=24.02 python=3.10 cuda-version=12.0	开发环境容易管理	开发者没有开发环境管理的习惯	推荐
pip	pip install --extra-index-url=https://pypi.nvidia.com cudf-cu11	操作简单	不同包之间容易互相干扰	推荐
源码安装	编译源码安装	下载源码包速度快	安装流程复杂	不推荐

资料来源：华创证券

安装完成 cuDF 后，用户可以用三种方式调用 cuDF，如下表所示：

图表 20 调用 GPU 加速 Pandas 的对比

命令	优点	缺点
%load_ext cudf.pandas import pandas as pd	需要 IPython 或 Jupyter Notebooks	用户不使用 Ipython 或 jupyter notebooks
python -m cudf.pandas script.py	简单直接	用户习惯使用 IDE 开发
import cudf.pandas cudf.pandas.install() import pandas as pd	简单直接	需要修改程序

资料来源：华创证券

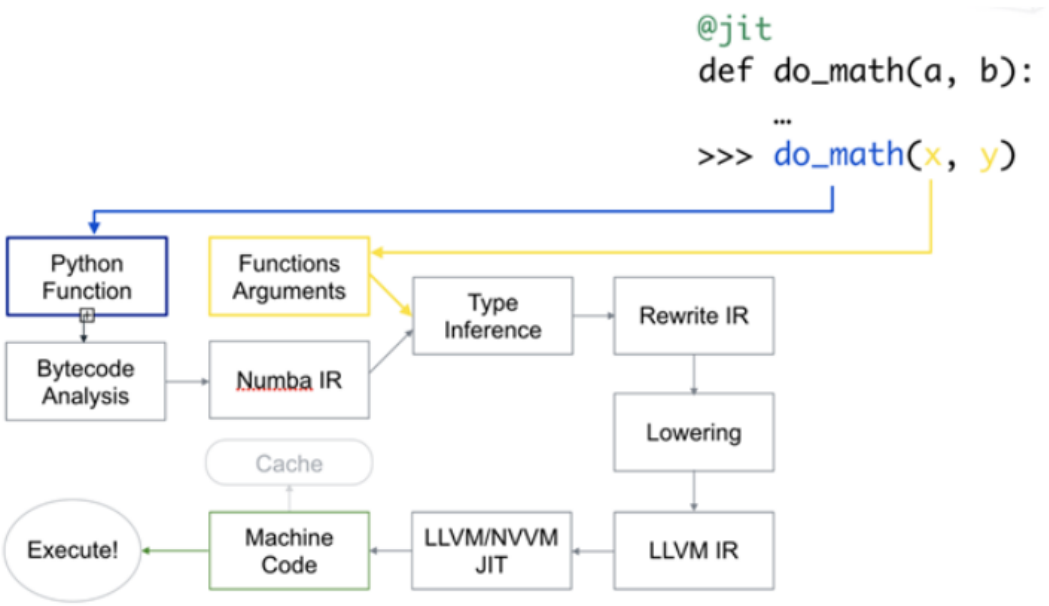
3、可以支持 numpy 的加速库——Numba

Numba 是一个适用于 Python 代码的开源式即时编译器。借助该编译器，开发者可以使用标准 Python 函数在 CPU 和 GPU 上加速数值函数。为了提高执行速度，Numba 会在执行前立即将 Python 字节代码转换为机器代码。Numba 无需新语法，也无需替换 Python 解释器或安装 C/C++ 编译器。只需将 @jit Numba 修饰器应用于 Python 函数即可，Numba 会自动利用 GPU 对函数进行优化，如果 Numba 发现报错会返回利用 CPU 进行运算。

Numba 比 python 快的原因在于 Numba 使用 LLVM 编译器替换了纯 Python 的 cpython 编译器。

Numba 的运行流程如下所示（图 21 所示）:

图表 21 Numba 运行流程



资料来源：英伟达官网

简要的来说，通过修饰函数，程序首先将 python 函数变成二进制代码，再将二进制代码，进入 LLVM 虚拟机编码，最后转成机器语言。示例程序如下所示:

```
from numba import jit

import pandas as pd

x = {'a': [1, 2, 3], 'b': [20, 30, 40]}

@jit(forceobj=True, looplift=True)

def use_pandas(a):

    df = pd.DataFrame.from_dict(a)

    df += 1

    return df.cov()

print(use_pandas(x))
@cuda.jit
def add_array(a, b, c, d):
    i = cuda.threadIdx.x + cuda.blockDim.x * cuda.blockIdx.x
    if i < a.size:
        d[i] = c[i]* a[i] + c[i]


num_columns = 3
num_rows = 8000000
random_data = np.random.rand(num_rows, num_columns)
a=list(random_data[:,0])
b=list(random_data[:,1])
c=list(random_data[:,2])


def gouadd(a,b,c):
    dev_a = cuda.to_device(a)
    dev_b = cuda.to_device(b)
    dev_c = cuda.to_device(c)
    dev_d = cuda.device_array_like(a)
    threads_per_block = 256
    blocks_per_grid = (num_rows + (threads_per_block - 1))
    add_array[blocks_per_grid, threads_per_block](dev_a, dev_b,dev_c, dev_d)
    d = dev_d.copy_to_host()
    return d
gouadd(random_data)
```

值得一提的是，一方面，GPU 计算需要将数据复制进入 device，再将计算好的结果复制进 host。如果数据量太少，直接使用 CPU 计算甚至会快于 GPU 计算；另一个方面，GPU 计算对 thread 数量设置和核函数写法有很高的优化要求。如无特殊需要，建议大家直接使用 numba 的修饰器。

Numba 的缺点主要包括，Numba 支持 python 原生函数和 numpy 函数，对 python 的类和其他库支持有限。

（四）CUDA

CUDA 允许开发人员使用 C 编程语言来编写并行程序，这些程序可以在 NVIDIA 的 GPU 上执行。如果没有必要，我们建议直接使用 NVIDIA 提供的一系列高性能 CUDA 库，涵盖了各种领域的计算需求，包括线性代数、图像处理、信号处理等等。开发者自己编写的核函数、数据传输和程序架构很难在性能上超过 NVIDIA 提供的计算库。

常用的 CUDA 计算库包括有 cuDNN（CUDA Deep Neural Network）、cuBLAS（CUDA Basic Linear Algebra Subprograms）等。

下面是利用 CUDA 写的计算残余 alpha 的示例（这里仅展示核函数，涉及到在 GPU 开辟计算空间，拷贝 host 数据进入 device 部分，由于篇幅所限不再展示）:

```
__global__ void addKernel(float* c, float* r, float* beta, float* rf, float* rm, int N)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if(i<N)
    {c[i] = r[i] - beta[i] * (rm[i] - rf[i]) - rf[i]; }
}
```

通过上述示例中 blockDim 和 threadIdx 可以定位 GPU 中的“须结构”，送入 GPU 进行计算。计算的结果需要从 GPU 拷贝进入 CPU，由于数据在 host 和 device 的转移需要时间，因此数据量太少反而利用 CUDA 会慢。

为了更好的让读者理解 CUDA 的工作流程，我们将简述其工作原理:

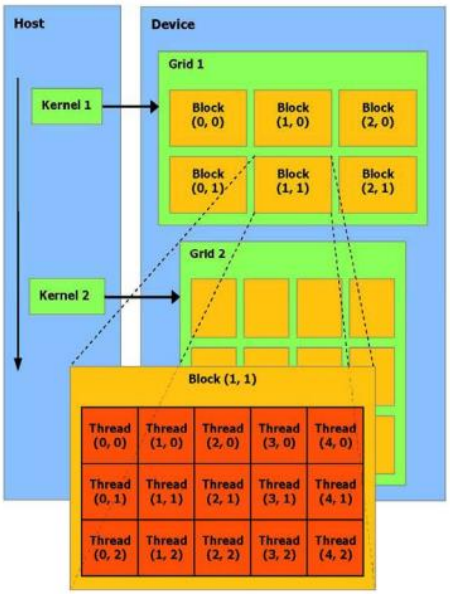
CPU（host）首先将任务分发给 GPU(device)，GPU 会对 CPU 分发的任务进行处理，GPU 的同一个线程块（grid）的众多须（Thread）会处理同样一个核（kernel）函数（如图 22）。

“须”在线程块中有线程编号，根据线程的编号可以进行复杂寻址，一个应用程序可以指定一个块作为一个二维或三维数组的任意大小，并且通过二维或三维索引代替来指定每条线程。比如对于一个大小为 (Dx，Dy)二维线程块，线程的索引是(x, y)，这个线程编号是(x + y Dx)，对于一个三维的大小为 (Dx，Dy，Dz)的块，这个线程的索引是(x，y，z)，线程的编号是(x + y Dx + z DxDy)。

一个块可以包含的线程最大数量是有限的，而不同的块之间线程协作的减少会造成性能的损失，即使在同一个栅格中，不同线程块的线程彼此之间也不能通讯和同步。一个须只能访问其所在的线程块的内存空间。

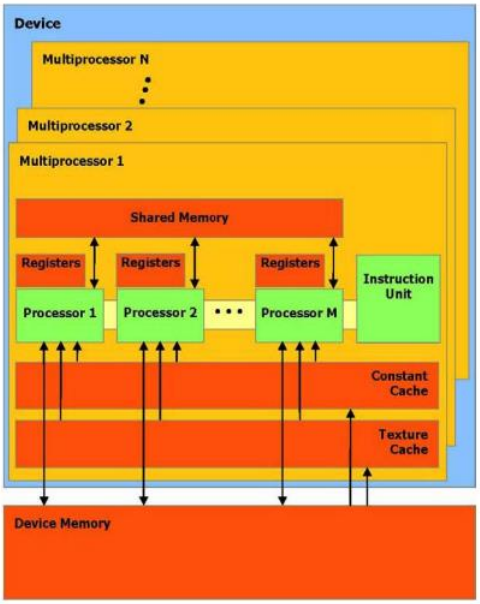
GPU 的存储可以同时被 CPU 和 GPU 访问，其内存大小有限（如图 23）。由于现在的深度学习模型越来越大，因此 GPU 的内存显得格外重要，小内存 GPU 无法承载庞大深度学习参数。另外，在深度学习中，通常会使用批处理训练数据来加速训练过程。较大的批处理通常会导致更多的内存使用，因为需要同时存储更多的参数和计算结果。

图表 22 批处理过程



资料来源：Nvidia CUDA Programming Guide

图表 23 GPU 内存空间



资料来源：Nvidia CUDA Programming Guide

三、案例

在数据表的操作中，聚合、分组计算和分组应用用户自定义函数都非常常见。我们以一款常见的台式机配置分别对不同的计算任务进行了 CPU 计算和 GPU 计算的比较。测试所用的硬件配置为：

图表 24 测试硬件比较

硬件	笔记本	台式机
CPU	12th Gen Intel(R) Core(TM) i7-12700H	13th Gen Intel(R) Core(TM) i9-13900K
GPU	NVIDIA T1200 Laptop GPU	NVIDIA RTX A2000
系统	Linux Ubuntu 22.04	Linux Ubuntu 22.04
内存	48G DDR5	32G DDR5

资料来源：华创证券整理

结果表明，用户应当根据自己的计算数据量大小选择合适的加速方式，数据量在 100000 以下时，建议使用 CPU，数据量超过 1000000 时，建议使用 GPU。例如，月度数据计算建议使用 CPU 计算，日度数据建议根据数据量选择 CPU 或 GPU，股票 tick 数据计算建议使用 GPU 计算。

（一）数据聚合操作

数据聚合是数据处理过程中常见的任务，它的作用是可以根据一个或多个键将不同的 DatFrame 连接起来。Merge 是 python pandas 一个常用的函数，类似于 mysql 中的 join 函数和 excel 中的 vlookup 函数。

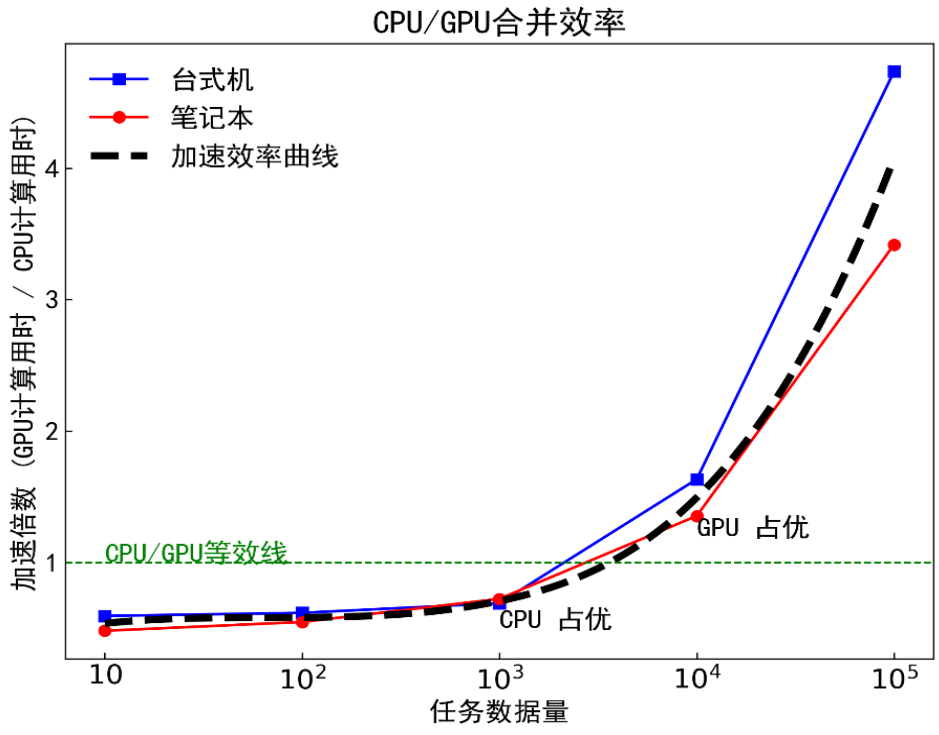
以下面程序示例：

```
pdf = pd.DataFrame(
    {
        "name1":    np.random.choice(
            ["Bank", "Tech", "Food", "Mining"], size=num_rows
        ),
        "name2": np.random.choice(
            ["Bank2", "Tech2", "Food2", "Mining2"], size=num_rows
        ),
    }
)
gdf = cudf.from_pandas(pdf)
pandas_groupby, cudf_groupby = timeit_pandas_cudf(
    pdf,
    gdf,
    lambda df: df.merge(df,on = 'name1'),
    number = 100
)
```

上述函数随机建立一个 dataframe，有 name1 和 name2 两列。程序是通过 name1 的键值进行匹配。

结果如下图所示（图 25 所示），当数据量在 1000 以下时，CPU 比 GPU 统计效率高，当数据量超过 10000 后 GPU 统计效率高。随着数据量的增加，GPU 的表现比 CPU 表现更好:

图表 25　CPU 和 GPU 合并数据时的效率比较



资料来源：华创证券

（二）数据分组统计操作

在 Pandas 中，groupby 函数根据一个或多个列的数值对 DataFrame 进行分组。一旦分组完成，可以在每个组上执行聚合操作，比如计算平均值、求和、计数等。类似于 sql 中，GROUP BY 用于将行分组为汇总行，并且结合聚合函数（如 COUNT、SUM、AVG 等）可以对每个组进行汇总计算。Excel 中类似的功能为数据透视表。

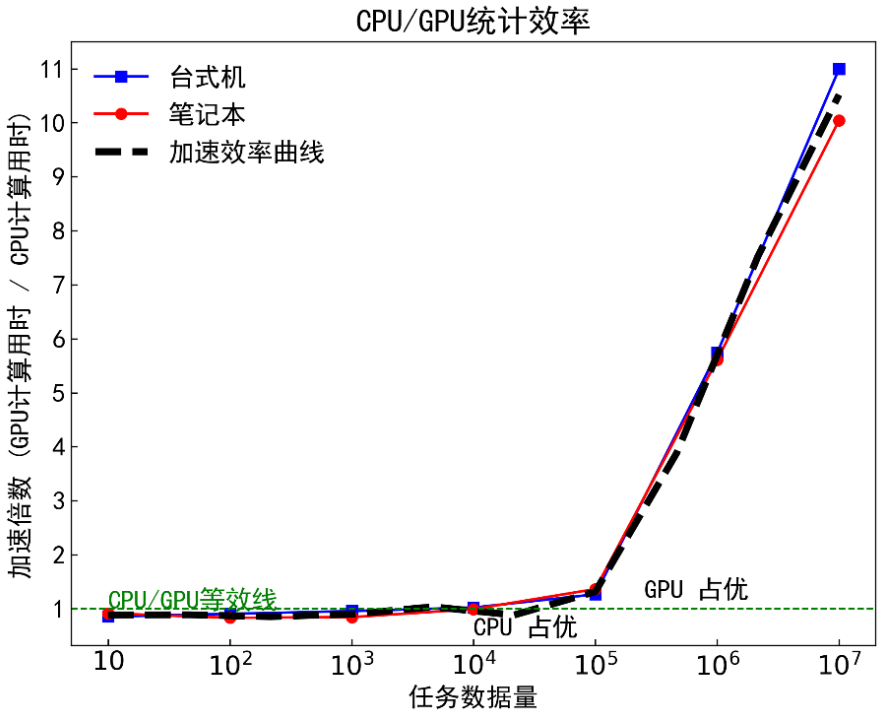
下面为示例函数:

```
pdf = pd.DataFrame(
    {
        "alpha": np.random.uniform(-1,1,num_rows ),
        "indu": np.random.choice(
            ["Bank", "Tech", "Food", "Mining"], size=num_rows
        ),
    }
)
gdf = cudf.from_pandas(pdf)
pandas_groupby, cudf_groupby = timeit_pandas_cudf(
    pdf,
    gdf,
    lambda df: df.groupby("indu").agg(["min", "max", "mean","count"]),
    number = 100
)
```

上述函数建立了一个若干行的 dataframe，包含两列，分别是行业和对应的收益，函数作用是统计不同行业的最小值，最大值，均值和个数。

结果如下图所示（图 26 所示），当数据量在 10000 以下时，CPU 比 GPU 统计效率高，当数据量超过 100000 后 GPU 统计效率高。随着数据量的增加，GPU 的表现比 CPU 表现更好，当数据量在千万量级时，台式机上 CPU 的计算时间是 GPU 的 11 倍，GPU 使得任务完成时间缩短 90%以上。

图表 26 CPU 和 GPU 统计数据时的效率比较



资料来源：华创证券

（三）运行用户自定义函数

Pandas 中 apply 是一个在数据处理中经常使用的函数，它允许在 DataFrame 或 Series 中的每一行或每一列上应用自定义的函数。在 Pandas 中，apply 可以用于对数据进行逐行或逐列的操作，通常与 lambda 函数或自定义函数一起使用。在 Excel 中，类似的功能是 VBA（Visual Basic for Applications）来编写自定义函数，例如用户可以编写一个自定义函数，然后在 Excel 中使用这个函数来对数据进行处理。

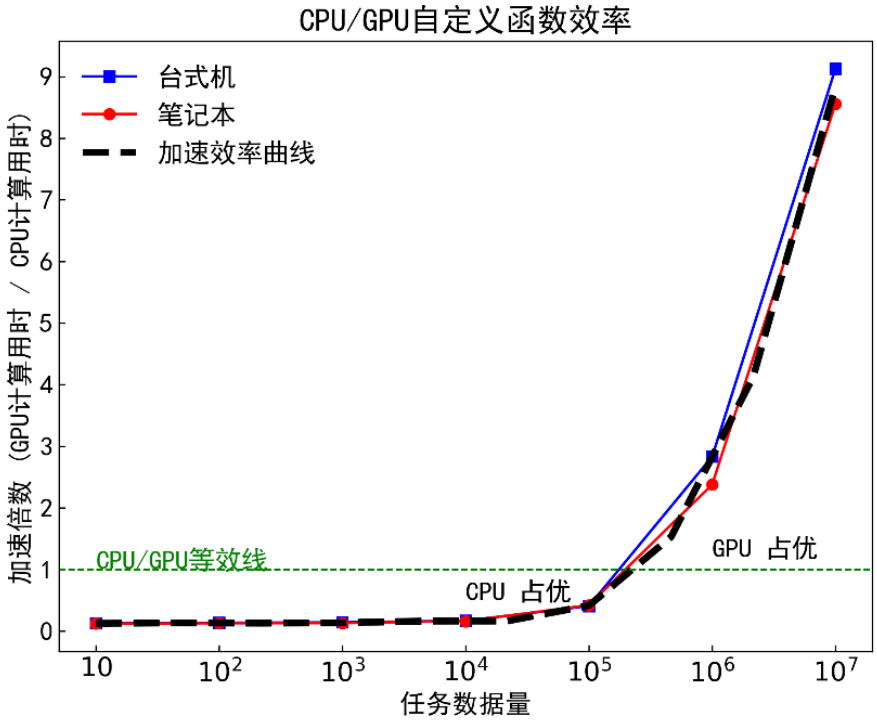
程序示例：

```
num_rows = 100000

pdf = pd.DataFrame()
pdf["key"] = np.random.randint(0, 5, num_rows)
pdf["val"] = np.random.randint(0, 7, num_rows)
def custom_formula_udf(df):
    df["out"] = df["key"] * df["val"] + df["val"]
    return df
gdf = cudf.from_pandas(pdf)
pandas_udf_groupby, cudf_udf_groupby = timeit_pandas_cudf(
    pdf,
    gdf,
    lambda df: df.groupby(["key"], group_keys=False).apply(custom_formula_udf),
    number=10,
)
```

我们生成了 num_rows 组数据，其中 key 为标签，val 为数值，希望通过程序计算不同 key 分组下每个数据经过运算后的输出值。

图表 27　CPU 和 GPU 运行用户自定义函数时的效率比较



资料来源：华创证券

结果如上图所示（图 27 所示），不论是台式机或者笔记本，当数据量在 100000 以下时，CPU 比 GPU 运算快，当数据量超过 1000000 后 GPU 运算快。随着数据量的增加，GPU 的表现比 CPU 表现更好。

四、总结

（一）CPU 和 GPU 并行计算的适用场景和选购方法

本报告分章节介绍了 CPU 和 GPU 的工作原理及其影响其计算效率的核心性能指标，用户应当选择适合自己任务的硬件。

CPU 的设计旨在处理通用计算任务，通常有更大的缓存和更高的时钟频率，适合于串行和低并行度任务。CPU 核心较少，但每个核心的性能较高，CPU 并行适合逻辑复杂性计算任务，如一般的计算任务、服务器应用、数据库管理等。

GPU 的设计旨在处理大规模并行计算任务，通常具有数百到数千个小型核心。这些核心被组织成多个处理单元，可同时执行大量相似的计算任务，GPU 并行适合重复性高计算性任务，如图形处理、深度学习、科学计算、密码学。

从软件层面，存在两种并行手段，分别是多进程和多线程。

多进程作用于软件层面，其适用于对于资源管理和保护要求高，不限制开销和效率的任务。因为多进程的每个进程互相独立，都会在内存中开辟计算空间，因此子进程崩溃并不会影响主进程的稳定，但是对内存的占用比较高。多进程的编程和调试相对于多线程也更加简单。对于有全局锁的编程语言，只能使用多进程。

多线程作用于硬件层面，其适用于需要频繁创建和销毁的任务场景，例如 Web 服务器连接、爬虫等。多线程占用内存少、切换简单、速度很快。但多线程中，子线程报错会影响全部线程。线程之间同步复杂，例如不同线程执行任务快慢不同、某些变量不能同时被两个线程修改。

影响 CPU 性能的要素包括处理器频率、核心数、缓存、指令集、流水线、超线程、内存带宽和架构设计等。其中影响并行计算重要的是核心数和超线程支持。同等条件下，CPU 核数越多，支持的线程越多，主频越高则并行效能越好。

GPU 的算力、CUDA 核心数量、GPU 内存带宽、CUDA 核心时钟频率、内存层次结构、数据并行性、算法设计和数据传输与内存访问优化等因素都会影响 GPU 的并行计算性能。且消费型 GPU 和专业计算 GPU 面向任务不同。建议用户按需选取，将算力作为重要考量对象，如果是大模型训练，也建议将 GPU 耗电功率纳入考量。

（二）Python 常用加速程序

承接上文所述的多进程和多线程并行计算，这部分介绍了常见的 python 执行多进程库，简要介绍了 python 多线程库。同时，也介绍了部分 C 语言并行的内容，由于 C 语言的并行和 CUDA 语言的直接使用者较少，因此不展开叙述。

常用 python 利用 CPU 并行的多进程库有 multiprocessing，加速 python pandas 的库有 pandarallel、joblib。测试结果表明，多进程计算效率随着进程数量增加，存在“快速提高计算效率”、“效率不再提升”和“效率逐渐下降”三个阶段。其原因在于，多进程数量较少时，计算机的内存和 I/O 资源是充分的，多进程会迅速提升计算效率，随着进程数量的增加，计算机的线程资源会互相竞争导致计算效率下降。用户应该根据 CPU 的性能选择合适的多线程数量。

常用的 python 调用 GPU 库有 pycuda、numba 和 RAPIDS。我们在两款硬件上测试了 GPU 对常见 Dataframe 操作的加速效果，测试结果表明对大量数据的合并、聚合统计和用户自

定义函数 GPU 的加速效果明显。处理的数据量在 100000 以下时，直接使用 CPU 是更好的选择。

五、风险提示

本报告中所有统计结果和模型方法均基于历史数据，不代表未来趋势。不同用户硬件差异对加速效果有影响。

金融工程组团队介绍

组长、首席分析师：王小川

同济大学管理学博士。2017 年加入华创证券研究所。

高级分析师：秦玄晋

上海对外经贸大学硕士。2018 年加入华创证券研究所。

助理研究员：杨宸祎

美国伊利诺伊大学香槟分校会计学、金融工程学硕士，CFA。2021 年加入华创证券研究所。

助理研究员：黄河

东华大学工学博士，FRM。2023 年加入华创证券研究所。

助理研究员：洪远

华南理工大学金融硕士。2023 年加入华创证券研究所。

华创证券机构销售通讯录

地区	姓名	职务	办公电话	企业邮箱
北京机构销售部	张昱洁	副总经理、北京机构销售总监	010-63214682	zhangyujie@hcyjs.com
	张菲菲	北京机构副总监	010-63214682	zhangfeifei@hcyjs.com
	刘懿	副总监	010-63214682	liuyi@hcyjs.com
	侯春钰	资深销售经理	010-63214682	houchunyu@hcyjs.com
	过云龙	高级销售经理	010-63214682	guoyunlong@hcyjs.com
	蔡依林	资深销售经理	010-66500808	caiyilin@hcyjs.com
	刘颖	资深销售经理	010-66500821	liuying5@hcyjs.com
	顾翎蓝	资深销售经理	010-63214682	gulinglan@hcyjs.com
	车一哲	销售经理		cheyizhe@hcyjs.com
深圳机构销售部	张娟	副总经理、深圳机构销售总监	0755-82828570	zhangjuan@hcyjs.com
	汪丽燕	高级销售经理	0755-83715428	wangliyan@hcyjs.com
	张嘉慧	高级销售经理	0755-82756804	zhangjiahui1@hcyjs.com
	董姝彤	销售经理	0755-82871425	dongshutong@hcyjs.com
	王春丽	高级销售经理	0755-82871425	wangchunli@hcyjs.com
上海机构销售部	许彩霞	总经理助理、上海机构销售总监	021-20572536	xucaixia@hcyjs.com
	官逸超	上海机构销售副总监	021-20572555	guanyichao@hcyjs.com
	黄畅	上海机构销售副总监	021-20572257-2552	huangchang@hcyjs.com
	吴俊	资深销售经理	021-20572506	wujun1@hcyjs.com
	张佳妮	资深销售经理	021-20572585	zhangjiani@hcyjs.com
	蒋瑜	高级销售经理	021-20572509	jiangyu@hcyjs.com
	施嘉玮	高级销售经理	021-20572548	shijiawei@hcyjs.com
	朱涨雨	高级销售经理	021-20572573	zhuzhangyu@hcyjs.com
	李凯月	高级销售经理		likaiyue@hcyjs.com
	易星	销售经理		yixing@hcyjs.com
	张玉恒	销售经理		zhangyuheng@hcyjs.com
广州机构销售部	段佳音	广州机构销售总监	0755-82756805	duanjiayin@hcyjs.com
	周玮	销售经理		zhouwei@hcyjs.com
	王世韬	销售经理		wangshitao1@hcyjs.com
私募销售组	潘亚琪	总监	021-20572559	panyaqi@hcyjs.com
	汪子阳	副总监	021-20572559	wangziyang@hcyjs.com
	江赛专	副总监	0755-82756805	jiangsaizhuan@hcyjs.com
	汪戈	高级销售经理	021-20572559	wangge@hcyjs.com
	宋丹玓	销售经理	021-25072549	songdanyu@hcyjs.com

华创行业公司投资评级体系

基准指数说明：

A 股市场基准为沪深 300 指数，香港市场基准为恒生指数，美国市场基准为标普 500/纳斯达克指数。

公司投资评级说明：

强推：预期未来 6 个月内超越基准指数 20%以上；
推荐：预期未来 6 个月内超越基准指数 10%－20%；
中性：预期未来 6 个月内相对基准指数变动幅度在-10%－10%之间；
回避：预期未来 6 个月内相对基准指数跌幅在 10%－20%之间。

行业投资评级说明：

推荐：预期未来 3-6 个月内该行业指数涨幅超过基准指数 5%以上；
中性：预期未来 3-6 个月内该行业指数变动幅度相对基准指数-5%－5%；
回避：预期未来 3-6 个月内该行业指数跌幅超过基准指数 5%以上。

分析师声明

每位负责撰写本研究报告全部或部分内容的分析师在此作以下声明：

分析师在本报告中对所提及的证券或发行人发表的任何建议和观点均准确地反映了其个人对该证券或发行人的看法和判断；分析师对任何其他券商发布的所有可能存在雷同的研究报告不负有任何直接或者间接的可能责任。

免责声明

华创证券研究所

北京总部	广深分部	上海分部
地址：北京市西城区锦什坊街 26 号恒奥中心 C 座 3A	地址：深圳市福田区香梅路 1061 号 中投国际商务中心 A 座 19 楼	地址：上海市浦东新区花园石桥路 33 号花旗大厦 12 层
邮编：100033	邮编：518034	邮编：200120
传真：010-66500801	传真：0755-82027731	传真：021-20572500
会议室：010-66500900	会议室：0755-82828562	会议室：021-20572522