南開大學
Nankai University

计算机学院

并行程序设计实验报告

SIMD 编程

姓名：张铭徐 张惠程

学号：2113615 2112241

专业：计算机科学与技术

2023 年 4 月 7 日

目录

1 实验目的及背景介绍

1.1 实验目的

SIMD（Single Instruction Multiple Data）指令集是一种用于在单个指令中处理多个数据元素的计算机指令集。使用 SIMD 指令集可以在相同的时间内处理多个数据元素，从而提高计算机程序的运行效率。

在深度学习中，神经网络的训练通常需要大量的计算。使用 SIMD 指令集可以在较短的时间内对大量的数据进行计算，从而加快神经网络的训练速度。此外，许多深度学习框架（如 TensorFlow、PyTorch 和 MXNet）都已经优化了其底层的计算库，使其能够利用 SIMD 指令集进行高效的计算。

具体来说，使用 SIMD 指令集可以优化以下计算任务：

- 矩阵乘法：在神经网络中，矩阵乘法是一项非常常见的计算任务。使用 SIMD 指令集可以加速矩阵乘法的计算，从而加快神经网络的训练速度。
- 向量运算：在深度学习中，许多计算都涉及到对向量的操作，如向量加法、向量点积等。使用 SIMD 指令集可以同时处理多个向量，从而加快这些向量运算的速度。
- 卷积运算：卷积运算是深度学习中常见的计算任务之一，也是计算量最大的任务之一。使用 SIMD 指令集可以加速卷积运算的计算，从而加快神经网络的训练速度。

总的来说，使用 SIMD 指令集可以显著提高深度学习的计算效率，从而使神经网络的训练速度更快、更有效。本次实验将聚焦于卷积运算以及矩阵乘法运算两个微小的操作来优化深度学习的训练过程。

1.2 实验背景知识介绍

在深度学习中，矩阵乘法和卷积运算是两个非常常见的操作。下面分别介绍它们的基本概念和使用 SIMD 优化的必要性。

矩阵乘法是深度学习中非常常见的操作之一。假设有两个矩阵 A 和 B，它们的维度分别为 $m \times n$ 和 $n \times p$，则它们的矩阵乘法结果为 C，维度为 $m \times p$。矩阵乘法的公式如下所示：

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

其中，$C_{i,j}$ 表示矩阵 C 中第 i 行第 j 列的元素，$A_{i,k}$ 表示矩阵 A 中第 i 行第 k 列的元素，$B_{k,j}$ 表示矩阵 B 中第 k 行第 j 列的元素。

在深度学习中，矩阵乘法通常用于计算两个矩阵的点积，以及神经网络中的前向传播和反向传播等操作。由于神经网络通常包含大量的矩阵计算，因此使用 SIMD 指令集可以加速这些计算，提高神经网络的训练速度。

卷积运算是深度学习中非常常见的操作之一，用于提取图像、音频等数据中的特征。假设有两个矩阵 I 和 K，它们的维度分别为 $m \times n$ 和 $k \times k$，则它们的卷积运算结果为 S，维度为 $(m-k+1) \times (n-k+1)$。卷积运算的公式如下所示：

$$S_{i,j} = \sum_{u=1}^{k} \sum_{v=1}^{k} I_{i+u-1,j+v-1} K_{u,v} S$$

其中，$S_{i,j}$ 表示卷积运算结果中第 i 行第 j 列的元素，$I_{i+u-1,j+v-1}$ 表示输入矩阵中第 $(i+u-1)$ 行第 $(j+v-1)$ 列的元素，$K_{u,v}$ 表示卷积核矩阵中第 u 行第 v 列的元素。

在深度学习中，卷积运算通常用于卷积神经网络中的卷积层，用于提取图像、音频等数据中的特征。由于神经网络中通常需要进行大量的卷积运算，因此使用 SIMD 指令集可以加速这些计算，提高神经网络的训练速度。

使用 SIMD 指令集可以对矩阵乘法和卷积运算进行高效的并行计算。在计算机硬件中，现代 CPU 和 GPU 都支持 SIMD 指令集。当使用 SIMD 指令集时，计算单元可以同时处理多个数据元素，从而加快计算速度。

例如，在使用 SIMD 指令集进行矩阵乘法计算时，可以同时计算多个元素的乘积和求和。这可以通过将矩阵拆分为多个小矩阵来实现，然后对每个小矩阵进行并行计算。

在使用 SIMD 指令集进行卷积运算时，可以利用卷积运算的局部性质，将输入矩阵拆分为多个小矩阵，然后对每个小矩阵进行并行计算。这可以减少数据的重复计算，从而加快计算速度。

因此，使用 SIMD 指令集可以显著提高深度学习模型的计算效率，从而加快模型的训练速度。

1.3 实验平台介绍

本次实验平台采用 X86 平台，操作系统为 Windows 11，CPU 为 Intel Core12th，实验电脑型号为联想拯救者 Y9000P2022 版，GPU 为移动端 130W 的 RTX3060，同时，计算机的 RAM 为 16G，编译器采用 Visual Studio。对于 Profiling 部分，采用 VTune 进行剖析，电脑采用 X86 平台，CPU 为 Intel Core11th, 操作系统和编译器同上。

2 对矩阵乘法进行 SIMD(使用 SSE 指令集) 优化

2.1 朴素算法设计思想

朴素版的矩阵乘法算法使用三重循环进行计算，对于矩阵 $A \in \mathbb{R}^{m\times n}$ 和矩阵 $B \in \mathbb{R}^{n\times p}$，可以得到其乘积矩阵 $C \in \mathbb{R}^{m\times p}$:

$$C_{i,j} = \sum_{k=1}^{n} A_{i,k}B_{k,j} \quad (i = 1, \ldots, m;\ j = 1, \ldots, p)$$

算法的时间复杂度为 $O(n^3)$，在矩阵较大时，计算代价会比较高。根据上述的设计思想，我们可以写出算法流程图，如算法1所示:

Algorithm 1 矩阵乘法朴素版

Input: 矩阵 A 和矩阵 B
Output: 矩阵 $C = A \times B$
$m \leftarrow$ 矩阵 A 的行数 $n \leftarrow$ 矩阵 A 的列数 $p \leftarrow$ 矩阵 B 的列数 初始化矩阵 C 为 $m \times p$ 的零矩阵 **for** $i \leftarrow 1$ **to** m **do**
  for $j \leftarrow 1$ **to** p **do**
    $c_{i,j} \leftarrow 0$ **for** $k \leftarrow 1$ **to** n **do**
      $c_{i,j} \leftarrow c_{i,j} + a_{i,k} \times b_{k,j}$
    end
  end
end

2.2 矩阵乘法 SIMD 优化算法 (使用 SSE 指令集) 设计思想

在 SIMD 优化的算法中，我们使用向量寄存器进行优化。将矩阵拆分为多个小矩阵，然后使用向量指令进行计算，可以一次性处理多个数据。具体步骤如下:

- 将矩阵 B 转置为 B^T。
- 初始化一个向量寄存器，用于存储 B^T 矩阵中的一列。
- 对于矩阵 A 的每一行，取出一列数据，与 B^T 中的一列进行向量乘法。
- 将所有向量乘积的结果求和，得到 C 矩阵的一个元素。
- 算法的时间复杂度为 $O(\frac{n^3}{w})$，其中 w 表示向量寄存器的长度。通过使用向量寄存器，可以显著提高计算效率。

对于上述的分析过程，我们可以写出对应的算法流程图, 如算法4所示：

Algorithm 2 SSE 优化矩阵乘法

Input: 矩阵 $A_{Kwin\times M}$ 和矩阵 $B_{Kwout\times Kwin}$
Output: 矩阵 $C_{Kwout\times M} = A \times B$
将 B 矩阵转置，存储在新矩阵 B' 中; 初始化矩阵 C 为 $Kwout \times M$ 的零矩阵; **for** $i \leftarrow 1$ **to** $Kwout$ **do**
  for $j \leftarrow 1$ **to** M **do**
    加载向量 c_i，长度为 4，初始值为 $[0,0,0,0]$; **for** $k \leftarrow 1$ **to** $Kwin$ **do**
      加载向量 $a_{k,j}$ 和 $b'i,k$，长度均为 4; 对 ak,j 和 $b'i,k$ 进行乘法运算; 将乘积向量加入 c_i;
    end
    将 c_i 的值写入矩阵 Ci,j;
  end
end

2.3 代码

根据上面的分析，我们在下面给出平凡算法和 SSE 优化的矩阵乘法代码，值得注意的是，我们仅仅给出矩阵乘法对应的函数，输入数据和整体代码详见后文的 GitHub 项目链接。

矩阵乘法平凡算法

```cpp
std::vector<std::vector<double>> matrix_multiply(const
    std::vector<std::vector<double>>& A, const std::vector<std::vector<double>>& B)
    {
  int m = A.size();
  int n = A[0].size();
  int p = B[0].size();
  std::vector<std::vector<double>> C(m, std::vector<double>(p, 0.0));
  for (int i = 0; i < m; i++) {
      for (int j = 0; j < p; j++) {
          for (int k = 0; k < n; k++) {
              C[i][j] += A[i][k] * B[k][j];
          }
      }
  }
  return C;
}
```

矩阵乘法 SSE 优化算法

```cpp
std::vector<std::vector<float>> matrix_multiply_sse(const
    std::vector<std::vector<float>>& A, const std::vector<std::vector<float>>& B) {
    int m = A.size();
    int n = A[0].size();
    int p = B[0].size();
    std::vector<std::vector<float>> C(m, std::vector<float>(p, 0.0));

    for (int i = 0; i < m; i++) {
        for (int j = 0; j < p; j++) {
            __m128 sum = _mm_setzero_ps();
            for (int k = 0; k < n; k += 4) {
                __m128 a = _mm_loadu_ps(&A[i][k]);
                __m128 b = _mm_loadu_ps(&B[k][j]);
                sum = _mm_add_ps(sum, _mm_mul_ps(a, b));
            }
            float temp[4];
            _mm_storeu_ps(temp, sum);
            C[i][j] = temp[0] + temp[1] + temp[2] + temp[3];

            for (int k = n - n % 4; k < n; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }

    return C;
}
```

本算法使用了 SSE 指令集的优化，可以实现更快的矩阵乘法计算，与平凡的矩阵乘法相比，它具有以下优点：

- 使用向量化的 SIMD 指令：SSE 指令集允许对多个浮点数进行并行计算，通过一次指令处理多个数据，从而减少了循环计算次数和存取次数，可以加速矩阵乘法运算。
- 使用更少的寄存器访问：相比于朴素矩阵乘法的三重循环嵌套，SSE 优化版本只需要访问两个矩阵 A 和 B，因此可以减少寄存器的访问。
- 提高了数据局部性：SSE 指令集可以实现连续存储的数据进行向量计算，这可以提高数据的局部性，减少了数据的访问冲突，进一步提高了计算效率。
- 可移植性强：SSE 指令集在现代的 x86 架构 CPU 上都得到了广泛支持，因此可以说这种方法具有较强的可移植性，可以在不同的 CPU 上获得类似的性能提升。

总的来说，使用 SSE 指令集的优化版本可以在保证结果正确性的前提下，通过 SIMD 指令的并行计算、数据局部性优化等手段，实现更快的矩阵乘法计算。

3 对卷积操作进行 SIMD(使用 SSE 指令集) 优化

3.1 朴素算法设计思想

朴素版的卷积运算算法使用四重循环进行计算，对于输入矩阵 $I \in \mathbb{R}^{m\times n}$ 和卷积核 $K \in \mathbb{R}^{k\times k}$，可以得到其输出矩阵 $S \in \mathbb{R}^{(m-k+1)\times(n-k+1)}$：

$$S_{i,j} = \sum_{u=1}^{k}\sum_{v=1}^{k} I_{i+u-1,j+v-1}K_{u,v} \quad (i = 1,\ldots,m-k+1;\ j = 1,\ldots,n-k+1)$$

算法的时间复杂度为 $O(n^4)$，在卷积核较大时，计算代价会比较高。对于朴素算法，其算法流程图如算法 3 所示：

Algorithm 3 卷积运算朴素版

Input: 输入矩阵 I 和卷积核 K
Output: 输出矩阵 S
$m \leftarrow$ 输入矩阵 I 的行数 $n \leftarrow$ 输入矩阵 I 的列数 $k \leftarrow$ 卷积核 K 的大小 初始化输出矩阵 S 为 $(m-k+1)\times(n-k+1)$ 的零矩阵 **for** $i \leftarrow 1$ **to** $m-k+1$ **do**
  for $j \leftarrow 1$ **to** $n-k+1$ **do**
    $S_{i,j} \leftarrow 0$ **for** $u \leftarrow 1$ **to** k **do**
      for $v \leftarrow 1$ **to** k **do**
        $S_{i,j} \leftarrow S_{i,j} + I_{i+u-1,j+v-1} \times K_{u,v}$
      end
    end
  end
end

3.2 卷积操作 SIMD(使用 SSE 指令集) 优化算法设计思想

在 SIMD 优化的算法中，我们同样使用向量寄存器进行优化。将输入矩阵和卷积核矩阵分别展开成一维向量，然后使用向量指令进行计算，可以一次性处理多个数据。具体步骤如下：

- 将卷积核矩阵展开成一维向量 K_{vec}。
- 对于输入矩阵 I 的每个子矩阵，将其展开成一维向量 I_{vec}。
- 使用向量指令对 I_{vec} 和 K_{vec} 进行向量乘法，得到一组乘积向量。
- 将乘积向量的所有元素相加，得到输出矩阵 S 的一个元素。
- 算法的时间复杂度为 $O(\frac{n^4}{w})$，其中 w 表示向量寄存器的长度。通过使用向量寄存器，可以显著提高计算效率。

综上所述，通过使用向量寄存器和向量指令进行 SIMD 优化，可以大大提高矩阵乘法和卷积运算的计算效率，从而加速深度学习模型的训练速度。在实际应用中，我们可以根据具体情况选择不同的优化方法，以达到最优的计算效率。基于上述的说明方法，我们可以写出对应的算法流程图，如算法**??**所示：

Algorithm 4 SSE 优化卷积操作

Input: 输入矩阵 I、核矩阵 K、输入矩阵的大小 $m \times n$、核矩阵的大小 $k \times k$
Output: 输出矩阵 O
输入: I, K, m, n, k
输出: O
for $i \leftarrow 0$ **to** $m-k$ **do**
  for $j \leftarrow 0$ **to** $n-k$ **do**
    $sum \leftarrow 0$; **for** $u \leftarrow 0$ **to** $k-1$ **do**
      for $v \leftarrow 0$ **to** $k-1$ **do**
        $pixel \leftarrow I_{i+u,j+v}$; $kernel \leftarrow K_{u,v}$; $sum \leftarrow sum + pixel \times kernel$;
      end
    end
    $O_{i,j} \leftarrow sum$;
  end
end
for $i \leftarrow 0$ **to** $\lfloor \frac{m-k}{4} \rfloor$ **do**
  for $j \leftarrow 0$ **to** $\lfloor \frac{n-k}{4} \rfloor$ **do**
    $sum \leftarrow$ SSE 向量$(0,0,0,0)$; **for** $u \leftarrow 0$ **to** $k-1$ **do**
      for $v \leftarrow 0$ **to** $k-1$ **do**
        $pixel \leftarrow$ SSE 向量$(I_{i+u,j\times4+v}, I_{i+u,(j\times4+v)+1}, I_{i+u,(j\times4+v)+2}, I_{i+u,(j\times4+v)+3})$; $kernel \leftarrow$ SSE 向量$(K_{u,v}, K_{u,v}, K_{u,v}, K_{u,v})$; $sum \leftarrow sum + pixel \times kernel$;
      end
    end
    将 sum 中的 4 个元素相加；将结果存储在输出矩阵 O 中；
  end
end

3.3 代码

根据上述代码描述，我们在下面给出平凡卷积算法和 SSE 优化卷积算法所对应的 C++ 代码，同样的，我们也仅仅给出对应的计算函数，全部代码将在 GitHub 的项目链接中给出。

卷积操作平凡代码

```cpp
std::vector<float> convolution_avx2(const std::vector<float>& signal, const
    std::vector<float>& kernel) {
    int signal_length = signal.size();
    int kernel_length = kernel.size();
    int result_length = signal_length + kernel_length - 1;
    std::vector<float> result(result_length, 0);

    for (int i = 0; i < signal_length; ++i) {
        for (int j = 0; j < kernel_length; ++j) {
            float kernel_elem = kernel[j];
            float signal_elem = signal[i];
            int result_idx = i + j;
            result[result_idx] += kernel_elem * signal_elem;
        }
    }

    return result;
```

```
}
```

卷积操作 SSE 优化算法

```
std::vector<float> convolution_sse(const std::vector<float>& signal, const
    std::vector<float>& kernel) {
    int signal_length = signal.size();
    int kernel_length = kernel.size();
    int result_length = signal_length + kernel_length - 1;
    std::vector<float> result(result_length, 0);
    for (int i = 0; i < signal_length; ++i) {
        for (int j = 0; j < kernel_length; ++j) {
            int result_idx = i + j;

            if (i % 4 == 0 && i + 3 < signal_length) {
                __m128 kernel_elem = _mm_set1_ps(kernel[j]);
                __m128 signal_elems = _mm_loadu_ps(&signal[i]);
                __m128 result_elems = _mm_loadu_ps(&result[result_idx]);
                result_elems = _mm_add_ps(result_elems, _mm_mul_ps(signal_elems,
                    kernel_elem));
                _mm_storeu_ps(&result[result_idx], result_elems);
                i += 3;
            }
            else {
                result[result_idx] += signal[i] * kernel[j];
            }
        }
    }
    return result;
}
```

4 实验结果分析

4.1 对矩阵乘法结果分析

我们上文给出了 SSE 优化算法和平凡算法的卷积操作和矩阵乘法，我们利用如下代码测定了不同数据规模下程序的运行时间：

auto start = std::chrono::high_resolution_clock::now();
auto end = std::chrono::high_resolution_clock::now();

最终程序的运行时间即为 $time = end - start$，我们在下表1中给出了不同数据规模之下程序的运行时间：为方便起见，我们在程序中设定参与运算的矩阵都是方阵。同时，为减少误差，参与运算的元素都是由系统随机数随机生成的 0-9 之间的数。

其中，参与运算的矩阵为矩阵 $A \in \mathbb{R}^{m \times n}$ 和矩阵 $B \in \mathbb{R}^{n \times p}$。我们绘制出了可视化的图表，如下图4.1所示：

表 1: 矩阵乘法运行时间比较

参数指标	origin	SSE 优化
$n = m = p = 800$	58.45s	7.04s
$n = m = p = 700$	33.22s	4.86s
$n = m = p = 600$	17.83s	3.10s
$n = m = p = 500$	9.40s	1.76s
$n = m = p = 400$	4.93s	1.04s
$n = m = p = 300$	2.12s	0.37s
$n = m = p = 200$	0.74s	0.11s
$n = m = p = 100$	0.08s	0.015s

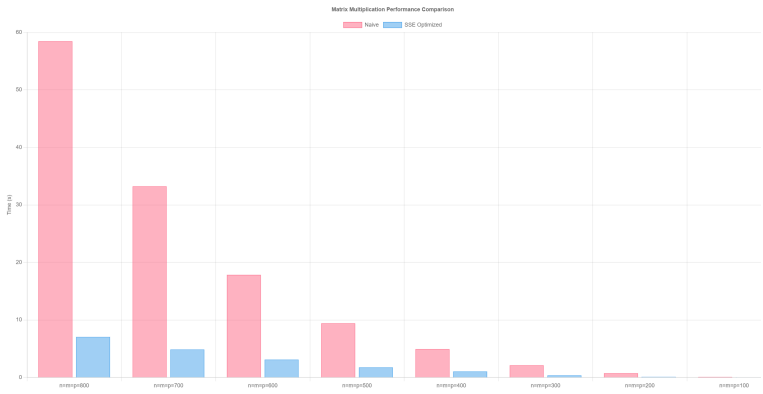

图 4.1: 矩阵乘法运行时间图

4.1.1 结果分析

根据给出的数据，SSE 优化算法相较于平凡算法实现矩阵乘法运行时间大幅度减少，且随着矩阵维度的增加，两者之间的差距也越来越大。这是由于 SSE 指令集在单个指令中能够处理更多数据元素，从而实现更高的并行度，能够显著提高矩阵乘法算法的计算效率，特别是在进行大规模矩阵操作时。

4.2 对卷积操作的结果分析

与矩阵乘法类似的，我们测定了在不同数据规模下 SSE 优化算法和平凡算法的程序执行时间，给出如下所示的表格和可视化图表。如表格2所示和图4.2所示：

表 2: 卷积运行时间比较

参数指标	运行时间（`origin`）	运行时间（`SSE` 优化）
30000 2400	2.850s	0.801s
20000 1600	1.417s	0.371s
10000 800	0.344s	0.089s
5000 400	0.087s	0.023s
4000 320	0.055s	0.019s
3000 240	0.039s	0.010s
2000 160	0.017s	0.005s
1000 80	0.003s	0.002s

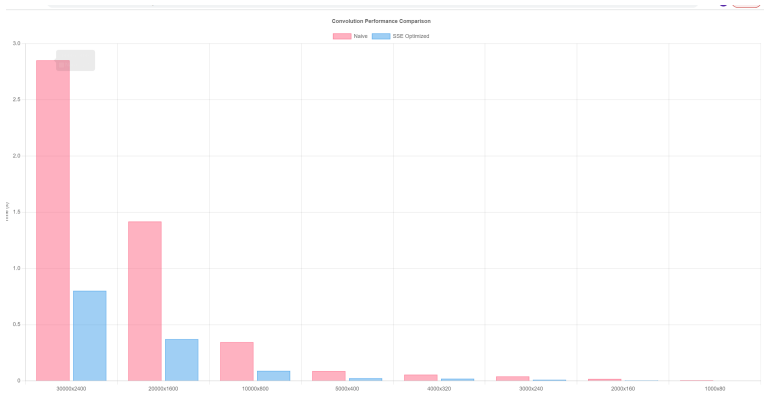

图 4.2: 卷积运行时间比较

4.2.1 结果分析

SSE 优化卷积操作相比于平凡算法速度更快，它利用了 CPU 的 SIMD 向量化指令，即在同一时间内可以处理更多的数据，提高了计算效率。

而平凡的卷积算法使用多重循环来实现，每次只能处理一个数据。在 SSE 优化卷积操作中，可以使用向量化指令一次性处理多个数据，例如 SSE 指令集中的`_mm_mul_ps`可以同时计算 4 个单精度浮点数的乘积。这种方法可以将数据的处理量提高到 4 倍，减少内存读写的开销，从而显著提高了卷积操作的计算速度。

此外，SSE 指令集还提供了其他优化指令，如`_mm_add_ps`、`_mm_sub_ps`等，可以进一步优化计算过程中的加减操作，从而进一步提高计算效率。因此，使用 SSE 指令集进行向量化优化的卷积操作可以大幅提高计算速度，特别是在对大量数据进行并行处理时，效果更加明显。从我们实验数据中，也可以发现这一点。

5 Profiling 分析

为探究出现上述结果的原因，我们使用 VTune 对所写的程序进行 profiling 分析：我们分析的指标以及具体的含义如下所示：

- CPI (Cycles Per Instruction)：此指标显示每条指令执行所需的平均周期数。较低的 CPI 表示指令执行效率更高。我们预测 SIMD 优化的版本可能具有较低的 CPI，因为它使用更高效的向量指令。
- Instructions Retired：已完成执行的指令总数。优化版本可能具有较低的 Instructions Retired，因为它使用向量指令一次处理多个数据元素。
- L1/L2/L3 Cache Misses：这些指标衡量缓存未命中的数量。如果优化的实现能够更有效地利用缓存，那么我们会看到较低的缓存未命中次数。
- Elapsed Time：这是程序执行所需的总时间。优化的实现应该具有较短的执行时间。
- CPU 时间：用于衡量应用程序在处理器上执行的时间。显然，优化算法应该具有更快的 CPU 时间

- Parallelism ： 并行性是一项用于衡量应用程序在多个处理器核心上的并行执行效果的指标。它可以帮助我们了解应用程序在并行计算方面的效率，从而确定是否可以通过优化代码进一步提高性能。

我们在 VTune 中分析两种算法的性能表现，具体每种算法的表现如下表3所示：

表 3: 卷积操作优化算法和平凡算法 VTune 分析指标

性能指标	卷积操作	
	平凡算法	SSE 优化算法
CPI	1.239	0.918
Instructions Retired	37358955	33972208
L1/L2/L3 Cache Misses	0%	0%
Elapsed Time	0.022s	0.017s
CPU Time	0.014s	0.009s
Parallelism	0.90%	3.30%

表 4: 矩阵乘法优化算法和平凡算法 VTune 分析指标

性能指标	平凡算法	SSE 优化算法
CPI	0.796	0.91
Instructions Retired	50993099958	9126021350
L1/L2/L3 Cache Misses	0.076(L1) 0.2(L2) 0.2(L3)	0.007(L1)
Elapsed Time	10.468s	1.741s
CPU Time	9.469s	1.562s
Parallelism	5.60%	5.60%

从上面的 VTune 分析参数可知，性能差异主要体现在 CPI，指令数上，显然，通过 AVX2 指令集的优化，我们的计算机执行算法的效率变高了，同时，指令数也减少了，体现在软件层面就是运行时间即 CPU 时间减少，总的来看，这些指标的变化符合我们上面的分析。也就是说，我们通过 profiling 分析，找到了优化算法性能表现更佳的原因。

6 实验总结与反思

6.1 SSE 与 AVX2 的异同

在另一位同学的实验中，采用了 SSE 指令集进行优化，我们可以看到，SSE 指令集相较于 AVX2 的效果较差一些，经过查阅资料和分析，我们找到了 SSE 和 AVX2 算法有以下异同之处：

相同之处：

- 都属于 SIMD 指令集：AVX2 和 SSE 都是 SIMD 指令集，它们通过在 CPU 中同时处理多个数据元素来实现并行性，从而提高程序的性能。
- 都是针对浮点数和整数操作的：AVX2 和 SSE 都支持浮点数和整数操作，包括加法、乘法、乘加等。
- 提高程序性能：通过使用 AVX2 或 SSE 指令集，您可以优化程序的性能，特别是在进行矩阵操作、图像处理、信号处理等高度数据并行的计算任务时。

不同之处：

- 寄存器大小：SSE 指令集使用 128 位的寄存器（XMM 寄存器），而 AVX2 指令集使用更大的 256 位寄存器（YMM 寄存器）。这意味着 AVX2 可以在单个指令中处理更多数据元素，从而实现更高的并行度。
- 指令集功能：AVX2 相比 SSE 提供了更多的功能和指令。例如，AVX2 支持整数的 Fused Multiply-Add（FMA）操作，而 SSE 不支持。此外，AVX2 还引入了许多针对整数操作的指令，包括排列、混合、位操作等，这在 SSE 中是不完整的或者完全缺失的。
- 性能差异：由于更大的寄存器大小和更丰富的指令集功能，AVX2 通常比 SSE 提供更好的性能。然而，实际性能差异可能因处理器架构、编译器优化选项以及输入数据的大小和特性而有所不同。
- 兼容性：AVX2 是在较新的处理器上引入的，而 SSE 是较早引入的指令集。因此，在较旧的处理器上，可能仅支持 SSE，而不支持 AVX2。在使用 AVX2 或 SSE 优化代码时，应检查处理器和编译器的兼容性。

总之，AVX2 和 SSE 都是针对数据并行计算的 SIMD 指令集，但 AVX2 提供了更大的寄存器、更丰富的指令集功能和更好的性能。然而，在选择使用 AVX2 或 SSE 时，应考虑到处理器和编译器的兼容性。既然 AVX2 使用了更大的寄存器，那么其有更好的性能表现也就不奇怪了，由于篇幅限制，在这里不展示 AVX2 与 SSE 指令集具体的性能表现的差异。在另一份实验报告中，将详细的体现 SSE 优化算法与平凡算法的性能表现差异，此外，详细的实验数据和 Profiling 指标源文件将在并行 GitHub 项目链接中给出。

6.2 实验分工

本次实验的前期搜集资料，阅读相关文献的过程由张铭徐完成，针对后期的 profiling 数据分析由张惠程完成；此外，张铭徐负责了 AVX2 优化算法，张惠程使用 SSE 优化算法，并一起讨论出了上述的实验结论。