重 庆 大 学

学 生 实 验 报 告

实验课程名称＿＿＿＿计算机组成与结构＿＿＿＿

开课实验室 ＿＿＿＿DS1501＿＿＿＿

学　　　院 软件学院 年级 2018 专业班 3(交换生) 班

学 生 姓 名 ＿＿都景会＿＿ 学 号 ＿＿L1800025＿＿

开 课 时 间 ＿＿2018＿＿至＿＿2019＿＿学年 第＿＿1＿＿学期

总 成 绩	
教师签名	

软件学院制

《计算机组成与结构》实验报告

开课实验室：DS1501 **2018 年 11 月 09 日**

<table>
<tr><td>学院</td><td>软件学院</td><td>年级、专业、班</td><td>交换生(大 3),大数据和软件工程</td><td>姓名</td><td>都景会</td><td>成绩</td><td></td></tr>
<tr><td>课程名称</td><td colspan="2">计算机组成与结构</td><td>实验项目名 称</td><td colspan="2">实验 6,7</td><td>指导教师</td><td>熊敏</td></tr>
<tr><td>教师评语</td><td colspan="7">教师签名：
年 月 日</td></tr>
</table>

一、实验目的

实验 6: SIMD 与 MMX 代码优化技术

实验 7: 多核程序设计

二、实验原理

实验六 C++程序内存布局, Intel inspector内存错误检测工具

Using SIMD to build a faster program and using Intel Amplifier to check the difference between SISD and SIMD.

实验七 OpenMP 多核程序设计技术, OpenMP 初体验, fork/join并行执行模式的概念, OpenMP的指令与子句, parallel 指令的用法, for指令的使用方法, sections和section指令的用法, Private子句, threadprivate 子句, shared 子句, reduction 子句, OpenMP 中的任务调度, Schedule子句用法, 静态调度(static)。

Understanding what OpenMP is and building a simple program using OpenMP

三、使用仪器、材料

VC++.

四、实验步骤

实验 6:

6.1 请使用第四章介绍的程序性能定量分析方法对SIMD代码示例中的基于SSE指令的算法和普通算法的性能进行比较。

6.2 请使用 SIMD 指令优化 float 类型的 400*400 阶方阵的矩阵乘法，并分析优化后的程序性能。

6.3 请自学内存对齐的知识

实验7:

没有习题.

五、实验过程原始记录(数据、图表、计算等)

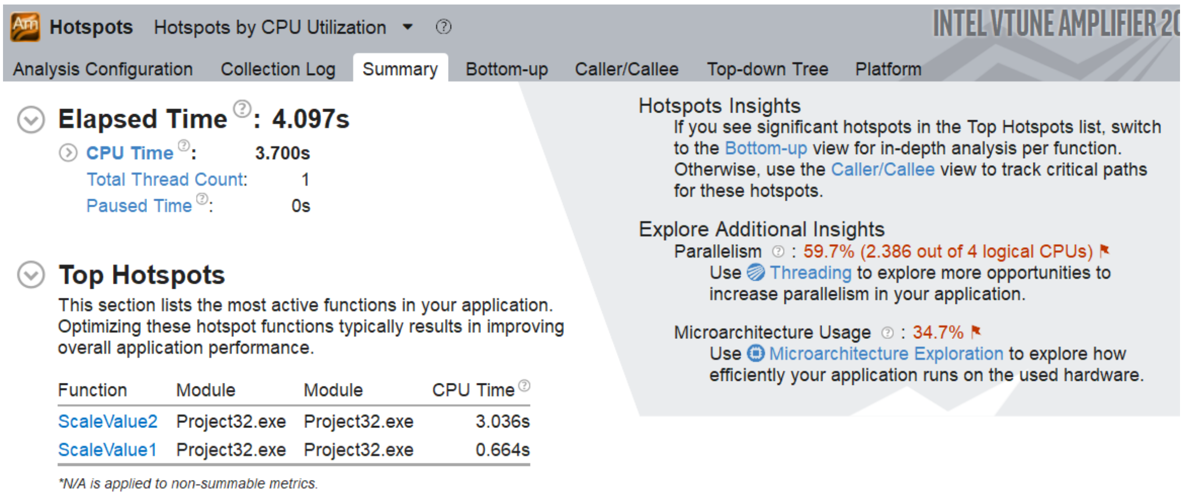
6.1

```cpp
#include <xmmintrin.h>
#include<iostream>
using namespace std;
void ScaleValue1(float *pArray, int dwCount, float fScale){
    int dwGroupCount = dwCount / 4;
    __m128 e_Scale = _mm_set_ps1(fScale);
    for (int i = 0; i < dwGroupCount; i++)
        *(__m128*)(pArray + i * 4) = _mm_mul_ps(*(__m128*)(pArray + i * 4),
            e_Scale);
}
void ScaleValue2(float *pArray, int dwCount, float fScale){
    for (int i = 0; i < dwCount; i++)
        pArray[i] *= fScale;
}
#define ARRAYCOUNT 10000
int __cdecl main(){
    float __declspec(align(16)) Array[ARRAYCOUNT];//内存对齐
    memset(Array, 0, sizeof(float) * ARRAYCOUNT);
    for (int i = 0; i < 100000; i++){
        ScaleValue1(Array, ARRAYCOUNT, 1000.0f);
    }
    for (int i = 0; i < 100000; i++){
        ScaleValue2(Array, ARRAYCOUNT, 1000.0f);
    }
    cout << "done" << endl;
    return 0;
}
```

Microsoft Visual Studio Debug Console

```
done

C:\Users\user\source\repos\Project31\Debug\Proj
Press any key to close this window . . .
```



```cpp
#include<iostream>
using namespace std;
void ScaleValue1(float *pArray, int dwCount, float fScale){
    int dwGroupCount = dwCount / 4;
    for (int i = 0; i < dwGroupCount; i++)
        *(pArray + i * 4) = *(pArray + i * 4)*(fScale);
}
void ScaleValue2(float *pArray, int dwCount, float fScale){
    for (int i = 0; i < dwCount; i++)
        pArray[i] *= fScale;
}
#define ARRAYCOUNT 10000
int main(){
    float Array[ARRAYCOUNT];//内存对齐
    for (int i = 0; i < 100000; i++){
        ScaleValue1(Array, ARRAYCOUNT, 1000.0f);
    }
    for (int i = 0; i < 100000; i++){
        ScaleValue2(Array, ARRAYCOUNT, 1000.0f);
    }
    cout << "done" << endl;
    return 0;
}
```

Microsoft Visual Studio Debug Console

```
done

C:\Users\user\source\repos\Project31\Deb
Press any key to close this window . . .
```

Hotspots Hotspots by CPU Utilization INTEL VTUNE AMPLIFIER 2019

Analysis Configuration | Collection Log | Summary | Bottom-up | Caller/Callee | Top-down Tree | Platform

Elapsed Time: 4.673s

- CPU Time: 4.087s
- Total Thread Count: 1
- Paused Time: 0s

Top Hotspots

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

Function	Module	Module	CPU Time
ScaleValue2	Project32.exe	Project32.exe	3.293s
ScaleValue1	Project32.exe	Project32.exe	0.794s

*N/A is applied to non-summable metrics.

INSIGHTS

Hotspots Insights

If you see significant hotspots in the Top Hotspots list, switch to the Bottom-up view for in-depth analysis per function. Otherwise, use the Caller/Callee view to track critical paths for these hotspots.

Explore Additional Insights

Microarchitecture Usage: 42.2%

Use Microarchitecture Exploration to explore how efficiently your application runs on the used hardware.

6.2

```cpp
#include <iostream>
using namespace std;
class Matrix {
private:
    int row; int col; float *mat; void makeMat();
public:
    friend void matMul(Matrix *a, Matrix *b, Matrix *r);
    Matrix(int r = 10, int c =10) {
        row = r; col = c; makeMat(); }
    ~Matrix() { clear(); };
    void setElem(int r, int c, float e) { mat[r*row+c] = e;  }
    void showElem(int r, int c) { cout << mat[r*row+c] << endl;  }
    void clear() { delete mat; }
};
void Matrix::makeMat() {
    mat = new float[row*col];
    for (int i = 0; i < row*col; i++)
        mat[i] = 0;
}
void matMul(Matrix *a, Matrix *b, Matrix *r) {
    for (int i = 0; i < a->row; i++)
        for (int j = 0; j < a->col; j++) {
            for (int k = 0; k < a->row; k++)
                r->mat[i*(a->row) + j] += (a->mat[i*a->row + k])
                * (b->mat[j + (k*a->row)]);
        }
}
int main() {
    Matrix a1(4, 4); Matrix a2(4, 4); Matrix a3(4, 4);
    a1.setElem(1, 3, 1); a2.setElem(3, 1, 1);
    matMul(&a1, &a2, &a3);
    a3.showElem(1, 1);
}
```

Microsoft Visual Studio Debug Console

```
1

C:\Users\user\source\repos\Project32\Debug\Projec
Press any key to close this window . . .
```

Hotspots Hotspots by CPU Utilization ⓘ INTEL VTUNE AMPLIFIER 201

Analysis Configuration | Collection Log | Summary | Bottom-up | Caller/Callee | Top-down Tree | Platform

Elapsed Time ⓘ: 0.335s

No data to show. The collected data is not sufficient.

Top Hotspots

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

No data to show. The collected data is not sufficient.

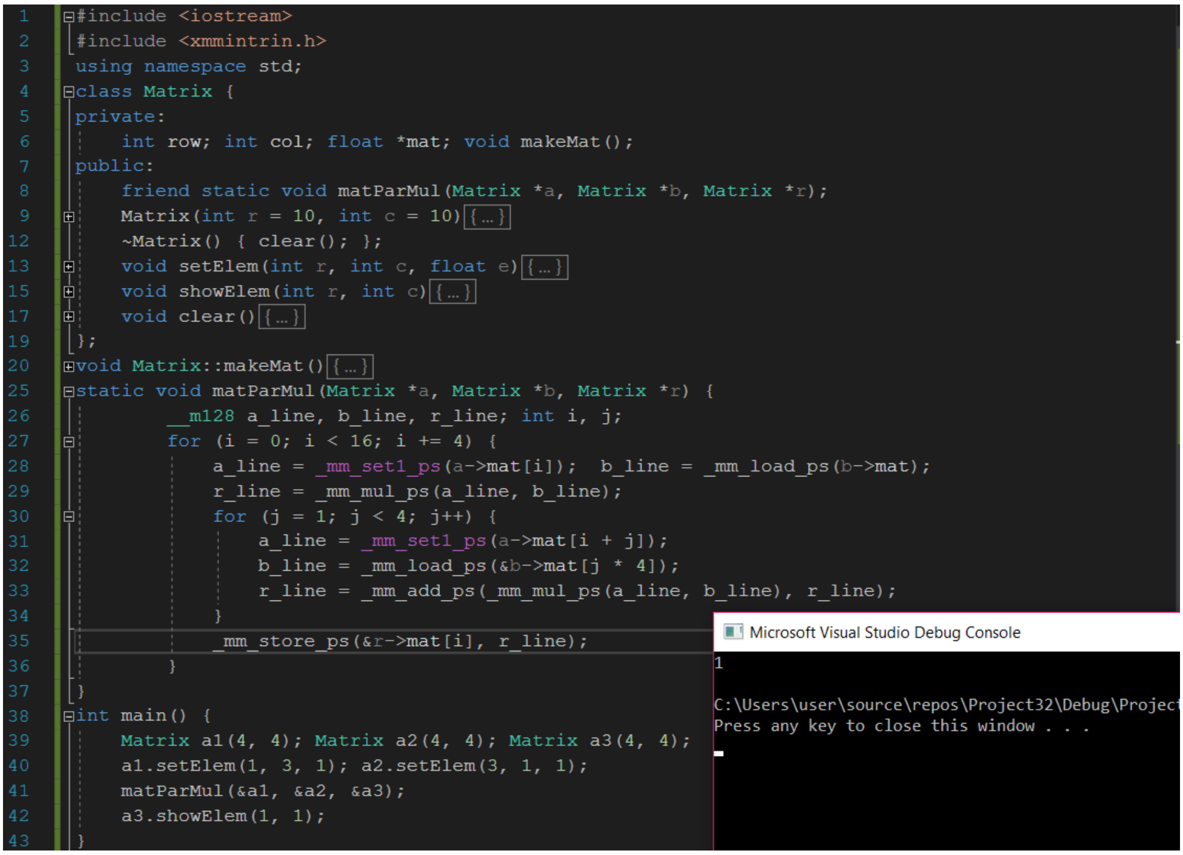
Hotspots Insights

If you see significant hotspots in the Top Hotspots list, switch to the Bottom-up view for in-depth analysis per function. Otherwise, use the Caller/Callee view to track critical paths for these hotspots.

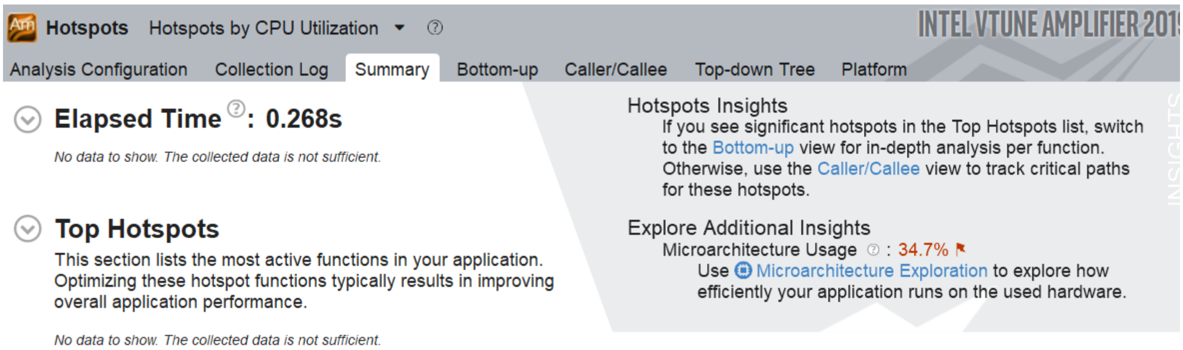
Explore Additional Insights

Microarchitecture Usage ⓘ : 35.0%

Use Microarchitecture Exploration to explore how efficiently your application runs on the used hardware.



```
#include <iostream>
#include <xmmintrin.h>
using namespace std;
class Matrix {
private:
    int row; int col; float *mat; void makeMat();
public:
    friend static void matParMul(Matrix *a, Matrix *b, Matrix *r);
    Matrix(int r = 10, int c = 10){...}
    ~Matrix() { clear(); };
    void setElem(int r, int c, float e){...}
    void showElem(int r, int c){...}
    void clear(){...}
};
void Matrix::makeMat(){...}
static void matParMul(Matrix *a, Matrix *b, Matrix *r) {
        __m128 a_line, b_line, r_line; int i, j;
        for (i = 0; i < 16; i += 4) {
            a_line = _mm_set1_ps(a->mat[i]);  b_line = _mm_load_ps(b->mat);
            r_line = _mm_mul_ps(a_line, b_line);
            for (j = 1; j < 4; j++) {
                a_line = _mm_set1_ps(a->mat[i + j]);
                b_line = _mm_load_ps(&b->mat[j * 4]);
                r_line = _mm_add_ps(_mm_mul_ps(a_line, b_line), r_line);
            }
            _mm_store_ps(&r->mat[i], r_line);
        }
}
int main() {
    Matrix a1(4, 4); Matrix a2(4, 4); Matrix a3(4, 4);
    a1.setElem(1, 3, 1); a2.setElem(3, 1, 1);
    matParMul(&a1, &a2, &a3);
    a3.showElem(1, 1);
}
```

6.3

```cpp
#include <omp.h>
#include <iostream>
#include <sstream>
int main(int argc, char *argv[]) {
    int th_id, nthreads;
#pragma omp parallel private(th_id)
    {
        th_id = omp_get_thread_num();
        std::ostringstream ss;
        ss << "Hello World : Thread " << th_id << std::endl;
        std::cout << ss.str();
#pragma omp barrier
#pragma omp master
        {
            nthreads = omp_get_num_threads();
            std::cout << nthreads << " Threads in total" << std::endl;
        }
    }
    return 0;
}
```

Microsoft Visual Studio Debug Console

```
Hello World : Thread 0
1 Threads in total
```

Figure 1. How programmers see memory

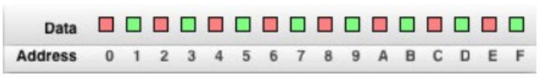


Figure 2. How processors see memory

Figure 3. Single-byte memory access granularity

Figure 4. Double-byte memory access granularity

Figure 5. Quad-byte memory access granularity

Figure 6. How processors handle unaligned memory access

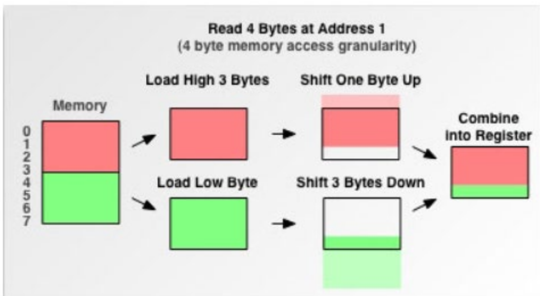


Figure 7. Single-byte access versus double-byte access

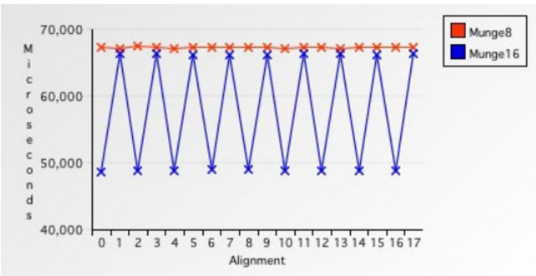


Figure 8. Single- versus double- versus quad-byte access

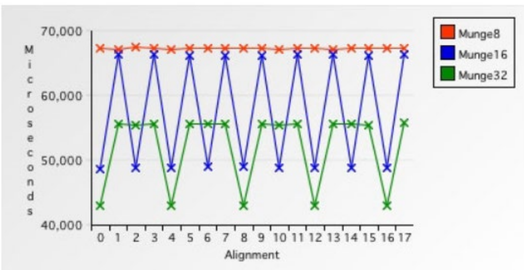


Figure 10. Multiple-byte access comparison #2

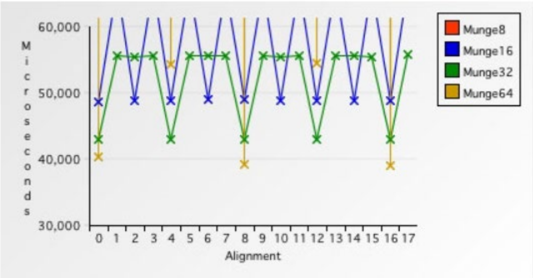


Figure 11. Multiple-byte access comparison #3

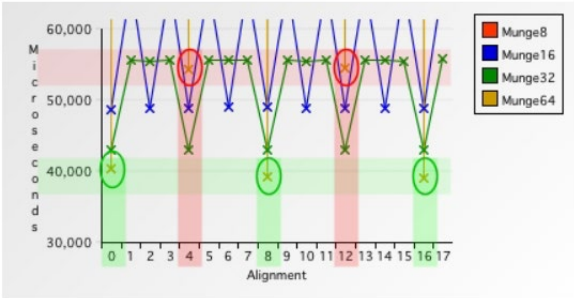


Table 2. Structure with compiler padding

Field Type	**Field Name**	**Field Offset**	**Field Size**	**Field End**
`char`	a	0	1	1
	padding	1	1	2
`long`	b	2	4	6
`char`	c	6	1	7
	padding	7	1	8
Total Size in Bytes:				8

Conclusion

If you don't understand and explicitly code for data alignment:

- Your software may hit performance-killing unaligned memory access exceptions, which invoke *very* expensive alignment exception handlers.
- Your application may attempt to atomically store to an unaligned address, causing your application to lock up.
- Your application may attempt to pass an unaligned address to Altivec, resulting in Altivec reading from and/or writing to the wrong part of memory, silently corrupting data or yielding incorrect results.

六、实验结果及分析

6.1

Elapsed Time: 4.143s Elapsed Time: 4.673s

6.2

Elapsed Time: 0.335s Elapsed Time: 0.268s

6.3

Microsoft Visual Studio Debug Console

```
Hello World : Thread 0
1 Threads in total
```

Conclusion

If you don't understand and explicitly code for data alignment:

- Your software may hit performance-killing unaligned memory access exceptions, which invoke *very* expensive alignment exception handlers.
- Your application may attempt to atomically store to an unaligned address, causing your application to lock up.
- Your application may attempt to pass an unaligned address to Altivec, resulting in Altivec reading from and/or writing to the wrong part of memory, silently corrupting data or yielding incorrect results.

七、部分作业答案

6.1

Using SIMD technique saved 0.53 seconds.

6.2

Using SIMD technique saved 0.067 seconds.