进阶实验篇第5章

矩阵乘：面向硬件加速器的优化

第一部分：SIMD

1 单指令多数据（SIMD）指令原理

2 任务实现：基于SIMD指令的矩阵乘法

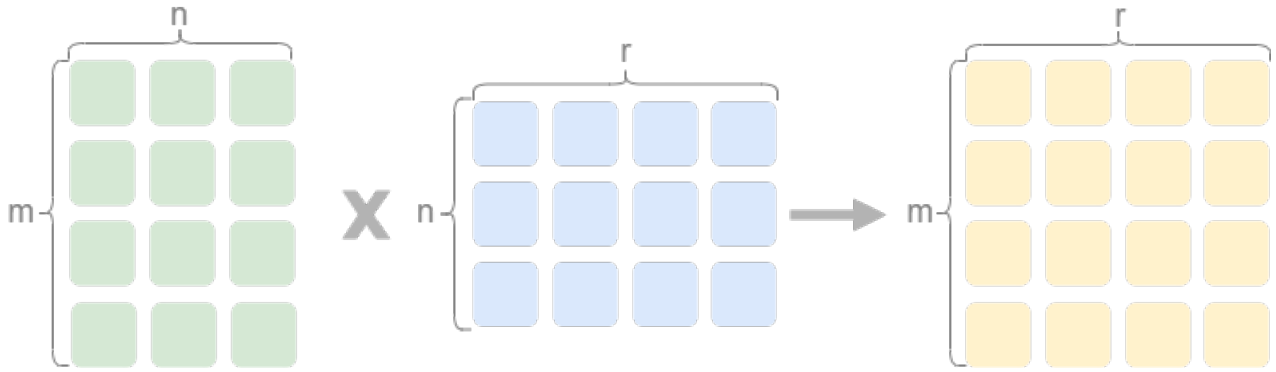
3 延伸阅读：面向SIMD的编译优化和智能优化

单指令多数据（SIMD）

指令原理

单指令多数据（SIMD）指令原理---回顾

```
...
for (int i=0; i<m; i++) {
    for (int j=0; j<r; j++) {
        for (int k=0; k<n; k++) {
            matrix[i][j] += matrix_1[i][k] * matrix_2[k][j];
        }
    }
}
...
```



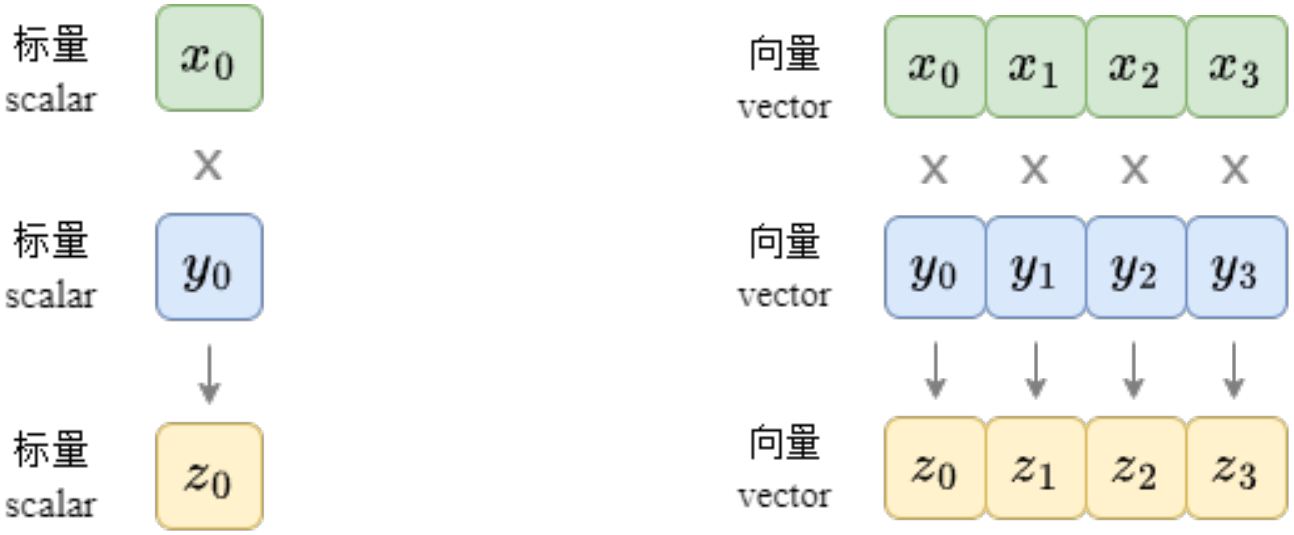
- 基于Cache局部性原理的优化
  - 降低访存开销
- 使用多线程提高并行度
  - 充分利用CPU资源
  - 同时处理数据，减少时间开销

- 硬件资源有限
- 不能减少程序指令数

当矩阵规模增大到一定规模则无法有效减少时间开销

Single Instruction Multiple Data

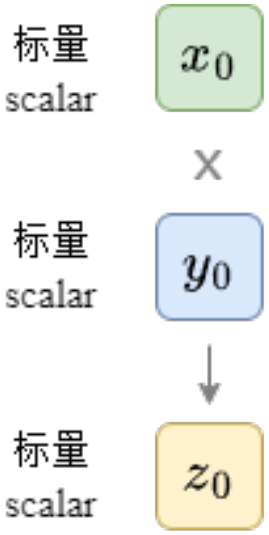
单指令多数据　　SIMD



一般乘法　　SIMD乘法

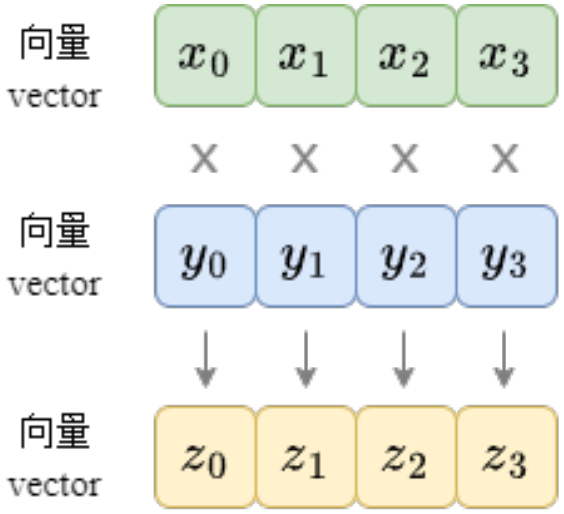
Single Instruction Multiple Data

单指令多数据



一般乘法

SIMD



SIMD乘法

龙芯 LSX 指令集

Loongson SIMD Extension

Intrinsic函数：

```
__m128 __lsx_vfmul_s (__m128 _1,__m128 _2);
```

指令：

```
vfmul.s vd,vj,vk
```

DST SRC

```
DST[31:0] := SRC1[31:0] * SRC2[31:0]
DST[63:32] := SRC1[63:32] * SRC2[63:32]
DST[95:64] := SRC1[95:64] * SRC2[95:64]
DST[127:96] := SRC1[127:96] * SRC2[127:96]
```

单指令多数据（SIMD）指令原理---SIMD优化矩阵乘示例

```
...
for (int i=0; i<m; i++) {
    for (int j=0; j<r; j++) {
        for (int k=0; k<n; k++) {
            matrix[i][j] += matrix_1[i][k] * matrix_2[k][j];
        }
    }
}
...
```

循环展开：发掘程序并行性

```
...
for (int i=0; i<m; i++) {
    for (int j=0; j<r; j++) {
        for (int k=0; k<n; k+=4) {
            matrix[i][j] += matrix_1[i][k] * matrix_2[k][j];
            matrix[i][j] += matrix_1[i][k+1] * matrix_2[k+1][j];
            matrix[i][j] += matrix_1[i][k+2] * matrix_2[k+2][j];
            matrix[i][j] += matrix_1[i][k+3] * matrix_2[k+3][j];
        }
    }
}
...
```

```
#include <lsxintrin.h>
...
for (int i=0; i<m; i++) {
    for (int j=0; j<r; j++) {
        for (int k=0; k<n; k+=4) {
            __lsx_vst(__lsx_vfmul_s(
                (__m128)__lsx_vld((float*)(matrix_1[i]+k), 0),
                __lsx_setr_s(matrix_2[k][j], matrix_2[k+1][j],
            matrix_2[k+2][j], matrix_2[k+3][j])), tmp, 0);
            matrix[i][j] += tmp[0] + tmp[1] + tmp[2] + tmp[3];
        }
    }
}
...
```

使用龙芯LSX Intrinsic函数

任务实现

基于SIMD指令的矩阵乘法

任务实现：基于SIMD指令的矩阵乘法---代码解析

```
...
for (int i=0; i<m; i++) {
    for (int j=0; j<r; j++) {
        for (int k=0; k<n; k++) {
            matrix[i][j] += matrix_1[i][k] * matrix_2[k][j];
        }
    }
}
...
```

循环展开

```
...
for (int i=0; i<m; i++) {
    for (int j=0; j<r; j++) {
        for (int k=0; k<n; k+=4) {
            matrix[i][j] += matrix_1[i][k] * matrix_2[k][j];
            matrix[i][j] += matrix_1[i][k+1] * matrix_2[k+1][j];
            matrix[i][j] += matrix_1[i][k+2] * matrix_2[k+2][j];
            matrix[i][j] += matrix_1[i][k+3] * matrix_2[k+3][j];
        }
    }
}
...
```

$$matrix = Z, matrix\_1 = X, matrix\_2 = Y$$

$$Z = X \times Y$$

$$z_{ij} = \sum_{k=0}^{n-1} x_{ik} y_{kj}$$

$$z_{ij} = \sum_{q=0}^{\frac{n}{4}-1} (x_{i(4q)} y_{(4q)j} + x_{i(4q+1)} y_{(4q+1)j} + x_{i(4q+2)} y_{(4q+2)j} + x_{i(4q+3)} y_{(4q+3)j})$$

一般矩阵乘原理

循环展开及SIMD优化后矩阵乘原理

任务实现：基于SIMD指令的矩阵乘法---代码解析

```
...
for (int i=0; i<m; i++) {
    for (int j=0; j<r; j++) {
        for (int k=0; k<n; k+=4) {
            matrix[i][j] += matrix_1[i][k] * matrix_2[k][j];
            matrix[i][j] += matrix_1[i][k+1] * matrix_2[k+1][j];
            matrix[i][j] += matrix_1[i][k+2] * matrix_2[k+2][j];
            matrix[i][j] += matrix_1[i][k+3] * matrix_2[k+3][j];
        }
    }
}
...
```

SIMD

```
#include <lsxintrin.h>
...
for (int i=0; i<m; i++) {
    for (int j=0; j<r; j++) {
        for (int k=0; k<n; k+=4) {
            __lsx_vst(__lsx_vfmul_s(
                (__m128)__lsx_vld((float*)(matrix_1[i]+k), 0),
                __lsx_setr_s(matrix_2[k][j], matrix_2[k+1][j],
                matrix_2[k+2][j], matrix_2[k+3][j])), tmp, 0);
            matrix[i][j] += tmp[0] + tmp[1] + tmp[2] + tmp[3];
        }
    }
}
...
```

$$z_{ij} = \sum_{q=0}^{\frac{n}{4}-1} \left(x_{i(4q)}y_{(4q)j} + x_{i(4q+1)}y_{(4q+1)j} + x_{i(4q+2)}y_{(4q+2)j} + x_{i(4q+3)}y_{(4q+3)j}\right)$$

循环展开用4条乘法指令完成，
SIMD仅用1条乘法指令完成

任务实现：基于SIMD指令的矩阵乘法---代码解析

```
#include <lsxintrin.h>
...
for (int i=0; i<m; i++) {
    for (int j=0; j<r; j++) {
        for (int k=0; k<n; k+=4) {
            __lsx_vst(__lsx_vfmul_s(
                (__m128)__lsx_vld((float*)(matrix_1[i]+k), 0),
                __lsx_setr_s(matrix_2[k][j], matrix_2[k+1][j],
            matrix_2[k+2][j], matrix_2[k+3][j])), tmp, 0);
            matrix[i][j] += tmp[0] + tmp[1] + tmp[2] + tmp[3];
        }
    }
}
...
```

?

补齐代码 →

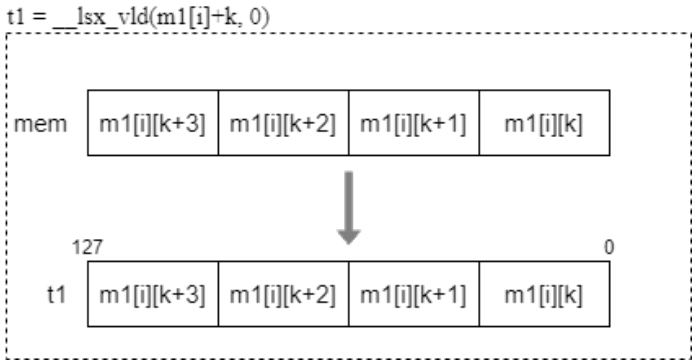
任务实现：基于SIMD指令的矩阵乘法---代码解析

```
#include <lsxintrin.h>
void lsx_matrix_multiplication(float**matrix, float**matrix_1, float**matrix_2,
    int m, int n, int r) {
float*tmp = (float*)malloc(4*sizeof(float));
for (int i=0; i<4; i++) tmp[i] = 0.f;
for (int i=0; i<m; i++) {
    for (int j=0; j<r; j++) {
        for (int k=0; k<n; k+=4) {
            __lsx_vst(__lsx_vfmul_s(
                (__m128)__lsx_vld((float*)(matrix_1[i]+k), 0),
                __lsx_setr_s(matrix_2[k][j], matrix_2[k+1][j],
            matrix_2[k+2][j], matrix_2[k+3][j])), tmp, 0);
            matrix[i][j] += tmp[0] + tmp[1] + tmp[2] + tmp[3];
        }
    }
}
}
```

结果矩阵　第一个矩阵　第二个矩阵

用于临时存储四个积，为加法做准备

任务实现：基于SIMD指令的矩阵乘法---代码解析




```
#include <lsxintrin.h>
void lsx_matrix_multiplication(float**matrix, float**matrix_1, float**matrix_2,
    int m, int n, int r) {
float*tmp = (float*)malloc(4*sizeof(float));
for (int i=0; i<4; i++) tmp[i] = 0.f;
for (int i=0; i<m; i++) {
    for (int j=0; j<r; j++) {
        for (int k=0; k<n; k+=4) {
            __lsx_vst(__lsx_vfmul_s(
                (__m128)__lsx_vld((float*)(matrix_1[i]+k), 0)
                 __lsx_setr_s(matrix_2[k][j], matrix_2[k+1][j],
            matrix_2[k+2][j], matrix_2[k+3][j])), tmp, 0);
            matrix[i][j] += tmp[0] + tmp[1] + tmp[2] + tmp[3];
        }
    }
}
}
```

基地址　　偏移（12bit立即数）

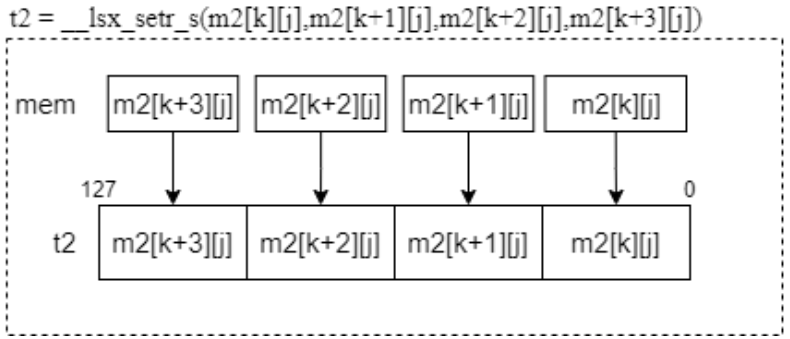
```
#define __lsx_vld(/* void* */_1, /* si12 */_2)
```

- mem_addr := 基地址 + 偏移
  dst[127:0] := MEM[mem_addr+127:mem_addr]

将128位连续内存数据载入到一个LSX寄存器

装载第一个矩阵一行中连续的四个数据

任务实现：基于SIMD指令的矩阵乘法---代码解析



__m128 __lsx_setr_s(float e3,float e2,float e1,float e0):

dst[31:0] := e3

dst[63:32] := e2

dst[95:64] := e1

dst[127:96] := e0

```
#include <lsxintrin.h>
void lsx_matrix_multiplication(float**matrix, float**matrix_1, float**matrix_2,
    int m, int n, int r) {
float*tmp = (float*)malloc(4*sizeof(float));
for (int i=0; i<4; i++) tmp[i] = 0.f;
for (int i=0; i<m; i++) {
    for (int j=0; j<r; j++) {
        for (int k=0; k<n; k+=4) {
            __lsx_vst(__lsx_vfmul_s(
                (__m128)__lsx_vld((float*)(matrix_1[i]+k), 0),
                __lsx_setr_s(matrix_2[k][j], matrix_2[k+1][j],
            matrix_2[k+2][j], matrix_2[k+3][j])), tmp, 0);
            matrix[i][j] += tmp[0] + tmp[1] + tmp[2] + tmp[3];
        }
    }
}
}
```

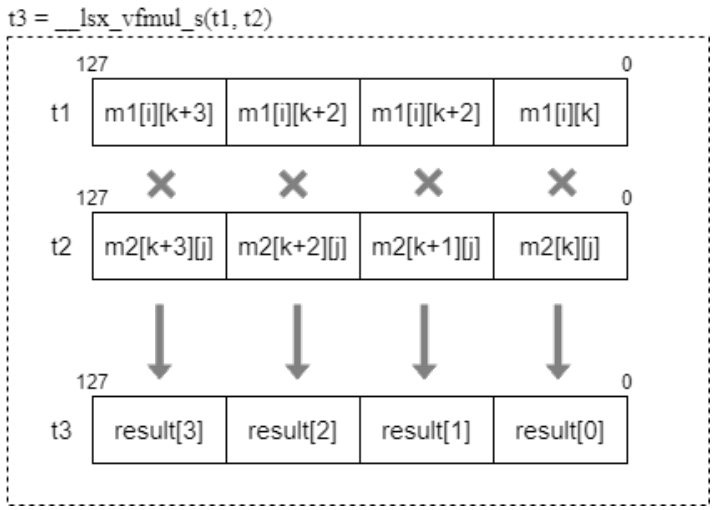
将128位不连续内存数据载入到一个LSX寄存器

装载第二个矩阵一列中连续的四个数据

为了计算方便，自定义功能为从非连续内存数据构造向量的函数：

```
__m128 __lsx_setr_s (float __A, float __B, float __C, float __D) {
    return __extension__ (__m128){ __A, __B, __C, __D };
}
```

任务实现：基于SIMD指令的矩阵乘法---代码解析

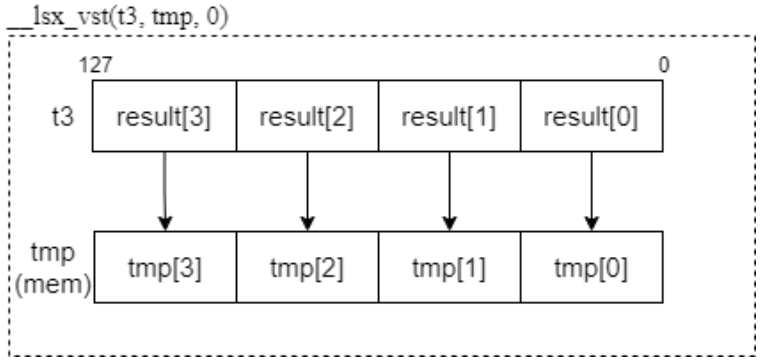



```
__m128 __lsx_vfmul_s(__m128 a,__m128 b):
  FOR j := 0 to 3
    i := j*32
    dst[i+31:i] := a[i+31:i]*b[i+31:i]
  ENDFOR
                                                    }
                                                }
```

```
#include <lsxintrin.h>
void lsx_matrix_multiplication(float**matrix, float**matrix_1, float**matrix_2,
    int m, int n, int r) {
float*tmp = (float*)malloc(4*sizeof(float));
for (int i=0; i<4; i++) tmp[i] = 0.f;
for (int i=0; i<m; i++) {
    for (int j=0; j<r; j++) {
        for (int k=0; k<n; k+=4) {
            __lsx_vst(__lsx_vfmul_s(
                (__m128)__lsx_vld((float*)(matrix_1[i]+k), 0),
                __lsx_setr_s(matrix_2[k][j], matrix_2[k+1][j],
            matrix_2[k+2][j], matrix_2[k+3][j])), tmp, 0);
            matrix[i][j] += tmp[0] + tmp[1] + tmp[2] + tmp[3];
        }
}
```

两个128位向量（LSX寄存器）中的四个数据对位相乘

任务实现：基于SIMD指令的矩阵乘法---代码解析



四个乘积存储在一个LSX寄存器中，存入连续128位内存空间中，以备他用

```
#include <lsxintrin.h>
void lsx_matrix_multiplication(float**matrix, float**matrix_1, float**matrix_2,
    int m, int n, int r) {
float*tmp = (float*)malloc(4*sizeof(float));
for (int i=0; i<4; i++) tmp[i] = 0.f;
for (int i=0; i<m; i++) {
    for (int j=0; j<r; j++) {
        for (int k=0; k<n; k+=4) {
            __lsx_vst(__lsx_vfmul_s(
                (__m128)__lsx_vld((float*)(matrix_1[i]+k), 0),
                __lsx_setr_s(matrix_2[k][j], matrix_2[k+1][j],
            matrix_2[k+2][j], matrix_2[k+3][j])), tmp, 0);
            matrix[i][j] += tmp[0] + tmp[1] + tmp[2] + tmp[3];
        }
    }
}
}
```

乘积累加到结果矩阵相应位置

LSX向量　　基地址　　偏移（12bit立即数）

```
#define __lsx_vst(/* __m128i */_1, /* void* */_2, /* si12 */_3)
```

mem_addr := 基地址 + 偏移
MEM[mem_addr+127:mem_addr] := _1[127:0]

任务实现：基于SIMD指令的矩阵乘法---拓展

```
...
for (int i=0; i<m; i++) {
    for (int j=0; j<r; j++) {
        for (int k=0; k<n; k+=4) {
            matrix[i][j] += matrix_1[i][k] * matrix_2[k][j];
            matrix[i][j] += matrix_1[i][k+1] * matrix_2[k+1][j];
            matrix[i][j] += matrix_1[i][k+2] * matrix_2[k+2][j];
            matrix[i][j] += matrix_1[i][k+3] * matrix_2[k+3][j];
        }
    }
}
...
```

SIMD

```
#include <lsxintrin.h>
...
for (int i=0; i<m; i++) {
    for (int j=0; j<r; j++) {
        for (int k=0; k<n; k+=4) {
            __lsx_vst(__lsx_vfmul_s(
                (__m128)__lsx_vld((float*)(matrix_1[i]+k), 0),
                __lsx_setr_s(matrix_2[k][j], matrix_2[k+1][j],
            matrix_2[k+2][j], matrix_2[k+3][j])), tmp, 0);
            matrix[i][j] += tmp[0] + tmp[1] + tmp[2] + tmp[3];
        }
    }
}
...
```

循环展开4次

循环展开8次

LSX的v0-v31寄存器为128位（32×4）

?

任务实现：基于SIMD指令的矩阵乘法---拓展

```
...
for (int i=0; i<m; i++) {
    for (int j=0; j<r; j++) {
        for (int k=0; k<n; k+=4) {
            matrix[i][j] += matrix_1[i][k] * matrix_2[k][j];
            matrix[i][j] += matrix_1[i][k+1] * matrix_2[k+1][j];
            matrix[i][j] += matrix_1[i][k+2] * matrix_2[k+2][j];
            matrix[i][j] += matrix_1[i][k+3] * matrix_2[k+3][j];
        }
    }
}
...
```

SIMD

```
#include <lsxintrin.h>
...
for (int i=0; i<m; i++) {
    for (int j=0; j<r; j++) {
        for (int k=0; k<n; k+=4) {
            __lsx_vst(__lsx_vfmul_s(
                (__m128)__lsx_vld((float*)(matrix_1[i]+k), 0),
                __lsx_setr_s(matrix_2[k][j], matrix_2[k+1][j],
            matrix_2[k+2][j], matrix_2[k+3][j])), tmp, 0);
            matrix[i][j] += tmp[0] + tmp[1] + tmp[2] + tmp[3];
        }
    }
}
...
```

循环展开4次

LSX的v0-v31寄存器为128位（32×4）

循环展开8次

v0-v31扩展为LASX 256位的x0-x31寄存器

Loongson Advance SIMD Extension

任务实现：基于SIMD指令的矩阵乘法---性能测试

实验对比：

- 一般矩阵乘
- LSX优化矩阵乘
- LASX优化矩阵乘

三种方法在不同矩阵规模下的时间开销

测试代码：https://github.com/NKU-EmbeddedSystem/OSbook/tree/main/lab_advanced/lab_SIMD/test_loongson.c

任务实现：基于SIMD指令的矩阵乘法---性能测试

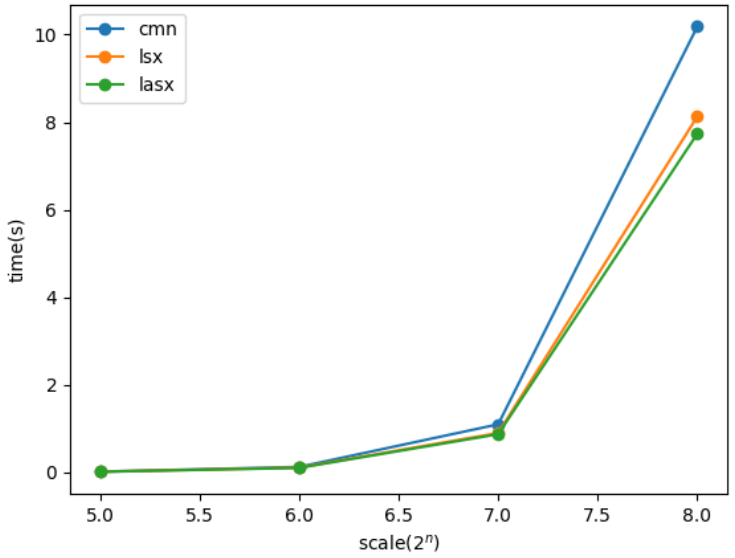


表 2: 时间开销

	128	256	512	1024
CMN	0.01522710	0.12287870	1.09685050	10.1845409
LSX	0.01453700	0.10806310	0.8990675	8.1297082
LASX	0.01268790	0.10143160	0.87148850	7.73629420

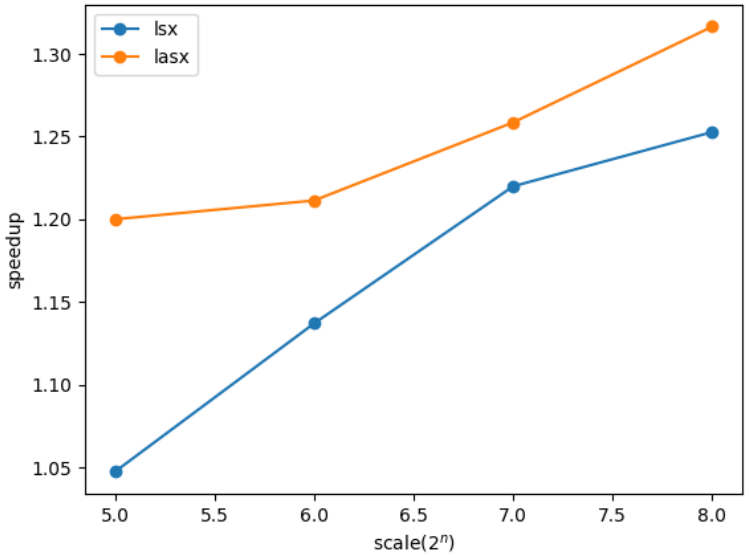


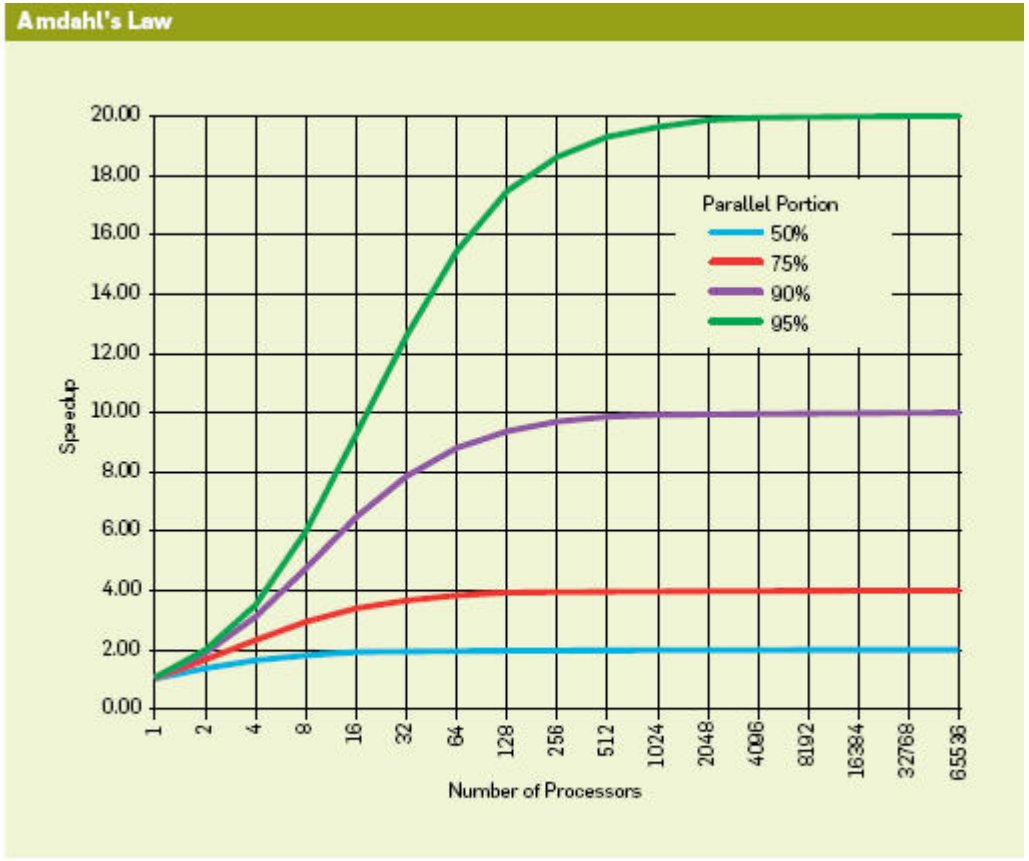
表 3: 加速比

	128	256	512	1024
CMN/LSX	1.0475	1.1371	1.2200	1.2528
CMN/LASX	1.2001	1.2114	1.2586	1.3165
LSX/LASX	1.1457	1.0654	1.0316	1.0509

在当前测试范围内：

- 经 SIMD 优化后的执行速度均高于一般矩阵乘（加速比均大于 1），所以SIMD 能优化矩阵乘
- LSX 优化最高加速比约为 1.25，LASX 优化最高加速比约为 1.32，LASX 优化后执行时间比 LSX 短

任务实现：基于SIMD指令的矩阵乘法---性能测试



Named after computer architect Gene Amdahl, Amdahl's Law is frequently used in parallel programming to predict the theoretical maximum speedup using multiple processors.

并行计算领域著名的Amdahl定律：

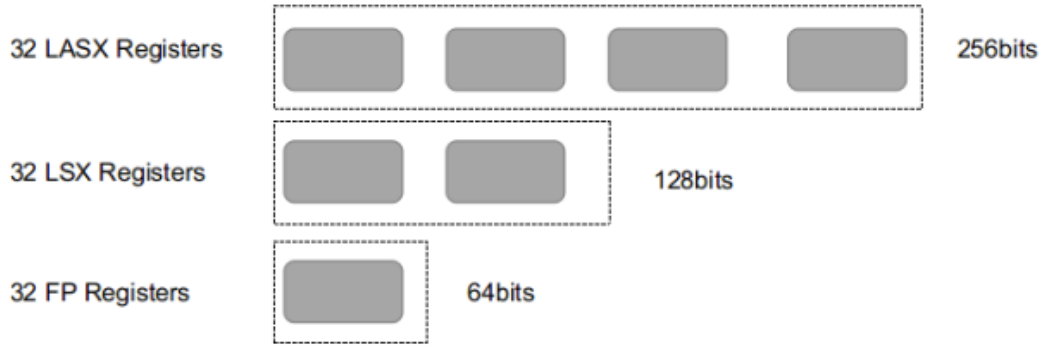
- 只有可以被优化措施影响到的部分，才是可以获取性能提升的区域
- 优化效果的上限，取决于被优化部分在整个计算任务中所占的比例

延伸阅读

面向SIMD的编译优化和智能优化

SIMD指令集

- MMX/SSE/SSE2/AVX/……(Intel)
- NEON(ARM)
- 3DNow!(AMD)
- LSX, LASX(Loongson 龙芯)
- ……



LSX与LASX涉及寄存器

如何使用SIMD扩展优化程序性能

- **汇编语言**
  - 不易使用，可移植性低
- **共享库**
  - 仅优化了部分特定算法
- **向量化编译器**
  - 为保证程序正确性，通常只进行保守优化，难以达到理想效果
  - 如GCC添加-ftree-vectorize编译选项后可自动使用SIMD优化程序

延伸阅读：面向SIMD的编译优化和智能优化

如何使用SIMD扩展优化程序性能

- **汇编语言**
  - 不易使用，可移植性低 → 编译器内建函数（intrinsics）
- **共享库**
  - 仅优化了部分特定算法
- **向量化编译器**
  - 为保证程序正确性，通常只进行保守优化，难以达到理想效果 → 借助机器学习方法的智能化编译器
  - 如GCC添加-ftree-vectorize编译选项后可自动使用SIMD优化程序

面向SIMD的智能优化

- A Survey on Compiler Autotuning using Machine Learning
- Using Machine Learning to Improve Automatic Vectorization
- ……

感谢阅读