SIMD 编程实验

——以高斯消去为例

杜忱莹
2021 年 4 月
安祺
2022 年 3 月
徐一帆
2023 年 3 月
曹珉浩
2024 年 4 月
华志远，张逸非，孔德嵘
2025 年 4 月

目录

1 实验介绍

1.1 实验选题

高斯消去（基础），ANN，NTT，口令猜测算法四选一。

1.2 实验要求

1.2.1 基本要求（最高获得 90% 分数）

ARM 平台上普通高斯消去计算的基础 SIMD 并行化实验，包括设计 Neon 算法、编程实现、进行实验，讨论一些基本的算法/编程策略对性能的影响，如对齐与不对齐、选择对串行算法的不同部分（4-6 行除法、8-13 行消去）进行向量化等，实验中应测试不同问题规模下串行算法/并行算法的性能、不同算法/编程策略对性能的影响等。

1.2.2 进阶要求（最高可获得 100% 分数）

ANN，NTT，口令猜测算法三选一，具体要求可参考各实验指导书。

1.2.3 实验报告

撰写研究报告（问题描述（特别是对自主选题，首先简要描述期末研究报告的大问题，然后具体描述本次 SIMD 编程实验涉及的子问题）、SIMD 算法设计（最好有复杂性分析）与实现（伪代码）、实验及结果分析），符合科技论文写作规范，附 Git 项目链接。**不超过 15 页，但并不是页数越多分越高，助教在批改时会重点检查你的算法设计以及实验结果和分析**。

2 实验设计指导

2.1 普通高斯消去算法

2.1.1 算法分析

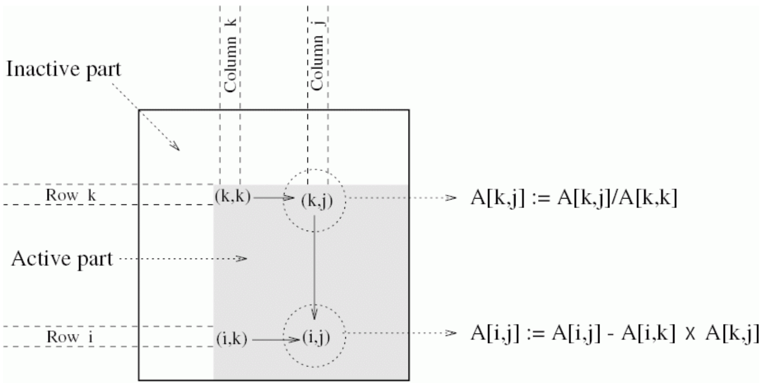

图 2.1: 高斯消去法示意图

高斯消去的计算模式如图8.3所示，主要分为消去过程和回代过程。在消去过程中进行第 k 步时，对第 k 行从 (k, k) 开始进行除法操作，并且将后续的 $k+1$ 至 N 行进行减去第 k 行的操作，全部结束后，得到如图2.2所示的结果。而回代过程从矩阵的最后一行开始向上回代，对于第 i 行，利用已知的 $x_{i+1}, x_{i+2}, ..., x_n$ 计算出 x_i。串行算法如下面伪代码所示。

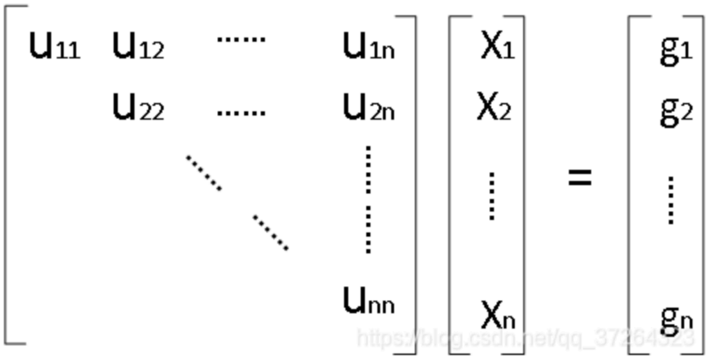

图 2.2: 高斯消去法消去过程结束后，结果的示意图

```
1  procedure Gaussian_Elimination(A, b)
2  begin
3      n := size(A)
4      // 消去过程
5      for k := 1 to n do
6          for i := k + 1 to n do
7              factor := A[i, k] / A[k, k]
8              for j := k + 1 to n do
9                  A[i, j] := A[i, j] − factor ∗ A[k, j]
10             endfor
11             b[i] := b[i] − factor ∗ b[k]
12         endfor
13     endfor
14
15     // 回代过程
16     x[n] := b[n] / A[n, n]
17     for i := n − 1 downto 1 do
18         sum := b[i]
19         for j := i + 1 to n do
20             sum := sum − A[i, j] ∗ x[j]
21         endfor
22         x[i] := sum / A[i, i]
23     endfor
24 end Gaussian_Elimination
```

观察高斯消去算法，在消去过程中：伪代码第 6，7 行第一个内嵌循环中的 $factor := A[k,j]/A[k,k]$ 以及伪代码第 8，9，10 行双层 for 循环中的 $A[i,j] := A[i,j] - factor \times A[k,j]$ 都是可以进行向量化的循环；在回代过程中，伪代码 19，20，21 行中的 $sum := sum - A[i,j] * x[j]$ 也可以进行向量化。我们可以通过 SIMD 扩展指令对这几步进行并行优化。

2.1.2 算法设计与编程

下面给出一个使用 SIMD Intrinsics 函数对普通高斯消元进行向量化的伪代码，见算法1，同学们基本上可以逐句翻译为 Neon/SSE/AVX(-512) 高斯消元函数，完成 ARM 和 X86 平台的基本实验（当然要补齐测试样例生成、时间测量等辅助程序）。篇幅所限，这里只给出了 4 路向量化的算法，对 AVX/AVX-512，同学们需将其改为 8 路/16 路向量化算法。此外，这里只给出了支持内存不对齐的 SIMD 访存操作。注意，在 x86 平台上还要考虑是否使用了支持内存不对齐的访存指令，使用内存对齐的访存指令需要对起始下标进行调整。

Algorithm 1: SIMD Intrinsic 版本的普通高斯消元

```
Data: 系数矩阵 A[n,n]
Result: 上三角矩阵 A[n,n]
for k = 0 to n-1 do
    vt ← dupTo4Float(A[k,k]);
    for j = k + 1; j + 4 <= n; j+ = 4 do
        va ← load4FloatFrom(&A[k,j]) ;    // 将四个单精度浮点数从内存加载到向量寄存器
        va ← va/vt ;                      // 这里是向量对位相除
        store4FloatTo(&A[k,j],va);        // 将四个单精度浮点数从向量寄存器存储到内存
    for j in 剩余所有下标 do
        A[k,j]=A[k,j]/A[k,k] ;            // 该行结尾处有几个元素还未计算
    A[k,k] ← 1.0;
    for i ← k+1 to n-1 do
        vaik ← dupToVector4(A[i,k]);
        for j = k + 1; j + 4 <= n; j+ = 4 do
            vakj ← load4FloatFrom(&A[k,j]);
            vaij ← load4FloatFrom(&A[i,j]);
            vx ← vakj*vaik;
            vaij ← vaij-vx;
            store4FloatTo(&A[i,j],vaij);
        for j in 剩余所有下标 do
            A[i,j] ← A[i,j] -A[k,j]*A[i,k];
        A[i,k] ← 0;
```

在设计算法时，我们需要注意以下要点：

1. **测试用例的确定**。

对本题而言，运行时间与问题规模的变化趋势不是关注重点，但是测试规模较小时，可能会出现并行算法比串行算法还要耗时的情况。此外，类似之前的作业，我们同样可以考虑 cache 大小等系统参数来设计实验中的不同问题规模。

此外，为避免出现极端情况等问题（计算结果可能会出现 Naf 或无穷），可参考如下代码生成测试用例。

Listing 1: 测试用例生成

```
float m[N][N];
void m_reset() {
    for(int i=0;i<N;i++) {
        m[i][j]=0;
        m[i][i]=1.0;
        for(int j=i+1;j<N;j++)
            m[i][j]=rand();
    }
    for(int k=0;k<N;k++)
        for(int i=k+1;i<N;i++)
            for(int j=0;j<N;j++)
                m[i][j]+=m[k][j];
}
```

2. **设计对齐与不对齐算法策略时，注意到高斯消去计算过程中，第 k 步消去的起始元素 k 是变化的，从而导致距 16 字节边界的偏移是变化的**。对于 ARM 平台的实验，在 AArch64 NEON 访存指令默认支持未对齐内存访问；在 NEON 汇编代码中可以指定对齐比特位数，这里可以进一步探究对齐 NEON 指令与未对齐性能差异，更多信息可参考官方手册对应内容（超链接）：

- 《NEON Programmer's Guide》中关于内存对齐访存指令语法和性能影响的说明

对于 x86 平台的实验，如果设计不对齐的算法策略，直接使用 _mm_loadu_ps 即可。如果设计对齐算法使用 _mm_load_ps 时，我们可以调整算法，先串行处理到对齐边界，然后进行 SIMD 的计算。可对比两种方法的性能。C++ 中数组的初始地址一般为 16 字节对齐，所以只要确保每次加载数据 $A[i:i+3]$ 中 i 为 4 的倍数即可，大家如果不确定地址是否对齐，可以直接将地址打印出来对比。同理当进行 AVX 算法设计时应该注意是否 32 字节对齐问题。还可查阅资料，不同平台和编译器下一般都有指定对齐方式的动态内存分配函数，可采用这种方式确保分配的内存起始地址是对齐的。

例如 c11 标准中可以使用 aligned_alloc 进行对齐内存分配。

```
//函数原型
void *aligned_alloc( size_t alignment, size_t size );

//例子
int *p2 = aligned_alloc(1024, 1024*sizeof *p2);
printf("1024-byte aligned addr: %p\n", (void*)p2);
free(p2);
```

3. **对不同部分的优化可进行对比实验。**

高斯消去法中有两个部分可以进行向量化，我们可以对比一下这两个部分 (一个二重循环、一个三重循环) 进行 SIMD 优化对程序速度的影响。

4. **并行计算结果的误差处理。**

并行计算由于重排了指令执行顺序，加上计算机表示浮点数是有误差的，可能导致即使数学上看是完全等价的，但并行计算结果与串行计算结果不一致。这不是算法问题，而是计算机表示、计算浮点数的误差导致，一种策略是允许一定误差，比如 $< 10e^{-6}$ 就行；另外一种策略，可在程序中加入一些数学上的处理，在运算过程中进行调整，来减小误差。我们用以下两个程序来展示。

（1）两个数相除再相乘：

```
 float a = 1.0;
float b = 3.0;
for (int i = 0; i < N; i++)
{
  a /= b;
}
for (int i = 0; i < N; i++)
{
  a *= b;
}
cout << a << endl;
```

当 N 大于一定值时输出的 a 不为 1：

```
...
N为77结果: 1
N为78结果: 1
N为79结果: 1
N为80结果: 1
N为81结果: 1
N为82结果: 0.999999
N为83结果: 0.999997
```

```
N为84结果: 0.999998
N为85结果: 0.99998
N为86结果: 1.00003
N为87结果: 1.00018
N为88结果: 1.00018
...
```

（2）N 个数求和

```
const int NUM = 2048;
const int LOGN = 12;

double elem[6][NUM], sum, sum1, sum2;
void init(double e[][NUM], int m)
{
    for (int i = 0; i < NUM; i++)
    {
        e[m][i] = (rand() % 10) / 7.0;
    }

}
void chain(int m, int n)
{
    sum = 0;
    for (int i = 0; i < n; i++) {
        sum += elem[m][i];
    }
}
void tree(int m, int n)
{
    int i, j;

    while (n >= 8) {
        for (i = 0, j = 0; i < n; i += 8) {
            elem[m][j]  = elem[m][i] + elem[m][i + 1];
            elem[m][j + 1] = elem[m][i + 2] + elem[m][i + 3];
            elem[m][j + 2] = elem[m][i + 4] + elem[m][i + 5];
            elem[m][j + 3] = elem[m][i + 6] + elem[m][i + 7];
            j += 4;
        }
        n >>= 1;
    }

    elem[m][0] += elem[m][1];
    elem[m][2] += elem[m][3];
    elem[m][0] += elem[m][2];
}
```

运行 chain 以及 tree 函数，由于这两个函数的求和顺序不一致，结果可能不一样。这里最终的一次结果为：

```
Tree: 1301.142857142856883001 51
Chain: 1301.142857142854609264 76
```

5. **更多探索**。

同学们如有余力，可探索更多的算法策略、程序优化方法，如循环展开、cache 优化、编译器不同优化力度等等。自主选题的同学不要局限于例子中循环展开、打包向量化的思路，可根据选题的特点选择恰当的 SIMD 指令进行并行优化。

2.1.3 NEON 的 C/C++ 编程

这里主要围绕 Neon Intrinsics（类似调用 C 语言函数）进行说明，如果希望进行更细粒度的编程可以考虑在源程序直接内嵌汇编代码（Neon assembly）。

使用 NEON Intrinsics 需要包含的头文件如下

```
#include <arm_neon.h>
```

编译选项：-march=native 或-march=armv8-a

一些常用的指令：

```
//数据类型
float32_t,,float32x4_t,float64x2_t,float32x4x2_t...
int8_t,int8x16_t,int16x8_t,int32x4_t,int64x2_t,int8x16x2_t...

//load：以float为例
float32x2_t vld1_f32(float32_t const *ptr);  //读取连续2个单精度浮点数到向量寄存器
float32x4_t vld1q_f32(float32_t const *ptr); //读取连续4个单精度浮点数到向量寄存器
float32x4x2_t vld2q_f32(float32_t const *ptr); //读取连续8个单精度浮点数到2个向量寄存器

//store:
void vst1q_s32(int32_t * ptr, int32x4_t val); //将向量元素保存为连续4个32位整数数
void vst1q_u32(uint32_t * ptr, uint32x4_t val);
//将向量元素保存为连续4个32位无符号整数数
void vst1q_f32(float32_t * ptr, float32x4_t val);
//将向量元素保存为连续4个单精度浮点数数

//move
int32x2_t vget_high_s32(int32x4_t a); //将a高位的两个元素复制到另一个向量寄存器
float32x2_t vget_low_f32(float32x4_t a); //将a低位的两个元素复制到另一个向量寄存器
float32_t vget_lane_f32(float32x2_t v, const int lane); //获得向量v第lane个通道的元素
float32_t vgetq_lane_f32(float32x4_t v, const int lane); //获得向量v第lane个通道的元素
float32x2_t vset_lane_f32(float32_t a, float32x2_t v, const int lane); //设置向量v第lane个通道的元素的值为a
float32x4_t vsetq_lane_f32(float32_t a, float32x4_t v, const int lane); ////设置向量v第lane个通道的元素的值为a

//arithmetic
float32x4_t vaddq_f32(float32x4_t a, float32x4_t b); //对位加法
float32x4_t vmulq_f32(float32x4_t a, float32x4_t b); //对位乘法
float32x4_t vsubq_f32(float32x4_t a, float32x4_t b); //对位减法
float32x4_t vdivq_f32(float32x4_t a, float32x4_t b); //对位除法
```

一个简单的例子：

```
float sum(float* array,int n)
{
  float sum=0;
  for (int k = 0; k < n; k++)
  {
    sum+=array[k];
  }

  return sum;
}

float sum_neon(float* array,int n)
{
  assert(n%4==0);                    // 假设n为4的倍数
  // 声明一个包含4个单精度浮点数的向量变量，用0初始化
  float32x4_t sum4=vmovq_n_f32(0);
  for(int i=0;i<n;i+=4){
    // 从(array+i)地址加载连续4个32位整数，保存到temp向量
```

```
    float32x4_t temp=vld1q_f32(array+i);
    // sum4与temp向量对位相加
    sum4=vaddq_f32(sum4,temp);
  }
  // 将低位两个元素保存到suml2向量
  float32x2_t suml2=vget_low_f32(sum4);
  // 将高位两个元素保存到sumh2向量
  float32x2_t sumh2=vget_high_f32(sum4);
  // 向量进行水平加法，得到suml2中两元素的和以及sumh2中两元素的和
  suml2=vpadd_f32(suml2,sumh2);
  // 再次进行水平加法，得到sum4向量4个元素的和
  float32_t sum=vpadds_f32(suml2);
  return (float)sum;
}
```

详细完整的 *NEON Intrinsic* 函数说明可以查询 *ARM* 官网文档（点击超链接）：Intrinsics –Arm Developer

2.1.4 SSE/AVX 的 C/C++ 编程

SSE 指令对应了 C/C++ 的 intrinsics（编译器能识别的函数，直接映射为一个或多个汇编语言指令）。使用 SSE intrinsics 所需的头文件：

```
#include <xmmintrin.h> //SSE
#include <emmintrin.h> //SSE2
#include <pmmintrin.h> //SSE3
#include <tmmintrin.h> //SSSE3
#include <smmintrin.h> //SSE4.1
#include <nmmintrin.h> //SSSE4.2
#include <immintrin.h> //AVX、AVX2
```

编译选项：-march=corei7、-march=corei7-avx、-march=native

一些常用的指令：

```
  //数据类型
__m128//float
__m128d// double
__m128i//integer
//load:
_mm_load_ps(float *p) //将从内存中地址p开始的4个float数据加载到寄存器中，要求p的地址是16字节对齐
_mm_loadu_ps(float *p)//类似_mm_load_ps但是不要求地址是16字节对齐
//set:
_mm_set_ps(float a,float b,float c,float d) //将a,b,c,d赋值给寄存器
//store:
_mm_store_ps(float *p, _m128 a) //将寄存器a的值存储到内存p中
//数据计算:
_mm_add_ps //加法
_mm_mul_ps //乘法
_mm_sub_ps //减法
_mm_div_ps //除法
```

AVX 的各个指令与 SSE 类似，如 _mm_loadu_ps 的 AVX 版本为 _mm256_loadu_ps。

一个简单的例子：

```
//串行加法:
void add() {
        for (int k = 0; k < N; k++)
         matrix[k] += 2;
}
//SSE优化
```

```
void add() {
      __m128 t1, t2;
    t1 = _mm_set1_ps(2); //t1中4个单精度浮点数设为2
    for (int k = 0; k < N; k += 4)
    {
        // 从(matrix+k)读取连续的4个单精度浮点数
      t2 = _mm_loadu_ps(matrix+k);
      // 两个向量的4个单精度浮点数对位相加
      t2 = _mm_add_ps(t1, t2);
      // 将向量t2，保存在地址(matrix+k)处
      _mm_store_ps(matrix+k, t2);
    }
}
```

可参考此例以及课程讲义中矩阵乘法的例子对 LU 中的关键循环进行向量化。更多 SSE/AVX 指令，以及 AVX 的编程大家可以参考课程讲义。

所有 SSE/AVX 指令的细节可以查询官网文档 (点击超链接)：Intel® IntrinsicsGuide

3 程序编译及运行

可以参考各选题指导书。

4 使用 VTune 等工具剖析程序性能

类似之前的实验，我们需要对程序性能进行剖析。实际上 NEON、SSE/AVX 优化与一般串行，对齐与不对齐等策略最终所执行的指令数，周期数，CPI 是不一样的，我们可以使用 perf、VTune 等 profiling 工具分析对比。具体使用方法参考体系结构调研相关及性能测试实验指导书。

5 分析汇编代码

通过研究汇编代码，我们可以更深刻地理解程序为什么会产生相应性能，以及如何更好的优化程序性能。我们可以使用 godbolt 分析程序的汇编代码。具体使用方法参考体系结构调研相关实验指导书。

6 ANN 选题：SIMD 实验

ANN 选题中 SIMD 的参考资料和要求可以查看选题指导书，这里不再赘述。下面以 SIMD 加速距离计算为例来说明如何使用 SIMD 对 ANN 算法进行加速。

6.1 准备工作

这一节只是可选项，不会对得分有任何影响。为了方便我们进行编程，我们可以首先对 SIMD 指令进行封装。假如当前架构的 SIMD 寄存器为 256 位，支持同时计算 8 个浮点数，我们可以将其封装为 simd8float32，代码如下所示：

```
struct simd8float32 {
    float32x4x2_t data;

    simd8float32() = default;

    explicit simd8float32(const float* x)
            : data{vld1q_f32(x), vld1q_f32(x + 4)} {}
```

```
    simd8float32 operator*(const simd8float32& other) const;
    simd8float32 operator+(const simd8float32& other) const;
    // 添加更多的成员函数
}
```

实现这一层封装后，我们在编写代码时可以不再需要关注底层架构，只需要将计算表达成向量化的形式即可。同时这样做也方便我们的代码移植到不同的架构中（假如你想在 x86 平台测试，只需要更改 simd8float32 类的实现即可）。

6.2 使用 SIMD 加速内积距离计算

内积的计算公式（这里假设下标从 0 开始）为:

$$\delta(x, y) = 1 - \sum_{i=0}^{d-1} x_i * y_i$$

将内积的计算向量化实际非常简单，我们将公式写为:

$$\delta(x, y) = 1 - \sum_{i=0}^{[(d-1)/8]} \sum_{j=8i}^{8i+7} x_i * y_i$$

在内层求和中，每次需要先从内层中读取浮点数，再计算 8 对浮点数的乘积，最后再求和，我们先忽略求和这一步骤，可以注意到其他步骤非常容易向量化，下方代码的第 6-7 行为忽略求和步骤的代码，此时变量 m 中存储着 8 对浮点数的乘积。注意到求和的顺序实际是可以交换的，我们可以先不进行求和，而是将结果先存在变量 sum 中，在乘积全部计算完成后再求和（代码第 11-13 行）。

```
1  float InnerProductSIMDNeon(float* b1, float* b2, size_t vecdim) {
2      assert(vecdim % 8 == 0); //假设维度能被8整除
3  
4      simd8float32 sum(0.0); //8xfloat32全部初始化为0
5      for(int i = 0; i < vecdim; i += 8) {
6          simd8float32 s1(b1 + i), s2(b2 + i);
7          simd8float32 m = s1 * s2;
8          sum += m;
9      }
10 
11     float tmp[8];
12     sum.storeu(tmp);
13     float dis = tmp[0] + tmp[1] + tmp[2] + tmp[3] + tmp[4] + tmp[5] + tmp[6] + tmp[7];
14     return 1 - dis;
15 }
```

上面的代码只是一个非常朴素的实现（并不是最优），实际上该代码还有很多优化空间，大家可以自行调研更多 neon 指令，看是否存在可以进一步优化上面代码的指令，或者尝试利用循环展开等技术继续对距离计算进行加速。

6.3 使用 SIMD 加速 PQ 距离计算

在阅读本节之前，你需要先了解 PQ 的原理，需要理解查找表（look-up-table，LUT），码本（codebook），子空间，聚类中心等概念。在计算 PQ 距离时，我们选择用每个子空间的聚类中心来代替原向量，假设我们将原空间切割为 4 份子空间，且原空间维度为 96 维，我们首先将距离计算公式拆成如下 4 份:

$$\delta(x, q) = 1 - (\sum_{i=0}^{23} x_i * q_i + ... + \sum_{i=72}^{95} x_i * q_i)$$

我们令第 k 份子空间向量 x 的码本为 o_k（即该向量在第 k 份子空间中对应的聚类中心编号为 o_k），并令 c_{o_k} 表示该向量在第 k 份子空间对应的聚类中心向量，$c_{o_k,i}$ 表示向量的第 i 维度的值，则距离计算公式可以写为:

$$\delta(x,q) = 1 - (\sum_{i=0}^{23} c_{o_1,i} * q_i + ... + \sum_{i=72}^{95} c_{o_4,i-72} * q_i)$$

注意到每个子空间我们一般只会聚 256 个或 16 个类，我们以 256 为例，o_k 的取值实际上只有 256 种（即 0-255），所以我们可以对每一段子空间预处理所有的 $\sum(c_{o_k,i} * q_i)$，这样我们在计算查询与 base 的距离时，只需要查询我们预处理的结果，然后将 4 个结果相加即可。我们令 $LUT_{k,i}$ 表示在第 k 段子空间中，查询 q 到聚类中心 i 的内积，那么 PQ 的距离计算可以写为:

$$\delta(x,q) = 1 - \sum_{k=1}^{4} LUT_{k,o_k}$$

这一部分向量化难度非常大，因为在计算每个向量时，o_k 的值是在 0-255 随机出现的，所以在访问 LUT 时并没有数据的局部性，所以我们可以考虑一次同时计算多个向量的 PQ 距离，并在这一步进行向量化（而不是向量化单个 PQ 距离的计算），这里细节留给大家自行推导。

实际上，目前对 PQ 距离计算的最佳方式是将 LUT 直接存储到 SIMD 寄存器中，那么此时在计算时便不再需要访问内存（连 cache 都不需要访问了），这一技术即 fastscan，大家可以参考以下资料进行实现（如果追求满分的话推荐实现 fastscan）。

- Cache locality is not enough:High-Performance Nearest Neighbor Search with Product Quantization Fast Scan。
- RaBitQ: Quantizing High-Dimensional Vectors with a Theoretical Error Bound for Approximate Nearest Neighbor Search的第 2 节，如果想实现 rabitq，可以阅读论文的 3.3 节来了解 rabitq 的 fastscan 算法细节。
- 知乎一篇对 ScaNN 介绍中的 4bitPQ 章节

7 口令猜测并行化选题：SIMD 实验

网上有很多现成的 MD5 哈希算法原理介绍，这里就不再赘述了。这里主要讲解代码框架与 SIMD 的并行思路。

7.1 SIMD 加速的 MD5 哈希算法

我们首先看看框架中实现的串行 MD5 哈希算法:

```
void MD5Hash(string input, bit32 *state)
{

  Byte *paddedMessage;
  int *messageLength = new int[1];
  for (int i = 0; i < 1; i += 1)
  {
    paddedMessage = StringProcess(input, &messageLength[i]);
    // cout<<messageLength[i]<<endl;
    assert(messageLength[i] == messageLength[0]);
  }
  int n_blocks = messageLength[0] / 64;
```

```

  state[0] = 0x67452301;
  state[1] = 0xefcdab89;
  state[2] = 0x98badcfe;
  state[3] = 0x10325476;

  // 逐block地更新state
  for (int i = 0; i < n_blocks; i += 1)
  {
    bit32 x[16];

    // 此处省去一些预处理

    bit32 a = state[0], b = state[1], c = state[2], d = state[3];

    auto start = system_clock::now();
    /* Round 1 */
    FF(a, b, c, d, x[0], s11, 0xd76aa478);
    // FF函数继续运行15次，此处省略

    /* Round 2 */
    GG(a, b, c, d, x[1], s21, 0xf61e2562);
    // GG函数继续运行15次，此处省略

    /* Round 3 */
    HH(a, b, c, d, x[5], s31, 0xfffa3942);
    // HH函数继续运行15次，此处省略

    /* Round 4 */
    II(a, b, c, d, x[0], s41, 0xf4292244);
    // II函数继续运行15次，此处省略

    state[0] += a;
    state[1] += b;
    state[2] += c;
    state[3] += d;
  }

  // 省去一些后续处理
}
```

对于 MD5 而言，输入的消息必须经过预处理，才能进行后续的运算。这个预处理过程非常复杂，但已经在 StringProcess 函数中封装好了。后续的运算则涉及 FF、GG、HH、II 四个函数的多次运算，这些函数正是你需要使用 SIMD 重点进行并行化的部分。

```
#define F(x, y, z) (((x) & (y)) | ((~x) & (z)))
#define G(x, y, z) (((x) & (z)) | ((y) & (~z)))
#define H(x, y, z) ((x) ^ (y) ^ (z))
#define I(x, y, z) ((y) ^ ((x) | (~z)))

#define FF(a, b, c, d, x, s, ac) { \
  (a) += F ((b), (c), (d)) + (x) + ac; \
  (a) = ROTATELEFT ((a), (s)); \
  (a) += (b); \
}

#define GG(a, b, c, d, x, s, ac) { \
  (a) += G ((b), (c), (d)) + (x) + ac; \
  (a) = ROTATELEFT ((a), (s)); \
  (a) += (b); \
}
```

```

#define HH(a, b, c, d, x, s, ac) { \
  (a) += H ((b), (c), (d)) + (x) + ac; \
  (a) = ROTATELEFT ((a), (s)); \
  (a) += (b); \
}

#define II(a, b, c, d, x, s, ac) { \
  (a) += I ((b), (c), (d)) + (x) + ac; \
  (a) = ROTATELEFT ((a), (s)); \
  (a) += (b); \
}
```

观察这些函数，这些函数所设计的运算不涉及条件判断，并且为较简单的运算（移位、与运算，etc.）。这就使得这些函数非常适合用 SIMD 进行并行化处理：一次性地让函数同时处理多个消息（口令），即可做到并行化。

7.2 口令猜测并行化选题：SIMD 基础要求

口令猜测并行化选题的 SIMD 实验工程难度较大，特别是正确性等细节上需要或长或短的调试时间。请务必预留一定时间进行实验。本选题的基础要求为（80% 的分数）：尝试在 ARM 服务器上，基于 NEON 实现 MD5 哈希算法。具体要求如下：

- 要求有基本的正确性，即能够利用 SIMD 指令，做到一次性产生多个消息（口令）的哈希值，并且哈希值正确。
- 不要求真正实现相对串行算法的加速，但是仍需要给出实验数据，并且分析为什么没有能够实现加速。

7.3 口令猜测并行化选题：SIMD 进阶要求

本选题的进阶要求**不需要每条都完成，选取一部分完成即可**，将会根据工作量、你的分析、实现难度，综合进行给分。具体要求如下：

- 实现相对串行算法的加速。
- 如果你实现了加速，尝试通过查阅资料、profiling 等方式，说明编译时的选项（例如：不进行编译优化、-O1 优化、-O2 优化）会对加速比产生怎样的影响，为什么会产生影响。
- 使用 SSE 指令集在你的 x86 设备上实现 SIMD 并行算法。
- 利用 perf 对串行和并行算法分别进行性能分析。
- SIMD 可以控制“单指令多数据”中，数据的总数。例如，可以一次计算两个、四个、八个消息（口令）的哈希值。尝试改变单次运算的并行度，并探索其加速效果会发生怎样的改变。

上述要求供大家参考。**如果完成了有一定难度、思维量、工作量的额外工作，但不在上述要求中，仍会包含到本次作业的总成绩中**。

8 NTT 选题：SIMD 实验

NTT 的 SIMD 优化需要分别实现以下两个难点 (也即给分点):

1. neon 的向量化操作不支持取模, 需要自行实现
2. NTT(递归版) 的循环主体需要使用蝴蝶变换, 导致向量内部需要进行手动的交换

前提级, 如果你实在无法实现某一个难点的向量化, 也可以回退到最基础的版本 (朴素多项式乘法和逐个取模), 最终得分根据实现的 SIMD 优化类型给分, 而非根据最终评测性能给分。

8.1 向量化取模

如果无法完成向量化取模, 甚至多项式乘法都难以实现, 在这里提出三个解决方案:

- 涉及到取模时单独处理向量里的每一个元素, 也就是除取模操作外其他操作均使用向量化
- 使用浮点数除法近似取模操作
- 使用 Montgomery 规约将模乘转化为支持向量化的操作

对于第一个解决方法不做描述。

浮点数近似取模原理比较简单:

$$a \times b\%p = a \times b - p \times \left\lfloor \frac{a \times b}{p} \right\rfloor = a \times b - p \times \left\lfloor \frac{1}{p} \times a \times b \right\rfloor$$

这一串式子中, $\frac{1}{p}$ 可以提前预处理出来, 因此所有操作均可由 SIMD 提供的函数实现。

需要注意的是, 由于模数较大, 且涉及到浮点数操作, 使用这种方法进行向量化取模会导致精度损失, 进而导致答案与正确答案存在一定偏差, 如果采用了这种方式实现 SIMD 取模一定会导致答案错误, 但在本次实验中对这部分误差进行了忽略, 即使你使用浮点数近似取模最终导致多项式乘法的答案错误 (除样例 0 和样例 1 外甚至相差非常多), 也会按照答案正确给分 (但不会超过 Montgomery 规约的分数)。

在了解 Montgomery 规约前需要保证你对乘法逆元等基础数论概念有所了解 (即 NTT 前置数论知识)。

Montgomery 规约是专门用于取模优化的算法 (后续实验也会介绍另一种规约), 目前网络上介绍原理的资料较多, 仅对原理进行简述, 本次实验核心也不在于 Montgomery 规约的原理而是如何使用 SIMD 优化 Montgomery 规约, 如果你参考了已有的某些网站 Montgomery 规约的代码或仓库, 必须在报告中指出, 否则会进行扣分 (当然 neon 优化的不可能找得到), 提醒, 这一部分的实现难度相对较大。

Montgomery 空间由模数 n 和一个满足 $r \geq n$ 且与 n 互质的正整数 r 定义, 通常取 $r = 2^{32}$ 或 $r = 2^{64}$。

定义 x 在 Montgomery 空间中的数值为 $\bar{x} = x \cdot r \bmod n$。

在 Montgomery 空间内, 加减等的运算与常规运算相同, 但乘法不同, 定义 $*$ 为 Montgomery 空间内的乘法, $\cdot$ 为常规运算乘法, 则

$$\bar{x} * \bar{y} = \overline{x \cdot y} = (x \cdot y) \cdot r \bmod n$$

$$\bar{x} \cdot \bar{y} = (x \cdot y) \cdot r \cdot r \bmod n$$

$$\bar{x} * \bar{y} = \bar{x} \cdot \bar{y} \cdot r^{-1} \bmod n$$

对于 $\bar{x} \cdot \bar{y} \bmod n$ 可以直接计算, 对于涉及到除法 $x \cdot r^{-1} \bmod n$, Montgomery 空间内除 2 不需要除法, 因此可以得到最终化简式 (n' 可求):

$$x \cdot r^{-1} \equiv (x - x \cdot n' \bmod r \cdot n)/r$$

通过规约将乘法取模转化。

一个简要的规约实现如下:

```
typedef __uint32_t u32;
typedef __uint64_t u64;

const u32 n = 1e9 + 7, nr = inverse(n, 1ull << 32);

u32 reduce(u64 x) {
    u32 q = u32(x) * nr;      // q = x * n' mod r
    u64 m = (u64) q * n;      // m = q * n
    u32 y = (x - m) >> 32; // y = (x - m) / r
    return x < m ? y + n : y; // 保证 y 非负
}
```

容易发现规约中的所有操作均可以使用 SIMD 优化, 你可以所有运算过程均在 Montgomery 数域下进行, 也可以只针对模乘。

- 维基百科对 Montgomery 模乘的详细介绍。
- 上古时代介绍 Montgomery 规约原理的论文
- 知乎一篇介绍几种优化取模方法

8.2 向量化蝴蝶变换

```

for(int mid = 1; mid < limit; mid <<= 1) {
    for(int j = 0; j < limit; j += (mid << 1)) {
        int w = 1;// 旋转因子
        for(int k = 0; k < mid; k++, w = w * Wn {
            // 运算主体
            // 计算 a[j + k],
            // 计算 a[j + k + mid];
        }
    }
}
```

以上是蝴蝶变换的主体, mid 代表当前运算的步长, 第一层循环是对分治的模拟, 第二层循环和第三层循环是具体的分治过程, 即模拟合并两块等步长的序列。第三层循环显然可以进行 SIMD 优化, 当步长小于向量长度时, 直接使用朴素方法计算, 当步长大于等于向量长度, 就可以使用向量进行优化。

在完成以上蝴蝶变换的基础要求后，在这里提出一个较难的算法：DIT/DIF, 其参考资料放在最下方。

DIT(Decimation in Time)，按时间抽取的 NTT 的一种实现就是 Cooley-Tukey 算法，即常见的实现，这里直接跳过。

DIF(Decimation in Frequency)，按频率抽取的 NTT。

DIT 的作用是输入一个正常顺序序列的 a，一个按位翻转后的序列 ω_{rev}，输出按位翻转后的按时间抽取的信号。

DIF 的作用是输入一个按位翻转后的信号，一个按位翻转后的序列 ω_{rev}^{-1}，输出正常顺序序列 a。

所以只需要将多项式先 DIT，做完运算后直接 DIF，这样就不需要位翻转, 只需要在初始化的时候进行一次。

如果你尝试了这种方式优化, 需要注意向量内部的变换。

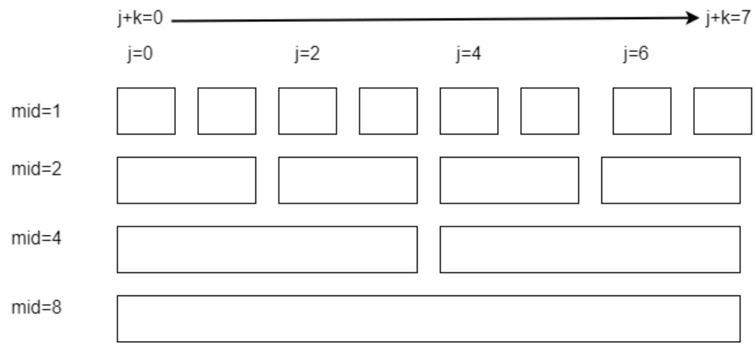

图 8.3: NTT 三层循环的理解

- DIF/DIT 的实现及原理。
- 目前较前卫的 SIMD 优化 NTT