南開大學
Nankai University

计算机学院

并行程序设计实验报告

OpenMP 和 Pthread 编程

姓名：王旭尧黄天昊

学号：2012527 2011763

专业：计算机科学与技术

2023 年 5 月 14 日

目录

1 问题描述

我们实验的总体计划为手动实现 transformer 框架，并实现并行化训练。截至第四次实验，我们初步用 C++ 实现了能够自动求导的 Var 框架，并尝试使用 pthread 和 openmp 进行加速。考虑到我们的最初拟定的项目主要工作在编程框架，并行化只是一种优化手段，因此我们考虑将普通高斯消元部分的实验也进行完成，作为学习 pthread 和 openmp 的过程，以期来优化我们的项目。我们项目在并行上的工作点主要在于每个算子都需要进行并行，虽然单个算子并行代码不多，但每个算子都需要进行并行计算。

2 分工情况

本次实验由王旭尧（2012527）和黄天昊（2011763）共同完成。我们共同尝试写自己的自动求导框架，各自对不同算子进行编写并并行化。对于普通高斯消元，我们也共同尝试进行了实验。

3 框架介绍及并行点分析

3.1 框架介绍

3.1.1 Tensor 部分

对于 Tensor，我们认为应该具有以下特性：

- 能够有自定义的多维度，而非固定的 2 维、3 维
- 能够像数据类型一样能够使用运算符，也支持自定义算子
- 能够用下标获取数值、提供下标对某个数据进行数值修改

因此我们定义了自己的 Tensor，代码如下：

```
class Tensor {

    double& operator[](int i);
    const double& operator[](int i) const;

    // 重载运算符 (), 用于获取或设置 Tensor 对象中的元素值
    double& operator()(std::vector<int> index);
    const double& operator()(std::vector<int> index) const;
    Tensor operator-() const;
    Tensor operator+(const Tensor& other) const;
    Tensor operator-(const Tensor& other) const;
    Tensor operator+(double scalar) const;
    Tensor operator-(double scalar) const;
    Tensor operator+(int scalar) const;
    Tensor operator-(int scalar) const;
```

```
16      Tensor operator*(const Tensor& other) const;
17      Tensor operator/(const Tensor& other) const;
18      Tensor operator*(double scalar) const;
19      Tensor operator/(double scalar) const;
20      Tensor operator*(int scalar) const;
21      Tensor operator/(int scalar) const;
22      bool operator==(const Tensor& other) const;
23      bool operator!=(const Tensor& other) const;
24      Tensor& operator=(const Tensor& other);
25
26      friend Tensor matmul(const Tensor& A, const Tensor& B); // 二维 Tensor 矩阵乘法
27      friend Tensor mul(const Tensor& A, const Tensor& B) ;   // Tensor 对应元素相乘
28      friend double dot(const Tensor& a, const Tensor& b);    // 向量点乘
29
30      Tensor T() const;   //矩阵转置
31      void print() const;
32      void reset();
33
34  private:
35      std::vector<int> shape_;    // Tensor 对象的形状
36      std::vector<double> data_;  // Tensor 对象的数据
37      int size_;                  // Tensor 对象的大小
38  private:
39      static bool is_valid_mm(const Tensor& A, const Tensor& B);
40      void print(int dim, std::vector<int> indices) const;
41      int indexToOffset(std::vector<int> index) const;        // 将索引转换为偏移量
42  };
```

对于三个诉求，我们对应的实现方式如下：

- 通过数组 shape_ 来存储不同维度的大小，通过一维数组 data_ 来存储数据。比如 shape_ 为 2，3，1，表示这个 Tensor 是一个 $2 \times 3 \times 1$ 的三维 Tensor，总计有 6 个数据。
- 为了使用运算符，我们采用重载运算符的方式。只是能重载的运算符是有限的，对于四则运算之外的算子，我们采用友元函数的方式来自定义算子，例如矩阵乘法或是卷积。
- 通过重载 [] 运算符，以及对 index 的映射，实现通过下标对值的获取或是赋值。

3.1.2 Var 部分

对于计算图（computing graph）3.1

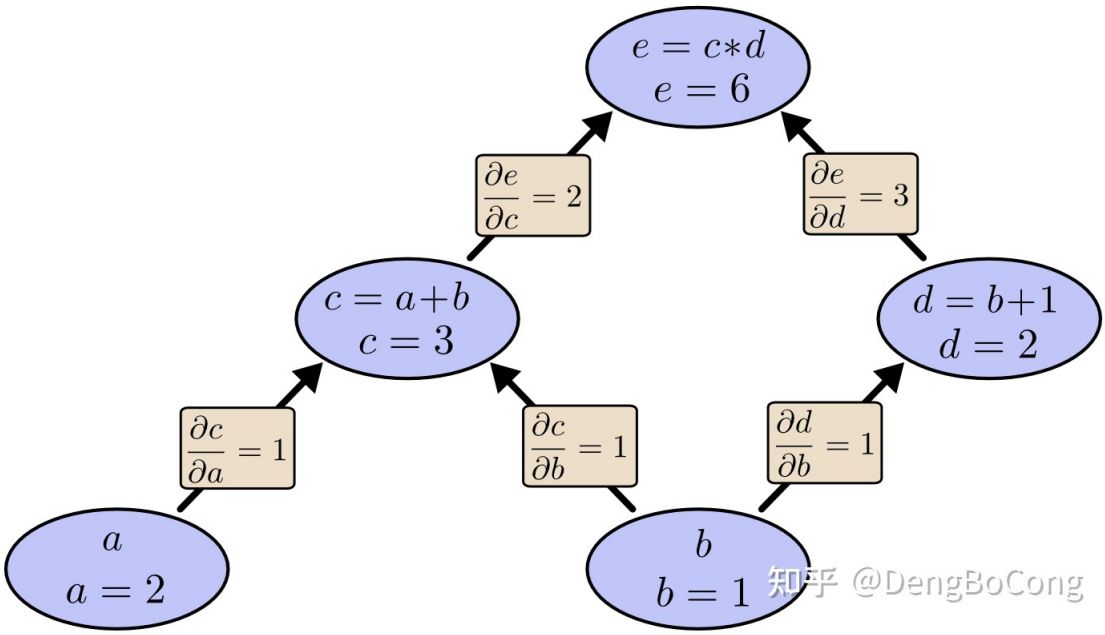


图 3.1: computing graph

可以注意到计算图本质上是一个拓扑图，且入度为 0 的节点为变量/常量，其余节点为父节点的运算结果以及运算符。因此可以将计算图的每个节点抽象为 < 计算值，算子 > 组成的二元组。因此我们可以如下地定义计算图

```
class Var{
protected:
    struct DataInfo{
        Operator* op= nullptr;
        Tensor data=Tensor();
        std::vector<Var*> predecessorNodes;
    }dataInfo;
};
```

为了引入常量/变量的概念，我们将 Var 定义为纯虚类，其下有两个子类：AutoDifferentialVar 和 NonAutoDifferentialVar，对于两个子类，定义运算规则如下：

- AutoDifferentialVar 与 AutoDifferentialVar 运算结果为 AutoDifferentialVar
- AutoDifferentialVar 与 NonAutoDifferentialVar 运算结果为 AutoDifferentialVar
- NonAutoDifferentialVar 与 AutoDifferentialVar 运算结果为 AutoDifferentialVar
- NonAutoDifferentialVar 与 NonAutoDifferentialVar 运算结果为 NonAutoDifferentialVar

如此定义是为了在反向传播时，遇到常数时可以进行剪枝操作，不用再继续传播，以此来避免无意义的计算。此外为了便于统一计算，我们定义了接口 GradientInterface，将需要反向求导相关的函数统一到这个接口中，便于计算。相关的部分代码如下：

```
class GradientInterface{
public:
    virtual void backward(const Tensor& gradient)=0;
    virtual void backward()=0;
    virtual bool isAutoDifferentiable()=0;
protected:
    double learnable;
};
```

```
class AutoDifferentialVar: public Var, public GradientInterface{
public:
    void backward(const Tensor& gradient) override;
    void backward() override;
    bool operator==(const Var& other) const override;
    void print() override;
    bool isAutoDifferentiable() override;
};

class AutoDifferentialVar: public Var, public GradientInterface{
public:
    void backward(const Tensor& gradient) override;
    void backward() override;
    bool operator==(const Var& other) const override;
    void print() override;
    bool isAutoDifferentiable() override;
};
```

对于反向传播算法，本质上是一个拓扑图上的搜索算法，从最终节点开始不断向祖先节点传播当前节点相对于损失函数的梯度。例如对于以下情况:

$$L = f(z)z = x + y$$

在节点 z 从后继节点得到梯度 $\frac{\partial L}{\partial z}$，通过算子 AddOperator 得到 z 相对于两个变量 x,y 对应的梯度 $\{\frac{\partial z}{\partial x}, \frac{\partial z}{\partial y}\}$，再将梯度进行累积，将 $\{\frac{\partial L}{\partial z}\frac{\partial z}{\partial x}, \frac{\partial L}{\partial z}\frac{\partial z}{\partial y}\}$ 向下传递。

- 如果 x 和 y 都是变量，那么，$\frac{\partial z}{\partial x}$ 和 $\frac{\partial z}{\partial y}$ 都是 1, 那么通过 AddOperator 得到 $\{\frac{\partial L}{\partial z}, \frac{\partial L}{\partial z}\}$
- 如果 x 是常数，y 是变量，那么，$\frac{\partial z}{\partial x}$ 和 $\frac{\partial z}{\partial y}$ 是 $0, 1$, 那么通过 AddOperator 得到 $\{0, \frac{\partial L}{\partial z}\}$
- 如果 x 是变量，y 是常数，那么，$\frac{\partial z}{\partial x}$ 和 $\frac{\partial z}{\partial y}$ 是 $1, 0$, 那么通过 AddOperator 得到 $\{\frac{\partial L}{\partial z}, 0\}$

对于反向传播，本质上是通过递归在计算图上不断搜索，直到找到最基本的变量。核心代码如下:

```
void AutoDifferentialVar::backward(const Tensor& gradient){
    double learning_rate=0.001;
    if (dataInfo.op== nullptr){
        dataInfo.data=dataInfo.data-gradient*learning_rate;
        return;
    }
    std::vector<Tensor>grads=dataInfo.op->backward(
            gradient,
            std::vector<Tensor>{
                dataInfo.predecessorNodes[0]->getData(),
                dataInfo.predecessorNodes[1]->getData()});
    for (int i = 0; i < dataInfo.predecessorNodes.size(); ++i) {
        dynamic_cast<GradientInterface*>(dataInfo.predecessorNodes[i])->backward(grads[i]);
    }
}
```

其中，dataInfo 是当前变量的信息，包含了变量的值、算子种类和前驱节点等信息；dataInfo.op 是当前变量所对应的操作符号，可以通过它来计算当前变量的梯度；dataInfo.predecessorNodes 是当前变量的前驱节点，即当前变量参与计算的输入变量。在计算当前变量的梯度时，需要将当前变量的梯度向前传递给前驱节点进行计算。

4 并行点分析

4.1 Tensor 基础运算并行优化

对于 Tensor 内的运算可以使用并行来优化的基本上就说 Tensor 的对位计算。比如对位加法、对位乘法，或者与标量进行运算等。比如下面的代码：

```
 重载 Tensor 乘法运算符，返回 Tensor 与 int 相乘的结果
nsor Tensor::operator*(int scalar) const {
  Tensor result(shape_);
  for (int i = 0; i < data_.size(); i++) {
      result.data_[i] = data_[i] * scalar;
  }
  return result;

```

这部分是 Tensor 与标量进行乘法的运算，很明显可以针对该部分进行优化，且由于没有明显的数据依赖问题，该部分的优化较为简单直观。

4.2 Var 反向传播并行优化

在反向传播的过程中，可以注意到

```
    for (int i = 0; i < dataInfo.predecessorNodes.size(); ++i) {
        dynamic_cast<GradientInterface*>(dataInfo.predecessorNodes[i])->backward(grads[i]);
    }
```

是一个循环，对于这部分循环可以尝试进行并行优化。

5 pthread

5.1 在普通高斯消元上学习 pthread 编程

5.1.1 正确性验证

为避免错误的测试用例导致计算结果是 Naf 或无穷，使用如下代码生成测试用例：

```

void ReSet(){
for(int i = 0;i < maxN; i++){
    for(int j = 0;j < i; j++)
        A[i][j] = 0;
    A[i][i] = 1.0;
    for(int j = i + 1;j < maxN; j++)
        A[i][j] = rand() % 100;
}
for(int i = 0;i < maxN; i++){
    int k1 = rand() % maxN;
    int k2 = rand() % maxN;
    for(int j = 0;j < maxN; j++){
        A[i][j] += A[0][j];
        A[k1][j] += A[k2][j];
    }
}

```

该代码首先初始化一个上三角矩阵，并将行列线性组合，打印结果如图5.2。可以看见，结果确实为上三角矩阵，且未出现 nan 和 inf。

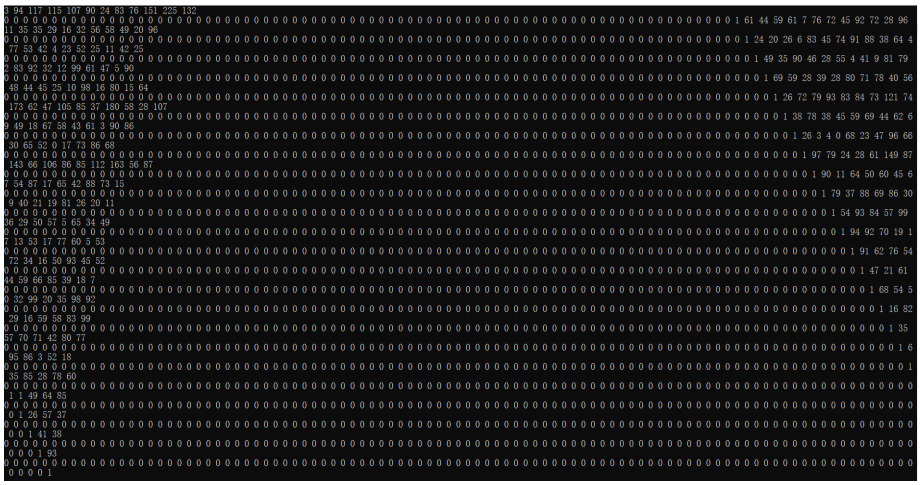


图 5.2: 普通高斯-正确性验证

5.1.2 复杂度分析

容易得到，串行版本的代码复杂度为 $O(N^3)$。pthread 版本的代码作用于除法和消去过程，以 NUM_THREADS=7，除法为多线程、消去为行划分为例，每次外层循环时，里面的内容被 7 等分(**循环划分而非块划分，故负载均衡**)。记除法最慢线程执行时间为 $T1$，消去最慢线程执行时间为 $T2$，则线程函数执行时间

$$T = N(T1 + T2) = (N^2 + N^3)/7 = O(N^3/7)$$

其中 $T1 = (TimeforDivision)/7 = O(N/7)$，$T2 = (TimeforElimination)/7 = (O(N^2)/7)$。记线程和 barrier 创建的时间为 t_start，销毁的时间为 t_end，初始化时间为 $t\_init = O(N)$，则程序总运行时间为

$$T = t\_init + t\_start + t\_end + (T1 + T2) = O(N) + t\_start + t\_end + O(N^3/7)$$

，加速比为 7。若加入 SIMD，则消去部分可优化为 $T2 = O(N^2/28)$，程序总复杂度为

$$T = t\_init + t\_start + t\_end + (T1 + T2) = O(N) + t\_start + t\_end + O(N^3/28)$$

5.1.3 串行和 pthread

分别在规模 500、1000、2000 下，对串行、pthread、pthread+Neon 三个版本的代码，在鲲鹏服务器提供的 ARM 平台进行测试，得到测试结果为图6.15。可以看到，无论在哪个规模下，运行时间的大小关系都是串行 >pthread。在规模较小 (500) 时，加速比为 4.66，规模 1000 时加速比为 6.01，规模 2000 时加速比为 4.96。

	clockticks	instructions retired	CPI	L1_HIT	L1_MISS
serial	21,465,600,000	25,979,200,000	0.826	6,696,010,044	3,100,093
pthread	34,020,800,000	112,073,600,000	0.304	33,046,049,569	37,201,116

表 1: 普通高斯-Vtune 分析指标

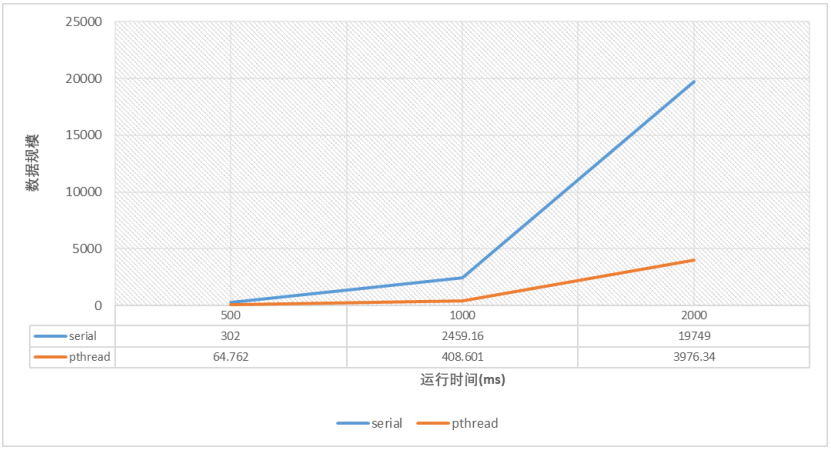
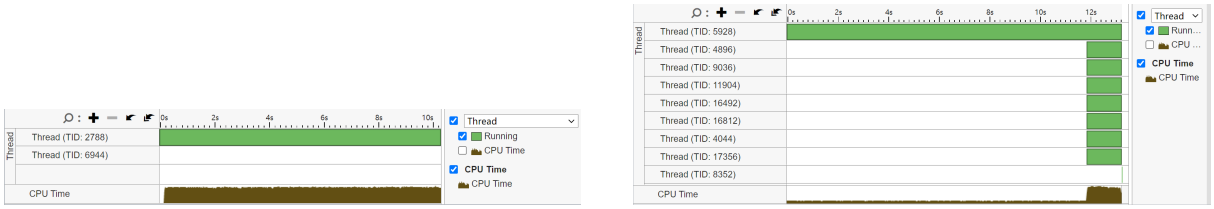


图 5.3: 普通高斯-串行/pthread/pthread+SIMD

使用 Vtune 对 x86 平台上的串行和 pthread 代码进行性能分析，可以看到串行和 pthread 的图像4(b)，图4(b)第一行为主线程，负责初始化和创建、销毁线程，因此出现在最开头和末尾；后面 7 行为 7 个负责高斯消去的工作线程，可以看到，由于任务划分时使用循环划分而非块划分，因此负载均衡，各工作线程基本同时开始、同时结束。此外，根据 Vtune 显示的各性能指标，总结如表3。可以看



(a) 串行 (b) pthread

图 5.4: 普通高斯-vtune 分析图

到，pthread 算法的 CPI 是串行算法的 36%，且 L1_HIT 远高于串行算法，显著提高了性能。

5.1.4 各种 SIMD+pthread

结合上节实验 SIMD 的知识，在 pthread 编程中加入 SIMD 并测量运行时间差别，结果如图6.17。可以看到，效率排序为 AVX>SSE>Neon> 未加入 SIMD，且性能梯度为：无 SIMD->Neon(30-46%)，Neon->SSE(22-40%)，SSE->AVX(21-38%)，**选用何种 SIMD 指令有一定影响，但 SIMD 的加入与否对性能影响最大**。

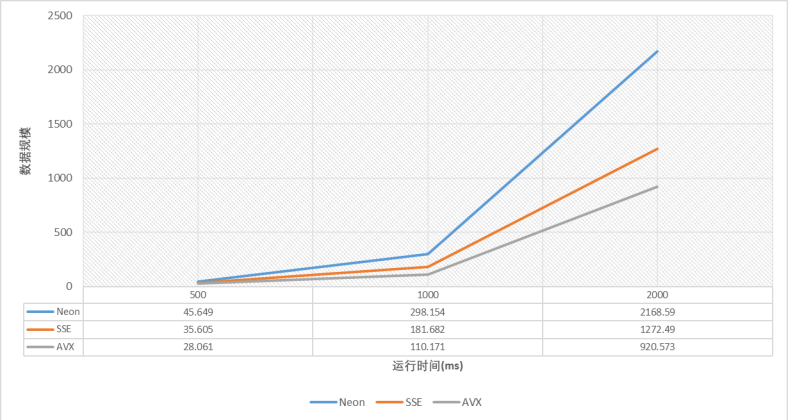


图 5.5: 普通高斯-不同 SIMD 指令

5.1.5 arm 和 x86

分别在规模 500、1000、2000 下，对 ARM 平台和 X86 平台的 pthread、pthread+SIMD 共 15 个版本的代码，在鲲鹏服务器提供的 ARM 平台和本地 X86 平台进行测试，得到测试结果为图6.18。

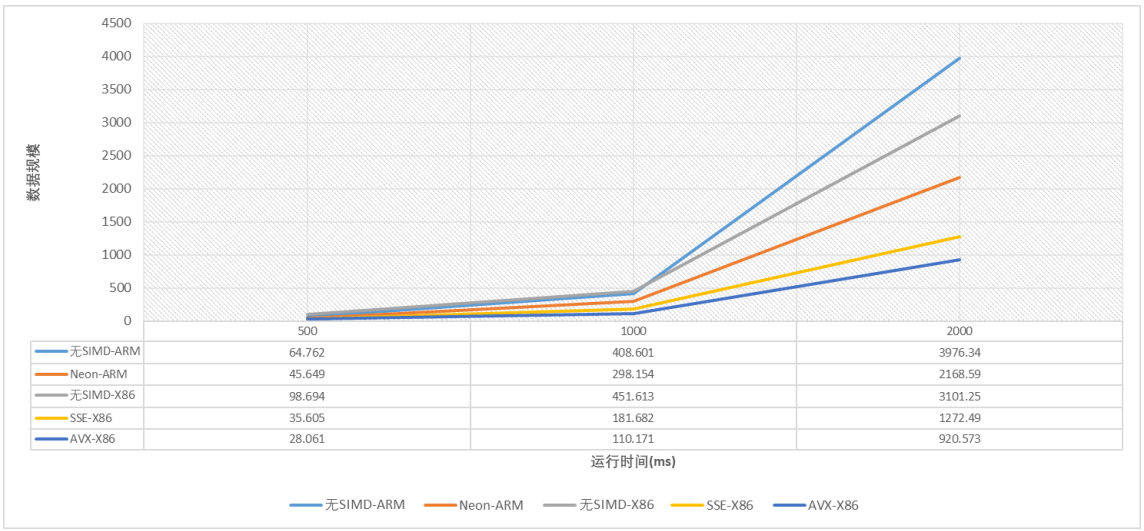


图 5.6: 普通高斯-不同平台

可以看到，总体而言 X86 表现优于 ARM，SIMD 代码的性能主要由不同 SIMD 语言决定，AVX 优于 SSE 优于 Neon。

5.1.6 不同优化力度

在 ARM 平台、2000 数据规模选择编译器的不同优化力度选项进行测试，得到数据如图6.20所示。可以看到，编译器的 O1 优化得到了质的飞跃，减少约 75%的运行时间，此后的 O2、O3 优化只有量的改变。

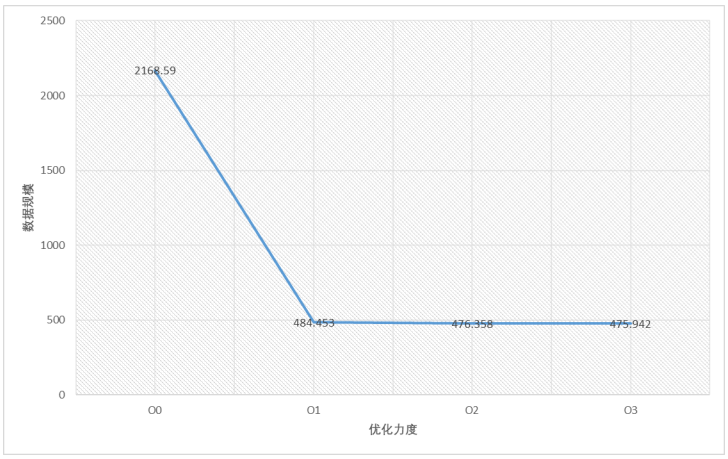
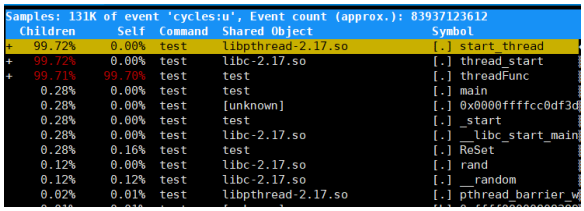
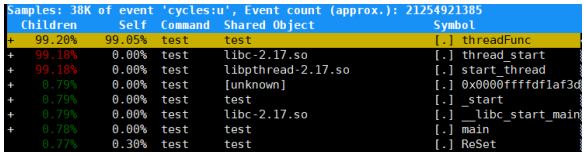


图 5.7: 普通高斯-不同优化力度

对 O0 和 O1 的代码进行 perf 分析，由图21(a)和图21(b)可看到 O0 版本为 131K，O1 版本为 38K。进入反汇编分析，发现 O1 主要优化了 start_thread 函数。在 O0 中多次使用了 str 指令，str 指令表示从源寄存器将数据写入目的存储器中，而向指定地址写入数据在各汇编指令中速度较慢，会在 CPU 的流水线中产生多个 bubble，因此耗时占比最大。而 O1 代码中针对 str 指令进行了优化，将移位加入 str 指令，如图22(a)和图22(b)所示。同样，编译器对占比第 2 的 ldr 指令进行了类似的优化。

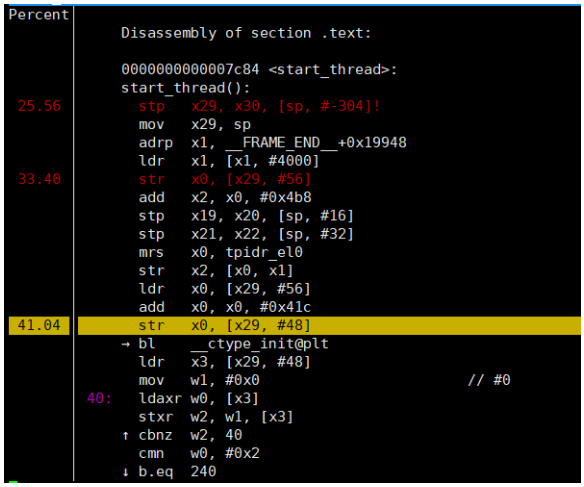


(a) O0

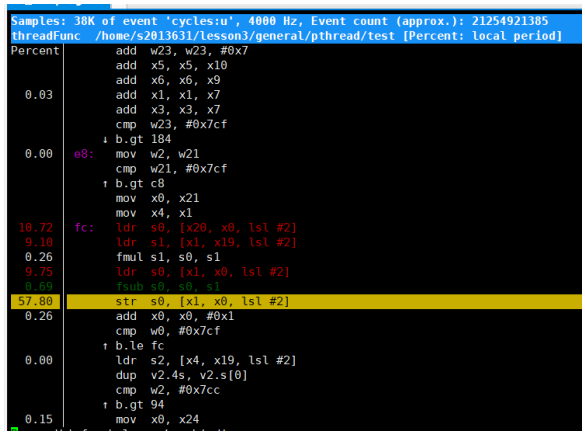


(b) O1

图 5.8: 普通高斯-不同优化力度-perf



(a) O0



(b) O1

图 5.9: 普通高斯-不同优化力度-perf

5.1.7 不同线程数

在 2000 规模下，对 7、5、3、1 线程数版本和串行版本对比，如图6.23所示。可以看到，线程数越多运行时间越短，且在线程数较大时变化率较小，在线程数较小时变化率较大，与特殊高斯处的不同线程数结果类似。

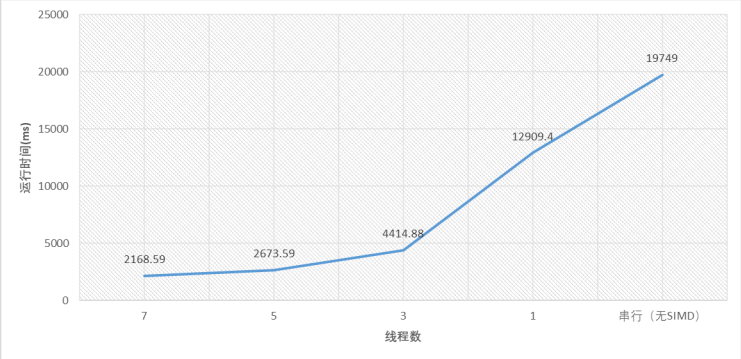


图 5.10: 普通高斯-不同线程数

5.1.8 不同算法优化

- 除法：在 2000 数据规模下比较仅 t_id==0 做除法和所有工作线程做除法的情况，如图5.11所示。可以看到，虽然仅改变了除法的线程数，但节约了 13%的时间。

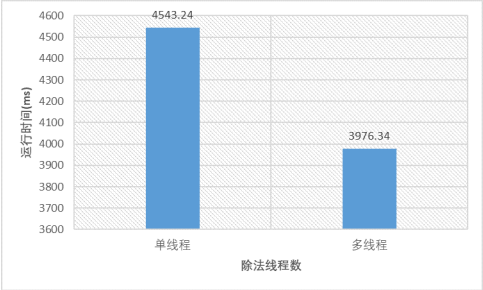


图 5.11: 普通高斯-除法线程

- 消去：设计不同的任务划分方式，对消去部分分别采用垂直划分 (列划分) 和水平划分 (行划分)，结果如图6.27所示。

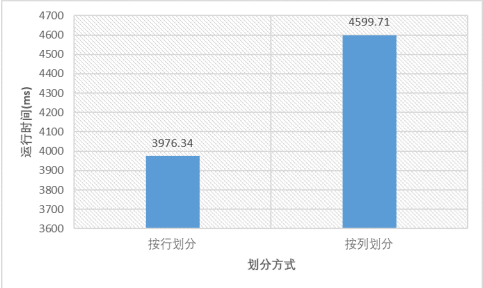


图 5.12: 普通高斯-不同划分

需要注意的是:

1. 为保持同步和正确性，需要修改代码。**由于列划分的消去代码中各线程不一定同时结束外层循环，所以不能在外层循环内就修改 A[i][k] 为 0.0，而应在外层循环结束后，指定某线程将 A[i][k] 置 0**。没有在外层循环结束时置 0，会做出列划分反而更快的结果，因为 A[i][k]==0 时运算比 A[i][k] 为非 0 浮点数时更快。代码如下:

```
//we divide the procedure by column.
    for(int i = k + 1;i < n;i ++){
        //消去
        for(int j = k + 1 + t_id;j < n; j += NUM_THREADS){
            A[i][j] = A[i][j] - A[i][k] * A[k][j];
        }

    }
    //第二个同步点
    pthread_barrier_wait(&barrier_Elimination);
    if(t_id == 0)
        for(int i = k + 1;i < n; i++)
             A[i][k] = 0.0;
}
```

2. cache 利用上，由于行划分较列划分能更好利用**空间局部性**，因此行划分的运行时间会更短。

- 对齐: 使用 2000 数据规模、Neon 的对齐/不对齐代码在 ARM 平台测试，得到结果如图6.28所示。由于 Neon 的对齐指令 CPI 更小，因此对齐版本的运行时间更少。

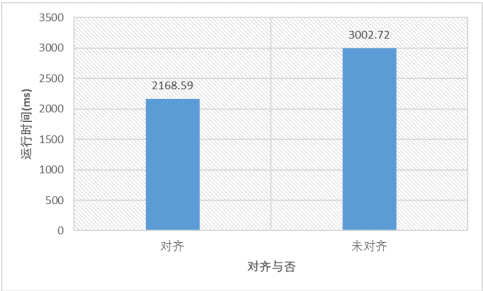


图 5.13: 普通高斯-对齐与否

5.1.9 不同 pthread 算法

对指导书的动态线程、静态线程 + 信号量、静态线程 +barrier 同步 3 个版本的代码，在 500 数据规模、ARM 平台下测试 (**选择 500 数据规模是因为这样更能体现动态线程创建和销毁的开销比例**)，结果如图5.14所示。可以看到，动态线程最慢，原因是在主循环中不断新建线程。我尝试**测量了线程的**

创建时间，经多次测量得到平均时间为 0.341ms，当运行 n 次 (最外层循环次数) 时，若 n==2000 则额外耗时 682ms，符合运行结果。

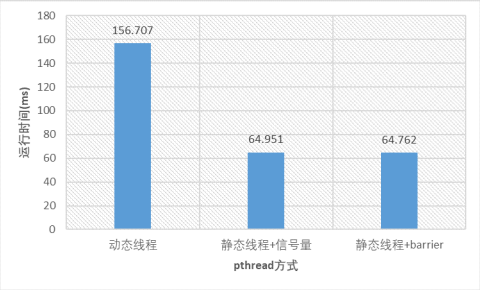


图 5.14: 普通高斯-不同 pthread 方式

动态线程和静态线程的主要区别是：动态线程在每次需要调用线程函数时都新建线程，相当于 new 和 delete，这样虽然不会占用系统太多资源，但由于线程创建开销本身较大，多次调用本身会造成过多额外开销；而静态线程的步骤为 (i) 主线程做除法，(ii) 主线程唤醒工作线程，(iii) 工作线程被唤醒后做消去，(iv) 工作线程完成后通知主线程，工作线程睡眠，(v) 所有线程进入下一轮，循环往复。

5.2 Tensor 上进行 Pthread 并行

该部分的并行主要是对基本运算进行并行，如

```
// 重载运算符 *
Tensor Tensor::operator*(const Tensor& other) const {
    // 检查两个 Tensor 对象的形状是否相同
    if (shape_ != other.shape_) {
        throw std::runtime_error("Cannot add two tensors with different shapes.");
    }
    // 创建一个新的 Tensor 对象
    Tensor result(shape_);
    // 对两个 Tensor 对象中的每一个元素执行乘法运算
    auto add_func = [&](int start, int end) {
        for (int i = start; i < end; i++) {
            result.data_[i] = data_[i] * other.data_[i];
        }
    };
    std::vector<std::thread> threads;
    int num_threads = std::thread::hardware_concurrency();
    int chunk_size = result.data_.size() / num_threads;
    for (int i = 0; i < num_threads; i++) {
        int start = i * chunk_size;
        int end = (i == num_threads - 1) ? result.data_.size() : (i + 1) * chunk_size;
        threads.emplace_back(add_func, start, end);
```

```
    }
    for (auto& thread : threads) {
        thread.join();
    }
    return result;
}
```

5.3 Var 上进行 Pthread 并行

```
void AutoDifferentialVar::backward(const Tensor& gradient){
    double learning_rate=0.001;

    if (dataInfo.op== nullptr){
        dataInfo.data=dataInfo.data-gradient*learning_rate;
        return;
    }
    std::vector<Tensor>grads=dataInfo.op->backward(gradient,
                                                   std::vector<Tensor>{dataInfo.predecessorNod
                                                                       dataInfo.predecessorNod
    pthread_t threads[dataInfo.predecessorNodes.size()];
    ThreadData threadData[dataInfo.predecessorNodes.size()];

    for (int i = 0; i < dataInfo.predecessorNodes.size(); ++i) {
        threadData[i].index = i;
        threadData[i].grad = grads[i];
        pthread_create(&threads[i], NULL, &AutoDifferentialVar::backwardThread, (void*)&thread
    }

    for (int i = 0; i < dataInfo.predecessorNodes.size(); ++i) {
        pthread_join(threads[i], NULL);
    }
}

static void* AutoDifferentialVar::backwardThread(void* arg) {
    ThreadData* threadData = (ThreadData*)arg;
    int index = threadData->index;
    Tensor grad = threadData->grad;

    dynamic_cast<GradientInterface*>(dataInfo.predecessorNodes[index])->backward(grad);

    pthread_exit(NULL);
```

```
33  }
34
```

6 OpenMP

6.1 在普通高斯消元上学习 OpenMP 编程

6.1.1 串行和 OpenMP

分别在规模 500、1000、2000 下，对串行、OpenMP、OpenMP+Neon 三个版本的代码，在鲲鹏服务器提供的 ARM 平台进行测试，得到测试结果为图6.15。可以看到，无论在哪个规模下，运行时间的大小关系都是串行 >OpenMP，加速比为 8-10。

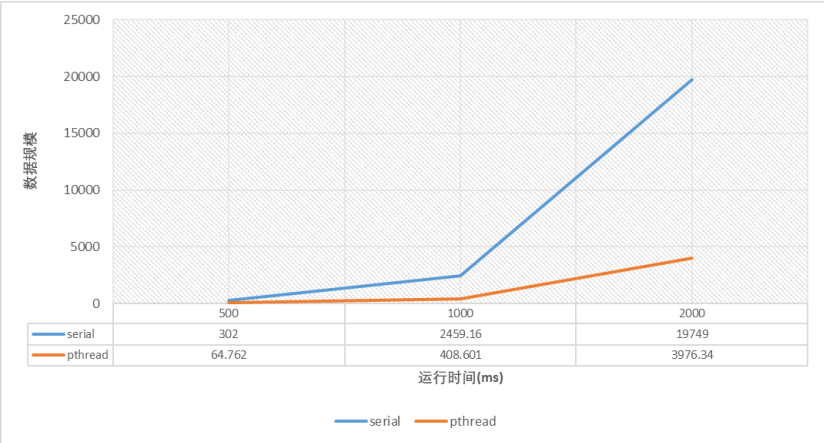
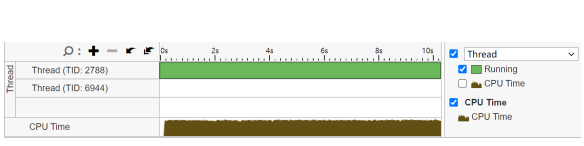
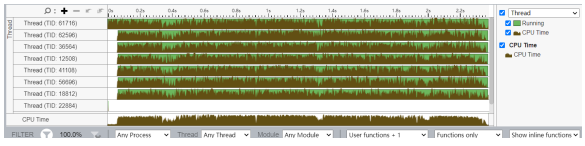


图 6.15: 普通高斯-串行/OpenMP/OpenMP+SIMD

使用 Vtune 对 x86 平台上的串行和 omp 代码进行性能分析，可以看到串行和 omp 的图像4(b)，图16(b)第一行为主线程，负责初始化和创建、销毁线程，并在中间参与工作线程的运算，因此一直从最开始工作到最末；后面 6 行为 6 个仅负责高斯消去的工作线程。我们对串行和 omp 的 clockticks、



(a) 串行



(b) omp

图 6.16: 普通高斯-vtune 分析图

instructions retired 和 CPI 列出如下比较表格2：

	clockticks	instructions retired	CPI
serial	21,465,600,000	25,979,200,000	0.826
OpenMP	6,867,500,000	12,630,000,000	0.53386

表 2

可以看到，omp 的始终周期数、平均执行的指令数和 CPI 均较小，故性能较高。

6.1.2 各种 SIMD+OpenMP

结合上节实验 SIMD 的知识，在 OpenMP 编程中加入 Neon、SSE、AVX 三种 SIMD 并测量运行时间，结果如图6.17。可以看到，效率排序为 AVX+OpenMP SSE+OpenMP>Neon+OpenMP> 无 SIMD 的 OpenMP> 无 OpenMP 和 SIMD 的串行，且性能梯度为：串行->OpenMP(90%)，无 SIMD->Neon(6.7%)，Neon->SSE(2.7%)。这说明了 2 点：第一，**OpenMP 的优化力度远大于 SIMD**；第二，**选用何种 SIMD 指令有一定影响，但 SIMD 的加入与否对性能影响最大**。

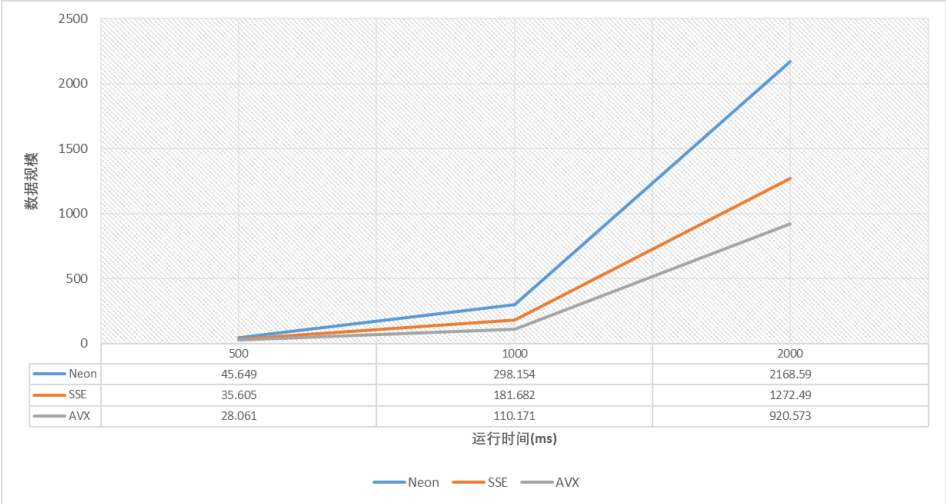


图 6.17: 普通高斯-不同 SIMD 指令

6.1.3 arm 和 x86

在规模 2000 下，对 ARM 平台和 X86 平台的 OpenMP、OpenMP+SIMD 的代码，在鲲鹏服务器提供的 ARM 平台和 Intel 提供的 Devcloud 平台进行测试，得到测试结果为图6.18。

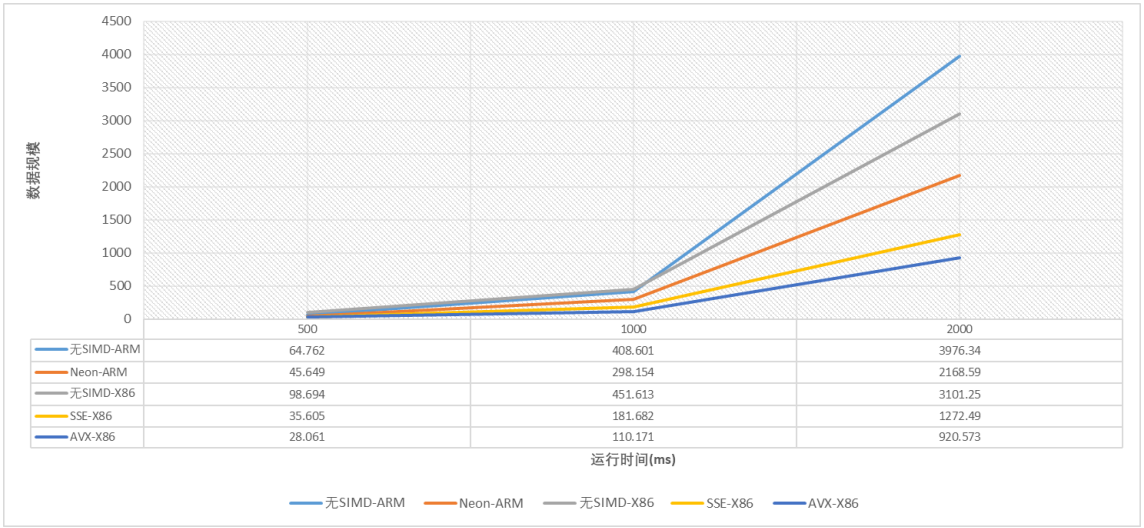


图 6.18: 普通高斯-ARM 和 X86

可以看到，总体而言 X86 表现优于 ARM。

6.1.4 Linux 和 Windows

在规模 2000、X86 架构下，分别在 Linux 和 Windows 平台运行串行、OpenMP、OpenMP+SSE、OpenMP+AVX 共 8 个版本的代码，在本机 Visual Studio 2022 提供的 Windows 和 Intel Devcloud 提供的 Linux 平台进行测试，得到测试结果为图6.19。

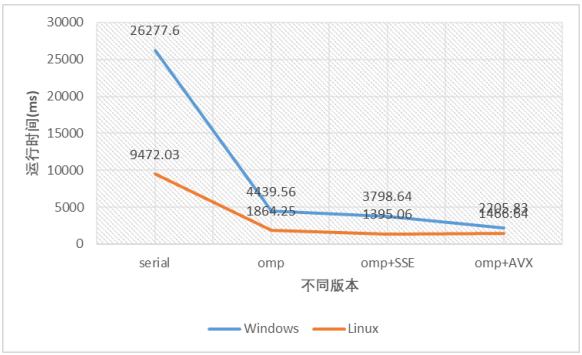


图 6.19: 普通高斯-Widows 和 Linux

可以看到，总体而言 Linux 表现优于 Windows，这与之前的实验相符。

6.1.5 不同优化力度

在 ARM 平台、2000 数据规模选择编译器的不同优化力度选项进行测试，得到数据如图6.20所示。可以看到，编译器的 O1 优化得到了质的飞跃，减少约 75%的运行时间，此后的 O2、O3 优化只有量的改变。

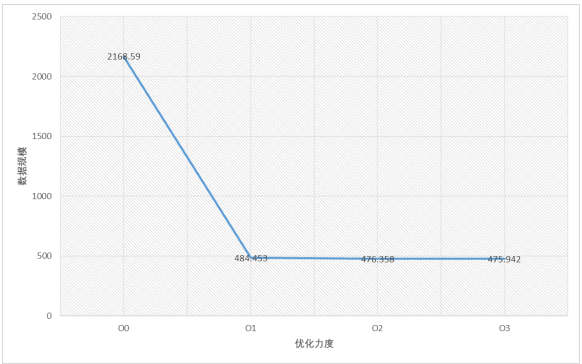


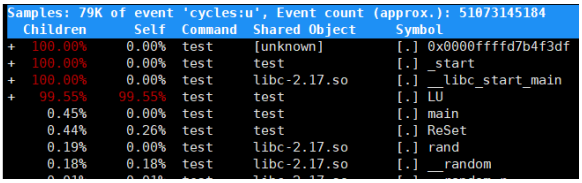
图 6.20: 普通高斯-不同优化力度

对 O0 和 O1 的代码进行 perf 分析，由图21(a)和图21(b)可看到 O0 版本为 79K，O1 版本仅为 6K。进入反汇编分析，发现 O1 优化在占比最大的函数中、**将占比最大的语句写为了 SIMD 语句**，如图22(a)。如图22(b)，经官网查阅，**得知 fmls 是 Neon 的乘法语句**，这就是 O1 优化产生极好效果的原因。

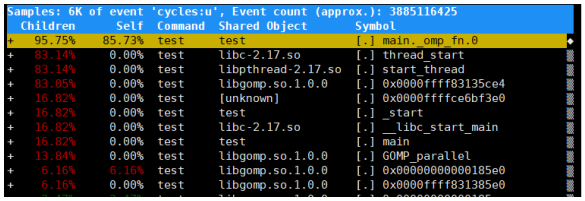
6.1.6 不同线程数

在 2000 规模下，对 7、5、3、2 线程数版本和串行版本对比，如图6.23所示。可以总结该结果如下：

1. 在 OpenMP 情况下，线程数和运行时间大致成反比；

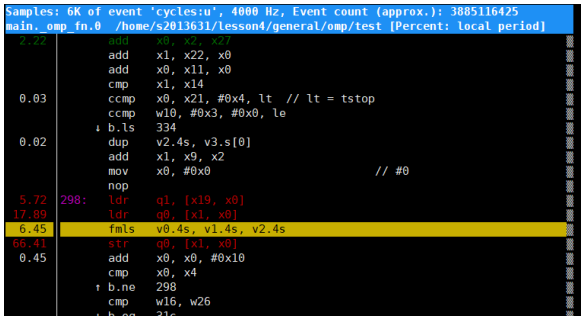


(a) O0

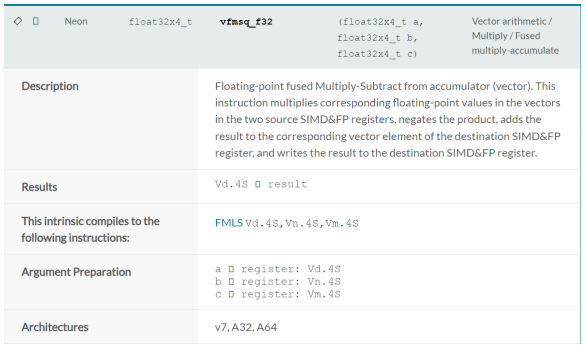


(b) O1

图 6.21: 普通高斯-不同优化力度-perf



(a) 反汇编中的 fmls 指令



(b) fmls 官网解释

图 6.22: 普通高斯-不同优化力度-perf

2. 在有无 OpenMP 情况下，运行时间和线程数的比值骤然增大，说明产生了质的改变。

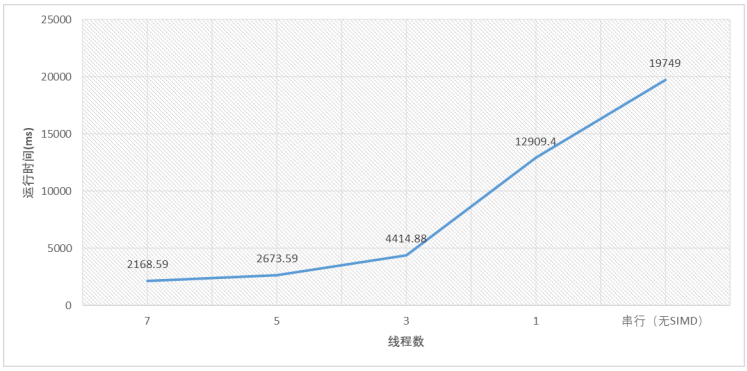


图 6.23: 普通高斯-不同线程数

6.1.7 不同算法优化

- OpenMP 自带 SIMD: 在规模 2000、1000、500 规模下，使用 #pragma omp simd 语句，将 OpenMP 自带 SIMD 和自己写的 SIMD 相比较，得到如图6.24的结果。可以看到，自己写的 SIMD 指令执行效果更优。

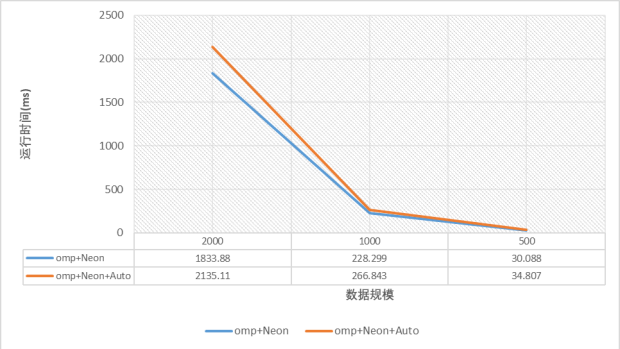


图 6.24: 普通高斯-自带 SIMD

- 负载均衡: 由于高斯消去的矩阵各行运算负载不均衡，因此我们需要对任务进行更细粒度、更灵活的划分。我在 500，1000，2000 这 3 个数据规模下尝试了：静态划分中的块划分、循环划分，动态划分，guided 划分，记录结果如图6.25所示。

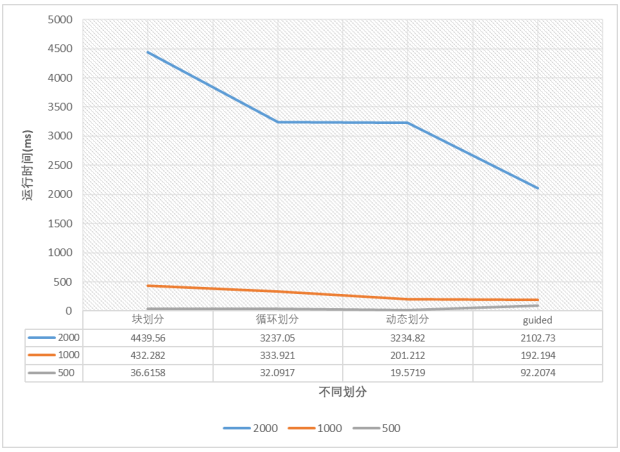


图 6.25: 普通高斯-负载均衡

可以看到，性能排序为**块划分 < 动态划分 <guided 划分**，不过在 N=500 时 guided 出现了滞后，这是因为它的 chunk_size 是指数级减小的，但对于 500、线程数为 7 这个规模而言，无论初始 chunk_size 设为多少都不太合适。因此，可以总结，当数据规模很大时使用 guided 和动态划分最好，尽量避免负载不均衡的块划分。用 Vtune 分析验证各种划分时线程的负载情况，关注核心指标 CPI Rate 并制表如3。可以看到，就 CPI 平均数而言，**动态划分的平均效率最高**；就 CPI 方差而言 (CPI 方差越大表明各线负载越不均衡，方差越笑表明各线程负载均衡)，**循环划分的负载最均衡**。

CPI Rate	块划分	循环划分	动态划分	guided 划分
平均数	0.53386	0.53414	0.51971	0.53143
标准差	0.01786	0.00923	0.01752	0.01162

表 3

- 虚假共享问题: 为探究虚假共享问题，我们在 1000、2000 数据规模下尝试了不同的块划分。**由于鲲鹏服务器的高速缓存行大小为 64B，因此我们将缓存行大小的数据划分为一个块。一个 float**

类型为 4byte，

$$64B = 64 * 8byte = 128 * 4byte$$

因此 128 个 float 型数据可以填充一个高速缓存行，我们设定最佳 chunk_size=128。实验结果如图6.26，的确 128 效果最优。

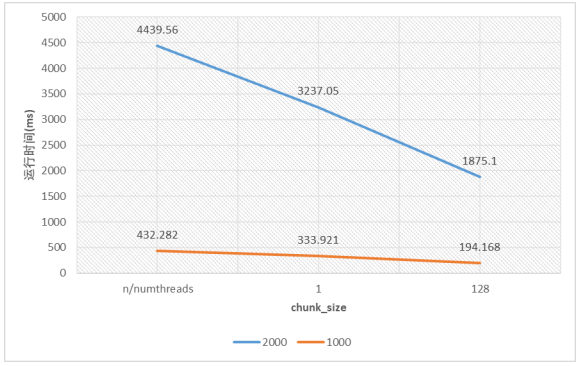


图 6.26: 普通高斯-虚假共享

- 行列划分: 设计不同的任务划分方式，对消去部分分别采用垂直划分 (列划分) 和水平划分 (行划分)，结果如图6.27所示，列划分的运行时间大于行划分，这是因为 **cache 利用上，由于行划分较列划分能更好利用空间局部性**，因此行划分的运行时间会更短。

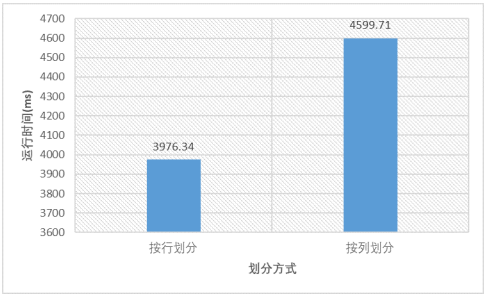


图 6.27: 普通高斯-不同划分

需要注意的是，为保持同步和正确性，需要修改代码。**由于列划分的消去代码中各线程不一定同时结束外层循环，所以不能在外层循环内就修改 A[i][k] 为 0.0，而应在外层循环结束后，指定某线程将 A[i][k] 置 0**。没有在外层循环结束时置 0，会做出列划分反而更快的结果，因为 A[i][k]==0 时运算比 A[i][k] 为非 0 浮点数时更快。代码如下：

```
1  //we divide the procedure by column.
2  for(i = k + 1;i < n; i++){
3          tmp = A[i][k];
4          //使用列划分
5          #pragma omp for
6          for(j = k + 1;j < n; j++){
7              A[i][j] = A[i][j] - tmp * A[k][j];
8          }
```

```
9
10      }
11      #pragma omp single
12      for(int i = k + 1;i < n; i++)//这样才能保证正确性
13          A[i][k] = 0.0;
```

- 对齐: 使用 2000 数据规模、SSE 的对齐/不对齐代码在 x86+Windows(Visual Studio 2022) 平台测试，得到结果如图6.28所示。由于 SSE 的对齐指令 CPI 更小，因此对齐版本的运行时间更少。

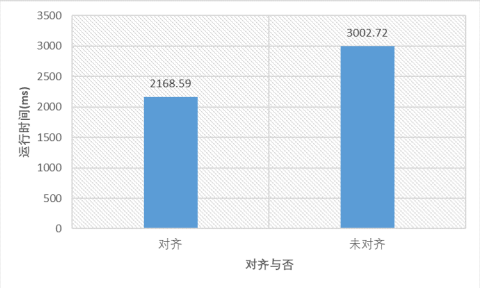


图 6.28: 普通高斯-对齐与否

6.1.8 OpenMP 和 Pthread 对比

加速比问题 在 500、1000、2000 的数据规模下实验，我们发现，omp 对串行程序的加速比远高于 pthread，如图6.29所示：

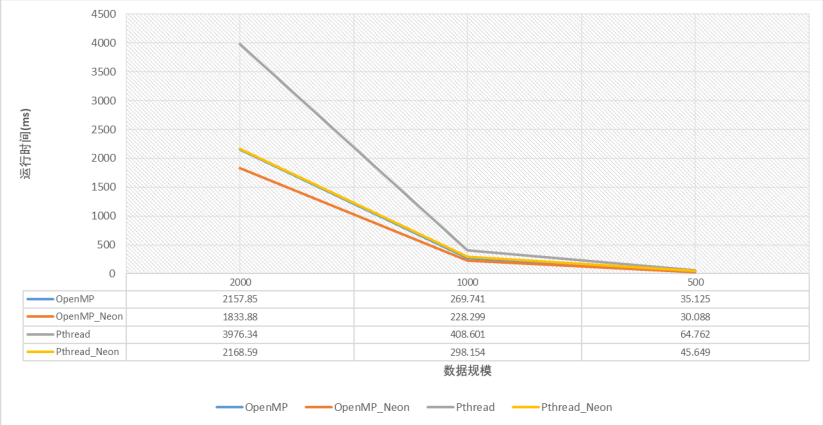
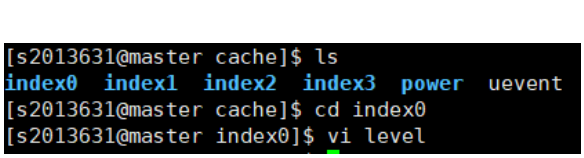


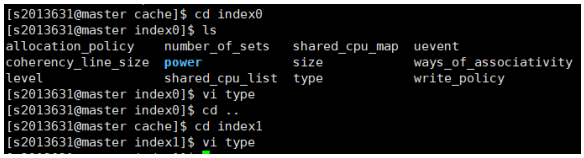
图 6.29: OpenMP 和 Pthread

无论在纯多线程代码还是加上了 Neon 的代码上，都是 OpenMP 优于 Pthread。这是为什么呢? 为查找原因，我首先在/sys/devices/system/cpu/目录下查看了鲲鹏服务器的 CPU6.30：可以看到，一个 CPU 有 2 个 L1 cache，1 个 L2 cache，1 个 L3 cache。我认为，首先，omp 在使用 cpu 的 cache 时比 pthread 划分更均匀，导致相同数据规模时，omp 的各个核心的 cache 能存下数据，保证了程序更高速的运行。

其次，为查看封装好的 omp 语言内部相比自己写的 pthread 做了哪些优化，我们在鲲鹏使用 perf 进行反汇编，比较 openmp 和 pthread 的汇编代码。如图6.31，在相同数据规模 2000 下，无论是 L1-dcache-load-misses、L1-icache-load-misses 还是 branch-load-misses，omp 都少于 pthread。进入反汇编分析，

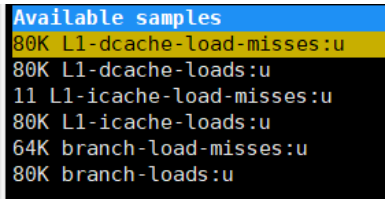


(a) CPU

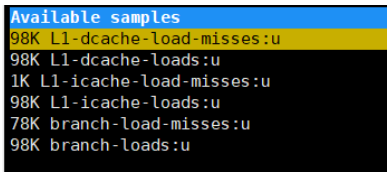


(b) Cache

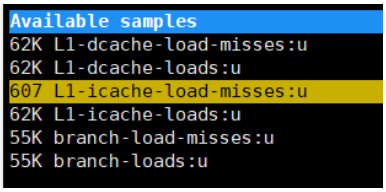
图 6.30: OpenMP 和 Pthread



(a) serial



(b) pthread



(c) omp

图 6.31: OpenMP 和 Pthread

看到在 pthread 的 miss 中占比最大的是 pthread_barrier_wait 函数（如图6.32），这是容易理解的：原本线程希望按照原有流程顺序执行，但 barrier_wait 强制线程等待，线程预测错误，产生 bubble 和 miss。进入该函数，占比最大的指令为 stxr6.33，经查阅，它是 ARM 汇编的自旋锁指令。

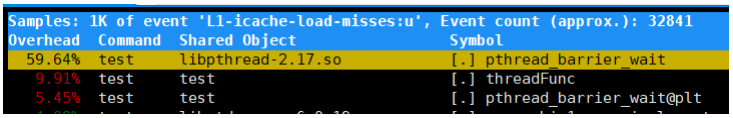


图 6.32: barrier_wait 函数在 miss 中占比最大

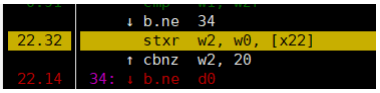


图 6.33: 在 barrier 函数中占比最大的指令是 stxr

随后，我们再对 omp 代码反汇编分析。在 omp 的 miss 中，占比最大的是主函数中的 omp function，进入该函数，可以看到，omp 在反汇编代码中调用了 GOMP_barrier 函数6.34，这就是我们在隐式使用 omp 的 barrier 时编译器帮我们做的事。我发现 omp 对于 barrier 的具体实现采用了 ldaxr 和 stlxr 指令，这也是 ARM 汇编的锁指令，查阅资料后我得知，ldaxr+stlxr(omp 隐式 barrier 使用的锁指令) 比 ldxr+stxr(pthread 显式 barrier 使用的锁指令) 性能高。这就是编译器在 omp 代码中做的优化。可以总结为如下几点：

1. 首先，omp 和 pthread 在 miss 中区别很大，而**在 miss 中占比最高的就是 barrier_wait 代码**；
2. 其次，**omp 在实现 barrier 时使用了 ldaxr+stlxr 锁指令，pthread 使用的是 stxr 锁指令，而 ldaxr+stlxr 指令效率更高**；
3. 因此，omp 的加速比会比 pthread 更好。

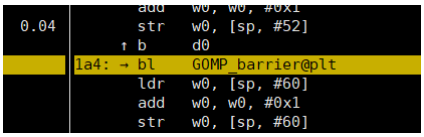
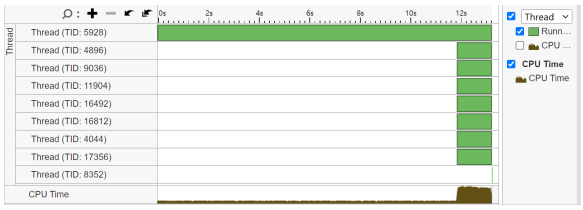
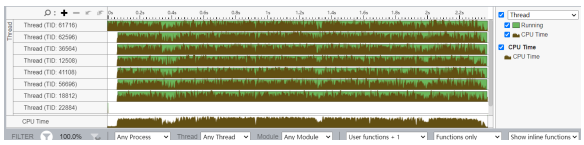


图 6.34: OpenMP 中的 barrier

工作线程问题 此外，通过 Vtune 可以看到 (图6.35)，OpenMP 的默认线程分配方式是：主线程调用工作线程，调用之后主线程也作为工作线程参与运算。而在 pthread 中，如果显示写出主线程调用、销毁工作线程，并不能使主线程也参与运算。这说明 **OpenMP 将“主线程参与工作线程的运算”这一优点封装好了，这样可以用更少的线程做同样多的运算，节省线程数。**



(a) pthread



(b) omp

图 6.35: Vtune 对比 OpenMP 和 Pthread

6.2 Tensor 上进行 OpenMP 并行

6.3 Var 上进行 OpenMP 并行

```
1       void AutoDifferentialVar::backward(const Tensor& gradient){
2       double learning_rate=0.001;
3
4       if (dataInfo.op== nullptr){
5           dataInfo.data=dataInfo.data-gradient*learning_rate;
6           return;
7       }
8
9       std::vector<Tensor>grads=dataInfo.op->backward(gradient,
10                                                     std::vector<Tensor>{dataInfo.predecessorNod
11                                                                         dataInfo.predecessorNod
12
13      #pragma omp parallel for
14      for (int i = 0; i < dataInfo.predecessorNodes.size(); ++i) {
15          dynamic_cast<GradientInterface*>(dataInfo.predecessorNodes[i])->backward(grads[i]);
16      }
17  }
```

7 实验结果

实验目前简单测试了在 windows 环境下进行并行计算的效率提升情况，采用 googletest 框架进行测试，测试数据为 10×10 的矩阵，测试代码为：

```
TEST(LossFunctionTest,MSESerialExecutionTime) {
    auto start_time = std::chrono::high_resolution_clock::now();

    // Your high-precision code to be measured goes here.
    LossFunction*lossFunction=LossFunctionFactory::getInstance().getMeanSquaredError();

    auto output3=*(*(*ADVar1-*ADVar2)**ADVar1)+*ADVar2;
    auto loss3=lossFunction->loss(output3,label);
    for (int i = 0; i < 100000; ++i) {
        loss3->backward();
        output3=*(*ADVar1-*ADVar2)**ADVar1;
        loss3=lossFunction->loss(output3,label);
    }
    output3->getData().print();
    EXPECT_EQ(output3->getData(),label->getData());

    auto end_time = std::chrono::high_resolution_clock::now();
    auto elapsed_time = std::chrono::duration_cast<std::chrono::microseconds>(end_time - start
    std::cout << "Elapsed time: " << elapsed_time << " ms" << std::endl;
}

```

测试情况如下：

8 实验不足与未来预期

由于大部分时间花费在实现自动求导部分，在框架的并行优化上剩余的时间并不算太多，因此只是进行了最基础的并行化。但最麻烦的自动求导已经基本解决，剩下的几次实验一方面将会注重对算法的并行进行优化，探索不同情况下的并行情况，另一方面将会继续在基本框架搭建完成的基础上搭建模型。

9 git 链接

https://github.com/Skyyyy0920/Parallel-Programming/tree/main

训练循环次数	普通算法耗时单位:ms	pthread 耗时单位:ms	openmp 耗时单位:ms
10000	3482042	338302	243232
20000	6934084	696404	496354
30000	10556126	1044606	744636
40000	13978166	1332808	992888
50000	17617713	1791012	1241212
60000	20892352	2029214	1499514
70000	24324498	2497426	1747726
80000	27856633	2735622	1998626
90000	31338372	3133838	2242830
100000	34828480	3482042	2432040
110000	38302762	3830248	2738248
120000	41786604	4128445	2982446
130000	45267346	4586647	3235643
140000	48943588	4824853	3483850
150000	52839630	5273052	3733752
160000	55312672	5521264	3981264
170000	59127714	5813466	4239866
180000	62681756	6266663	4487268
190000	66138898	6612882	4735780
200000	69662840	6968329	4964082

表 4