

计算机系统Lab1实验报告

22级 JOHN班 涂文良

任务1：向量按元素乘法

实现思路

为了保证加速前后分配内存的开销一致，我统一采取了使用 C 库函数 `aligned_alloc` 进行动态内存分配：

```
int * v1 = (int *)aligned_alloc(32, n * sizeof(int));  
int * v2 = (int *)aligned_alloc(32, n * sizeof(int));  
int * res = (int *)aligned_alloc(32, n * sizeof(int));
```

其中 `v1` 数组存储第一个向量，`v2` 数组存储第二个向量，`res` 数组存储结果。

然后，我首先写出了加速前的朴素思路，即遍历一遍数组，对应位置依次相乘：

```
for(int i = 0; i < n; i++)  
{  
    res[i] = v1[i] * v2[i];  
}
```

这一部分代码写在文件 vecmult_origin.cpp 中。

接着我思考对这一实现进行加速。首先我声明了所指类型为 `__m256i` 的指针。 `__m256i` 是包含256个比特、存储整数的向量，也就是说存储 `int` 的话可以存储8个。

接着我使用了 `immintrin.h` 库中的 `_mm256_mullo_epi32` 函数， 它会把两个 `__m256i` 每32位视为一个整数，作为一个整体做整数乘法并存储到另一个 `__m256i` 中。这个函数只会存储结果中低的一半，但由于我们的数据范围保证了每个数都不超过 32768， 所以这里的整数乘法是不会越界的。这保证了程序的正确性。然后这里每次 `r++` `v11++` `v12++` 都会移动8个整数， 所以循环只用进行 `n / 8` 轮就可以。

代码实现如下：

```
__m256i * v11 = (__m256i*)v1;
__m256i * v12 = (__m256i*)v2;
__m256i * r = (__m256i*)res;

for(int i = 0; i < n / 8; i++)
{
    *r = _mm256_mullo_epi32(*v11, *v12);
    r++; v11++; v12++;
}
```

这部分代码实现在 `vecmult.cpp` 中。

性能分析

我使用的编译命令如下：（vecmult_origin.cpp是未使用avx2拓展的朴素代码实现，vecmult.cpp是使用avx2拓展加速的代码实现。）

```
g++ -O0 -mavx2 -o vm ./vecmult.cpp
g++ -O0 -o vmo ./vecmult_origin.cpp
```

我使用了C++中的chrono库记录程序从输入完毕开始到准备输出所经过的执行时间。这一部分代码如下

```
#include <chrono>
...
auto start_time = std::chrono::high_resolution_clock::now();
...
auto end_time = std::chrono::high_resolution_clock::now();
auto duration =
std::chrono::duration_cast<std::chrono::milliseconds>
(end_time - start_time);
```

我的实验数据如下：

1.数据大小 $N = 4096 * 256$

执行时间：

```
● tuwenliang@LAPTOP-HP0IMVKG:/mnt/d/AVX_Extension$ ./vmo
Before AVX acceleration, execution time taken: 5 milliseconds
● tuwenliang@LAPTOP-HP0IMVKG:/mnt/d/AVX_Extension$ ./vm
After AVX acceleration, execution time taken: 1 milliseconds
```

加速效果: 80%

2.数据大小 $N = 4096 * 512$

执行时间:

```
● tuwenliang@LAPTOP-HP0IMVKG:/mnt/d/AVX_Extension$ ./vmo
Before AVX acceleration, execution time taken: 9 milliseconds
● tuwenliang@LAPTOP-HP0IMVKG:/mnt/d/AVX_Extension$ ./vm
After AVX acceleration, execution time taken: 2 milliseconds
```

加速效果: 77.8%

3.数据大小 $N = 4096 * 1024$

执行时间:

```
● tuwenliang@LAPTOP-HP0IMVKG:/mnt/d/AVX_Extension$ ./vmo
Before AVX acceleration, execution time taken: 18 milliseconds
● tuwenliang@LAPTOP-HP0IMVKG:/mnt/d/AVX_Extension$ ./vm
After AVX acceleration, execution time taken: 7 milliseconds
```

加速效果: 61.2%

4.数据大小 $N = 4096 * 2048$

执行时间:

```
● tuwenliang@LAPTOP-HP0IMVKG:/mnt/d/AVX_Extension$ ./vmo
Before AVX acceleration, execution time taken: 37 milliseconds
● tuwenliang@LAPTOP-HP0IMVKG:/mnt/d/AVX_Extension$ ./vm
After AVX acceleration, execution time taken: 17 milliseconds
```

加速效果: 54.1%

5.数据大小 $N = 4096 * 4096$

执行时间:

```
● tuwenliang@LAPTOP-HP0IMVKG:/mnt/d/AVX_Extension$ ./vmo
Before AVX acceleration, execution time taken: 79 milliseconds
● tuwenliang@LAPTOP-HP0IMVKG:/mnt/d/AVX_Extension$ ./vm
After AVX acceleration, execution time taken: 45 milliseconds
```

加速效果: 43.0%

5.数据大小 $N = 4096 \ 4096 \ 2$

执行时间:

```
● tuwenliang@LAPTOP-HP0IMVKG:/mnt/d/AVX_Extension$ ./vmo
Before AVX acceleration, execution time taken: 156 milliseconds
● tuwenliang@LAPTOP-HP0IMVKG:/mnt/d/AVX_Extension$ ./vm
After AVX acceleration, execution time taken: 90 milliseconds
```

加速效果: 42.3%

5.数据大小 $N = 4096 \ 4096 \ 4$

执行时间:

```
● tuwenliang@LAPTOP-HP0IMVKG:/mnt/d/AVX_Extension$ ./vmo
Before AVX acceleration, execution time taken: 380 milliseconds
● tuwenliang@LAPTOP-HP0IMVKG:/mnt/d/AVX_Extension$ ./vm
After AVX acceleration, execution time taken: 159 milliseconds
```

加速效果: 58.1%

5.数据大小 $N = 4096 \ 4096 \ 8$

执行时间:

```
● tuwenliang@LAPTOP-HP0IMVKG:/mnt/d/AVX_Extension$ ./vmo
Before AVX acceleration, execution time taken: 680 milliseconds
● tuwenliang@LAPTOP-HP0IMVKG:/mnt/d/AVX_Extension$ ./vm
After AVX acceleration, execution time taken: 289 milliseconds
```

加速效果: 57.5%

5.数据大小 $N = 4096 \ 4096 \ 16$

执行时间:

```
● tuwenliang@LAPTOP-HP0IMVKG:/mnt/d/AVX_Extension$ ./vmo
Before AVX acceleration, execution time taken: 1888 milliseconds
● tuwenliang@LAPTOP-HP0IMVKG:/mnt/d/AVX_Extension$ ./vm
After AVX acceleration, execution time taken: 653 milliseconds
```

加速效果: 65.4%

6.数据大小 $N = 4096 \ 4096 \ 32$

执行时间:

```
● tuwenliang@LAPTOP-HP0IMVKG:/mnt/d/AVX_Extension$ ./vmo
Before AVX acceleration, execution time taken: 33689 milliseconds
● tuwenliang@LAPTOP-HP0IMVKG:/mnt/d/AVX_Extension$ ./vm
After AVX acceleration, execution time taken: 15485 milliseconds
```

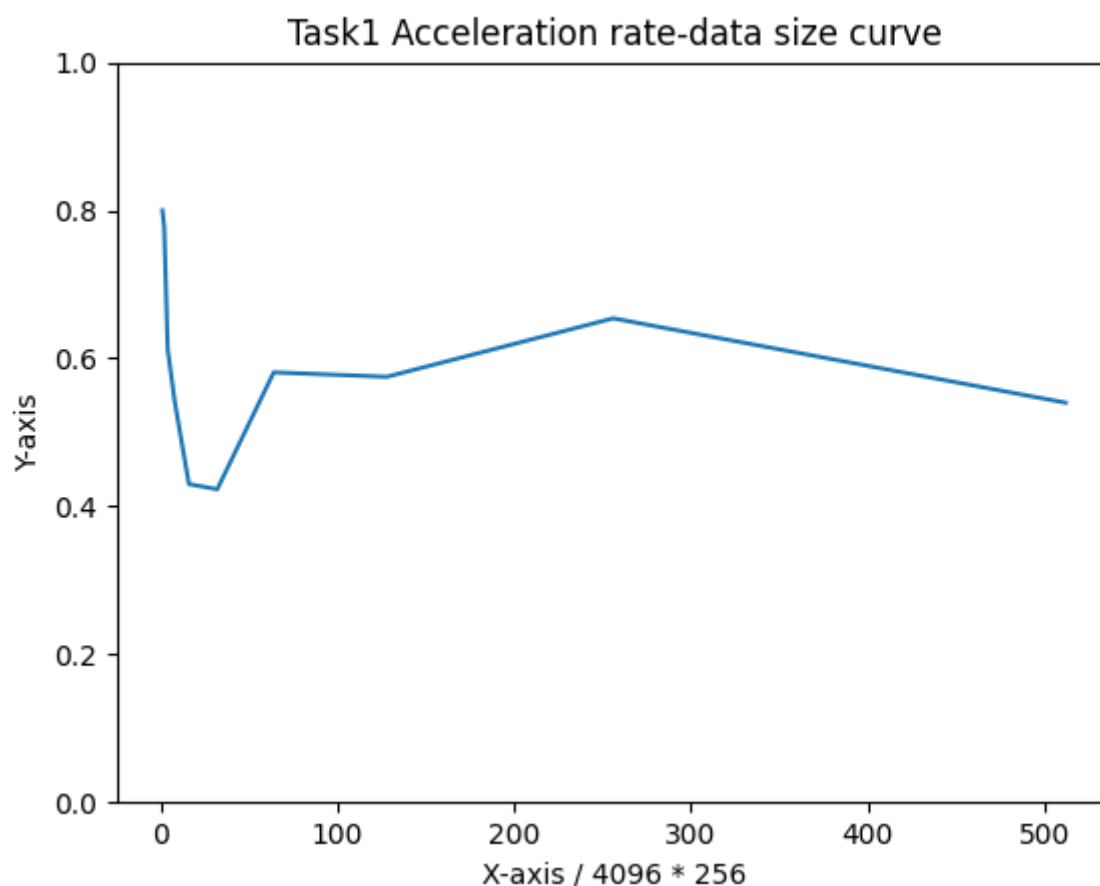
加速效果: 54.0%

列出数据表格如下:

数据大小 N	加速前执行时间 / milliseconds	加速后执行时间 / milliseconds	加速效果
4096 x 256	5	1	80%
4096 x 512	9	2	77.8%
4096 x 1024	18	7	61.2%
4096 x 2048	37	17	54.1%
4096 x 4096	79	45	43.0%
4096 x 4096 x 2	156	90	42.3%
4096 x 4096 x 4	380	159	58.1%
4096 x 4096 x 8	680	289	57.5%
4096 x 4096 x 16	1888	653	65.4%
4096 x 4096 x	33689	15485	54.0%

数据大小 N	加速前执行时间 / milliseconds	加速后执行时间 / milliseconds	加速效果
32			

使用 python 的 matplotlib 作出加速效果-数据大小曲线图如下：



可以看出，任务2的加速效果大致在40 - 60%左右浮动，当数据很小时加速效果甚至达到80%，加速效果较好。

任务2：向量矩阵乘法

实现思路

这里首先由于结果可能超出 int 的范围，所以我使用 long long 存储输入和结果。

同样为了保证加速前后分配内存的开销一致，我统一采取了使用 C 库函数 aligned_alloc 进行动态内存分配：

```
long long * v = (long long *)aligned_alloc(64, n *
sizeof(long long));
long long * m = (long long *)aligned_alloc(64, n * n *
sizeof(long long));
long long * res = (long long *)aligned_alloc(64, n *
sizeof(long long));
```

其中 v 数组存储向量， m 数组存储矩阵， 相当于二维数组拉直， res 数组存储结果。

然后，我首先写出了加速前的朴素思路，即遍历一遍 m，使用矩阵乘法的定义计算：

```
long long sum = 0;

for(int i = 0; i < n; i++)
{
    sum = 0;
    for(int j = 0; j < n; j++)
    {
        sum += m[i * n + j] * v[j];
    }
}
```

```
    res[i] = sum;  
}
```

这一部分代码写在文件 `vmmo.cpp` 中。

接着我思考对这一实现进行加速。首先我声明了所指类型为 `__m256i` 的指针。 `__m256i` 是包含256个比特、存储整数的向量，也就是说存储 `long long` 的话可以存储4个。

接着我使用了 `immintrin.h` 库中的 `_mm256_mullo_epi32` 函数和 `_mm256_add_epi64` 函数，前一个函数用来做整数乘法，由于数据范围的保证，这部分结果还是在 `int` 范围内，后一个函数用来把结果累加到一个临时向量 `Tmp1` 中，这部分结果可能超出 `int` 的范围，所以必须声明类型为 `long long`。

另外还有些实现细节如下：我的矩阵是存在一个一维数组 `m` 中，所以相当于把二维数组拉直了。那么我需要每次算完一行之后把结果存入，并把指向 `v` 向量的指针置回开头。我的做法是循环一共进行 $n * n / 4$ 轮，每次判断是否 $i * 4 \% n == 0$ ，即是否算完了一行，如果是，那么我把结果写入 `res` 数组，把 `v1` 指到 `v` 开头。如果不是，我把 `v1++`。

代码实现如下：

```
__m256i * v1 = (__m256i*)v;  
__m256i * m1 = (__m256i*)m;
```

```

__m256i * r = (__m256i*)res;
__m256i * Tmp1 = (__m256i*)Tmp;

__m256i * temp = v1;

for(int i = 0; i < n * n / 4; )
{
    *Tmp1 = _mm256_add_epi64(*Tmp1,
    _mm256_mullo_epi32(*m1, *v1));
    m1++;
    i++;
    if(i * 4 % n == 0)
    {
        v1 = temp;
        for(int k = 0; k < 4; k++)
        {
            res[i * 4 / n - 1] += Tmp[k];
            Tmp[k] = 0;
        }
    }
    else
    {
        v1++;
    }
}
}

```

这部分代码实现在 vmm.cpp 中。

性能分析

我使用的编译命令如下：（vmmo.cpp是未使用avx2拓展的朴素代码实现，vmm.cpp是使用avx2拓展加速的代码实现。）

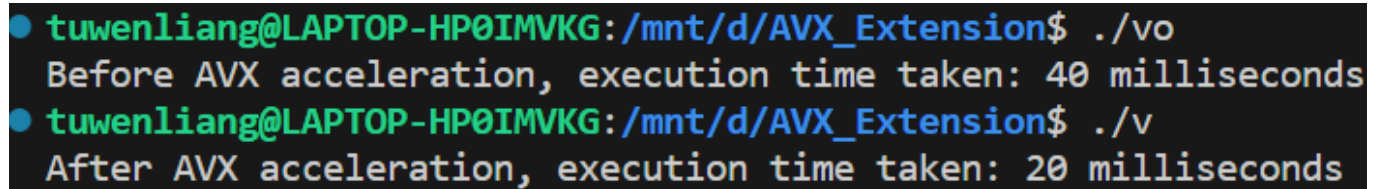
```
g++ -O0 -mavx2 -o v ./vmm.cpp
g++ -O0 -o vo ./vmmo.cpp
```

我同样使用了C++中的 chrono 库记录程序从输入完毕开始到准备输出所经过的执行时间。

我的实验数据如下：

1.数据大小 $N = 4096$

执行时间：

A terminal window showing the execution of two programs. The first command is './vo' and the output is 'Before AVX acceleration, execution time taken: 40 milliseconds'. The second command is './v' and the output is 'After AVX acceleration, execution time taken: 20 milliseconds'.

```
tuwenliang@LAPTOP-HP0IMVKG:/mnt/d/AVX_Extension$ ./vo
Before AVX acceleration, execution time taken: 40 milliseconds
tuwenliang@LAPTOP-HP0IMVKG:/mnt/d/AVX_Extension$ ./v
After AVX acceleration, execution time taken: 20 milliseconds
```

加速效果：50%

2.数据大小 $N = 4096 * 2$

执行时间：

```
● tuwenliang@LAPTOP-HP0IMVKG:/mnt/d/AVX_Extension$ ./vo
Before AVX acceleration, execution time taken: 168 milliseconds
● tuwenliang@LAPTOP-HP0IMVKG:/mnt/d/AVX_Extension$ ./v
After AVX acceleration, execution time taken: 90 milliseconds
```

加速效果：42.9%

3.数据大小 $N = 4096 * 4$

执行时间：

```
● tuwenliang@LAPTOP-HP0IMVKG:/mnt/d/AVX_Extension$ ./vo
Before AVX acceleration, execution time taken: 749 milliseconds
● tuwenliang@LAPTOP-HP0IMVKG:/mnt/d/AVX_Extension$ ./v
After AVX acceleration, execution time taken: 368 milliseconds
```

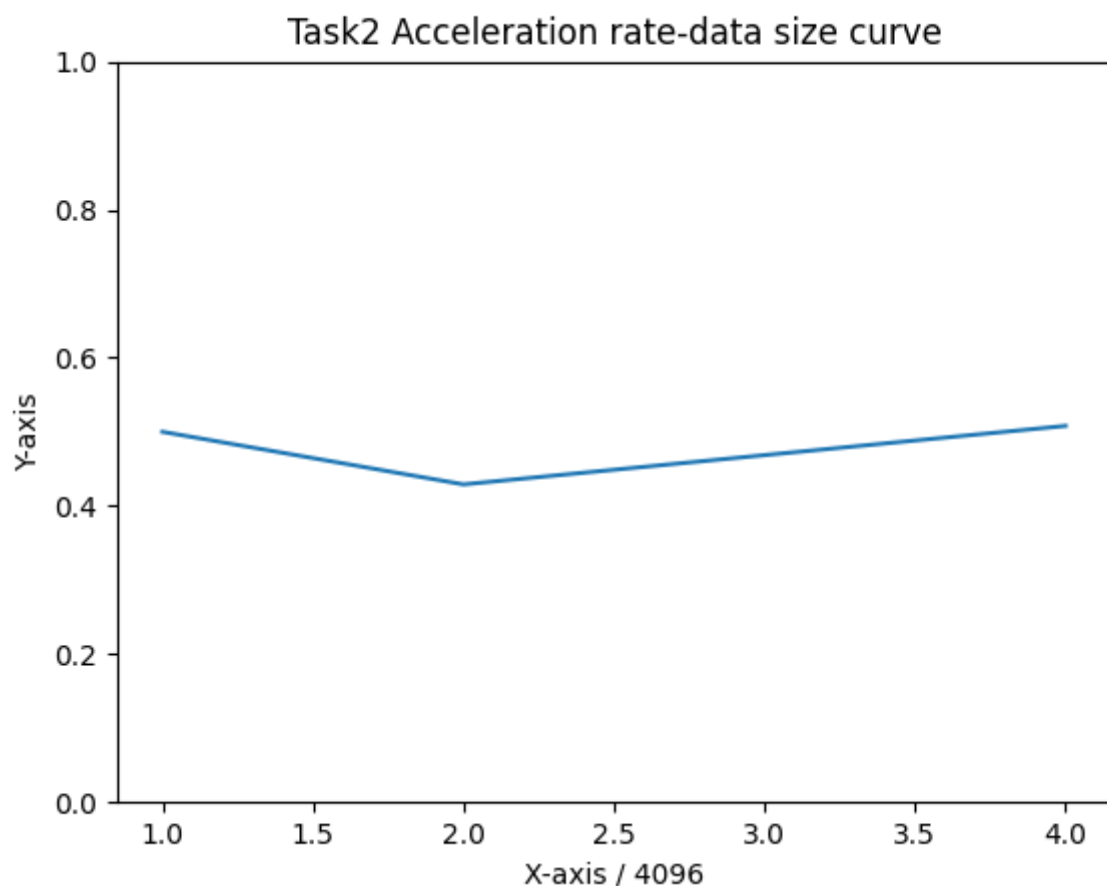
加速效果：50.8%

数据更大之后内存无法容纳。

列出数据表格如下：

数据大小 N	加速前执行时间 / milliseconds	加速后执行时间 / milliseconds	加速效果
4096	40	20	50%
4096 x 2	168	90	42.9%
4096 x 4	749	368	50.8%

使用 python 的 matplotlib 作出加速效果-数据大小曲线图如下：



可以看出，任务2的加速效果大致在50%左右浮动，加速效果较好。

对SIMD指令拓展的理解

SIMD 与 AVX 的基本概念

SIMD (Single Instruction, Multiple Data) 是一种并行处理技术，是使用单个指令同时在多个数据元素上执行相同的操作。SIMD指令通常是硬件架构的一部分，比如这里的AVX就是x86架构的一部分，可以用于在涉及大量数据的应用中进行并行处理。

在C++中，SIMD 指令通过 SIMD 内联函数提供了一种在代码中直接使用 SIMD 指令的方式。SIMD 内联函数是特殊的函数，它们映射到底层硬件架构支持的特定 SIMD 指令。它们允许程序员直接编写矢量化代码，而无需使用汇编语言。

AVX2 (Advanced Vector Extensions 2) 是 Intel 针对 x86 架构推出的 SIMD 指令集的一部分。AVX2 提供了更宽的矢量寄存器和一些新的指令，用于执行并行操作，从而提高向量化代码的性能。

AVX2 的底层实现是通过硬件支持的特殊矢量寄存器和相应的指令来完成的。

在编写使用 AVX2 指令的代码时，程序员通常使用相应的内联函数来表示这些指令。这些内联函数被编译器映射到底层的机器指令，以便在相应的硬件上执行。

AVX2 这种 SIMD 指令拓展实现加速的一些关键原因：

1.AVX2 引入了 256 位的寄存器，宽度很大。这样可以在一个指令周期内同时处理更多的数据元素，提高了计算密集型任务的吞吐量。

2.AVX2 提供了更多丰富的指令集，涵盖了更多的数学和逻辑操作。这些操作包括矢量加法、矢量乘法、矢量逻辑运算等，使得在一个指令中可以执行更复杂的计算。

3.SIMD 允许单个指令同时作用于多个数据元素。这种数据级并行性质使得相同的操作可以在多个数据上同时执行，从而提高了计算效率。

4.降低内存带宽需求： SIMD 可以减少内存带宽的需求，因为一个 SIMD 指令可以加载或存储多个数据元素。

5.向量化编程： AVX2 提供了更高级别的向量化编程能力，使得开发者可以更容易地利用硬件的并行计算能力。向量化编程允许使用高级语言编写代码，而不必直接使用低级的汇编语言。

SIMD 指令拓展的特点与优势：

1.SIMD 指令集引入了特殊的向量寄存器，这些寄存器能够同时存储和处理多个数据元素。这些向量寄存器的宽度决定了每个指令可以操作的数据元素数量。

2.SIMD 指令允许执行相同的操作，但是同时作用于多个数据元素。这种并行计算的方式适用于那些可以被分解为相互独立、相同的操作的任务，例如向量加法、矩阵乘法等。

3.SIMD 操作在指令级别上实现了并行性。相同的指令被应用于多个数据元素，从而在一个指令周期内完成多次操作。这提高了计算的吞吐量，特别是在处理大量数据时。

4.SIMD 指令集通常包含各种指令，用于执行不同类型的计算任务，例如数学运算、逻辑运算、位操作等。这使得开发者可以选择合适的指令来优化特定的计算任务。

5.SIMD 指令集的拓展需要硬件的支持。现代 CPU 和 GPU 往往集成了对 SIMD 的支持，这意味着开发者可以在不使用低级汇编语言的情况下，通过高级语言编写 SIMD 代码。

6.SIMD 指令集的使用可以用于提升代码性能，特别是在大规模数据计算任务中。

总体而言，SIMD 指令集拓展是通过硬件提供的一种机制，使得在单个指令中能够同时处理多个数据元素，从而实现并行计算，提高计算性能。