山东大学 计算机科学与技术 学院

汇编语言 课程实验报告

学号：202200130053	姓名：陈红瑞	班级：3 班
实验题目：实验 8：GCC 内联汇编优化		
实验学时：2	实验日期：20241202	
实验目的：掌握 AT&T 语法下的 AMD64 汇编编写。掌握 SIMD 指令的使用，并对程序进行向量化优化。掌握 C 与内联汇编联合编程的方法。		
实验环境：Windows11、DOSBox-0.74、Masm64		
源程序清单： vector.c		

编译及运行结果：

如图，这里定义了 COMPUTE_KERNEL，用于进行一系列的计算，并将最终的计算结果保存到数组 dest 中。

```
uint32_t src0[ARRAY_SIZE];
uint32_t src1[ARRAY_SIZE];
uint32_t dest[ARRAY_SIZE];

#define COMPUTE_KERNEL() \
do \
{ \
    uint32_t temp; \
    temp = src0[i] * 0x12345678; \
    temp += src1[i] * 0x76543210; \
    temp *= 0xA0A00505; \
    dest[i] = temp + src1[i]; \
} \
while(0)
```

这里再定义了函数 rdtsc 和 checksum，分别用于计算经过的时钟周期数以及对最终得到的 dest 数组的校验和。

```c
uint64_t rdtsc(void)
{
    uint32_t lo, hi;
    __asm__ __volatile__ ("rdtsc" : "=a" (lo), "=d" (hi));
    return ((uint64_t)hi << 32) | lo;
}

uint32_t checksum(void)
{
    uint32_t sum = 0;

    for (int i = 0; i < ARRAY_SIZE; i++)
        sum += dest[i];

    return sum;
}
```

下面再通过编译器自带的优化指令来对前面宏定义的部分进行优化。这里分别使用了不同类型的优化，分别为 O0，O2 以及 O3，其中 O3 的优化包括使用 sse 指令集和 avx 指令集。

```c
void __attribute__((optimize("O0"))) raw_calc_native(void)
{
    for (int i = 0; i < ARRAY_SIZE; i++)
        COMPUTE_KERNEL();
}

void __attribute__((optimize("O2"))) raw_calc_expert(void)
{
    for (int i = 0; i < ARRAY_SIZE; i++)
        COMPUTE_KERNEL();
}

void __attribute__((optimize("O3"))) raw_calc_sse(void)
{
    for (int i = 0; i < ARRAY_SIZE; i++)
        COMPUTE_KERNEL();
}

void __attribute__((optimize("O3"), __target__("arch=core-avx2"))) raw_calc_avx_auto(void)
{
    for (int i = 0; i < ARRAY_SIZE; i++)
        COMPUTE_KERNEL();
}
```

下面是实现内联汇编的部分。

这里使用的是 avx 指令集，并使用 O3 进行优化。这里的注释部分是

不进行循环展开的实现过程，这里在 RCX 中存储循环的计数值，在 RBX，RSI，RDI 中分别存储 src0，src1，dest 数组的地址，然后使用 vpbroadcastd 指令将 const0，const1，const2 的值广播为向量，然后 vpmulld 指令实现对 src0 和 src1 中的一组值同时与前面广播得到的向量同时进行计算，然后再将计算完的结果写到 dest 对应的内存中。由于 ymm 中每次处理的数据为 32 位，因此 CX 需要将这个值加 0x20，并进入下一次的循环部分。

```
void __attribute__((optimize("O3"), __target__("arch=core-avx2"))) raw_calc_avx_manual(void)
{
    const uint32_t const0 = 0x12345678;
    const uint32_t const1 = 0x76543210;
    const uint32_t const2 = 0xA0A00505;
    const uint64_t limit = ARRAY_SIZE * 4;

    /* No unroll
    __asm__ __volatile__
    (
        "xor          %%rcx,              %%rcx\n"            //rcx contains the counter
        "lea          %[src0],              %%rbx \n"         //rbx/rsi contains src
        "lea          %[src1],              %%rsi \n"
        "lea          %[dest],              %%rdi \n"         //rdi contains dest
        "vpbroadcastd  %[const0],              %%ymm13\n"     //ymm 13,14 and 15 contains the constants
        "vpbroadcastd  %[const1],              %%ymm14\n"
        "vpbroadcastd  %[const2],              %%ymm15\n"
        "1:\n"
        "vpmulld      (%%rbx,%%rcx,1),      %%ymm13,      %%ymm0\n"  //temp=src0[i] * 0x12345678;
        "vmovdqu      (%%rsi,%%rcx,1),      %%ymm1\n"               //temp += src1[i] * 0x76543210;
        "vpmulld      %%ymm1,              %%ymm14,      %%ymm2\n"
        "vpaddd      %%ymm0,              %%ymm2,      %%ymm2\n"
        "vpmulld      %%ymm2,              %%ymm15,      %%ymm2\n"  //temp *= 0xA0A00505;
        "vpaddd      %%ymm2,              %%ymm1,      %%ymm2\n"  //dest[i] = temp + src1[i];
        "vmovdqu      %%ymm2,              (%%rdi,%%rcx,1)\n"
        "add          $0x20,              %%rcx\n"
        "cmp          %[limit],              %%rcx\n"
        "jne          1b\n"
        :[dest]"=m" (dest)
        :[src0]"m" (src0),[src1]"m" (src1),[const0]"m"(const0),[const1]"m"(const1),[const2]"m"(const2),[limit]"r"(limit)
        :"%rbx","%rcx","%rsi","%rdi","memory","cc"
    ); */
```

下面是实现循环展开后的汇编代码。

这里在 AX DX BP 分别存储计数值，其中 CX 从 0 开始，这三个寄存器分别从 0x20，0x40，0x60 开始，然后在计算的部分，除了前面按照 CX 的偏移量取出对应的值外，还需要将 AX DX BP 分别作为偏移量，得到 4 个不同的向量进行计算，并使用了不同的 ymm 寄存器，如先计算 src0[i]

* 0x12345678，这里会依次用到 ymm0, ymm3, ymm6, ymm9 分别存储四次计算得到的 temp 的结果。由于这里将 4 次循环进行展开，因此，CX AX DX BP 在循环的最后需要加 0x80 再进入下一次循环。

```
/* Manual unroll with data dependency interleaving */
__asm__ __volatile__
(
    "xor          %%rcx,              %%rcx\n"          //rcx contains the counter
    "mov          $0x20,              %%rax\n"          //rax contains the second counter
    "mov          $0x40,              %%rdx\n"          //rbx contains the third counter
    "mov          $0x60,              %%rbp\n"          //rbp contains the fourth counter
    "lea          %[src0],              %%rbx \n"         //rbx/rsi contains src
    "lea          %[src1],              %%rsi \n"
    "lea          %[dest],              %%rdi \n"         //rdi contains dest
    "vpbroadcastd  %[const0],              %%ymm13\n"     //ymm13,14 and 15 contains the constant
    "vpbroadcastd  %[const1],              %%ymm14\n"
    "vpbroadcastd  %[const2],              %%ymm15\n"

    "1:\n"
    //interleave unroll - rcx/0,1,2 rax/3,4,5 rdx/6,7,8 rbp/9,10,11
    "vpmulld      (%%rbx,%%rcx,1),      %%ymm13,      %%ymm0\n"  //temp=src0[i] * 0x12345678;
    "vpmulld      (%%rbx,%%rax,1),      %%ymm13,      %%ymm3\n"
    "vpmulld      (%%rbx,%%rdx,1),      %%ymm13,      %%ymm6\n"
    "vpmulld      (%%rbx,%%rbp,1),      %%ymm13,      %%ymm9\n"

    "vmovdqu      (%%rsi,%%rcx,1),      %%ymm1\n"               //temp += src1[i] * 0x76543210;
    "vpmulld      %%ymm1,               %%ymm14,      %%ymm2\n"
    "vmovdqu      (%%rsi,%%rax,1),      %%ymm4\n"
    "vpmulld      %%ymm4,               %%ymm14,      %%ymm5\n"
    "vmovdqu      (%%rsi,%%rdx,1),      %%ymm7\n"
    "vpmulld      %%ymm7,               %%ymm14,      %%ymm8\n"
    "vmovdqu      (%%rsi,%%rbp,1),      %%ymm10\n"
    "vpmulld      %%ymm10,              %%ymm14,      %%ymm11\n"

    "vpaddd      %%ymm0,               %%ymm2,      %%ymm2\n"
    "vpaddd      %%ymm3,               %%ymm5,      %%ymm5\n"
    "vpaddd      %%ymm6,               %%ymm8,      %%ymm8\n"
    "vpaddd      %%ymm9,               %%ymm11,     %%ymm11\n"
```

```
"vpaddd      %%ymm0,             %%ymm2,      %%ymm2\n"
"vpaddd      %%ymm3,             %%ymm5,      %%ymm5\n"
"vpaddd      %%ymm6,             %%ymm8,      %%ymm8\n"
"vpaddd      %%ymm9,             %%ymm11,     %%ymm11\n"

"vpmulld      %%ymm2,             %%ymm15,      %%ymm2\n"  //temp *= 0xA0A00505;
"vpmulld      %%ymm5,             %%ymm15,      %%ymm5\n"
"vpmulld      %%ymm8,             %%ymm15,      %%ymm8\n"
"vpmulld      %%ymm11,            %%ymm15,      %%ymm11\n"

"vpaddd      %%ymm2,             %%ymm1,      %%ymm2\n"  //dest[i] = temp + src1[i];
"vmovdqu      %%ymm2,             (%%rdi,%%rcx,1)\n"
"vpaddd      %%ymm5,             %%ymm4,      %%ymm5\n"
"vmovdqu      %%ymm5,             (%%rdi,%%rax,1)\n"
"vpaddd      %%ymm8,             %%ymm7,      %%ymm8\n"
"vmovdqu      %%ymm8,             (%%rdi,%%rdx,1)\n"
"vpaddd      %%ymm11,            %%ymm10,     %%ymm11\n"
"vmovdqu      %%ymm11,            (%%rdi,%%rbp,1)\n"

"add          $0x80,              %%rcx\n"
"add          $0x80,              %%rax\n"
"add          $0x80,              %%rdx\n"
"add          $0x80,              %%rbp\n"

"cmp          %%rcx,              %[limit]\n"
"jne          1b\n"
:[dest]"=m"(dest)
:[src0]"m"(src0), [src1]"m"(src1), [const0]"m"(const0), [const1]"m"(const1), [const2]"m"(const2), [limit]"r"(limit)
:"%rax","%rbx","%rcx","%rdx","%rsi","%rdi","%rbp","memory","cc"
```

最后，由于这里 dest 是内存中被写入的部分，剩下的部分是读取，因此这里需要在输出列表中加上 dest，=m 表示写入到内存。在输入列表中，需要加上 src0, src1, const0, const1, const2, limit，其中 m 表示从内存中读取，r 表示将变量加载到寄存器中。最后还需要加上破坏列表，用于记录哪些寄存器修改过。

最后运行程序。

这里分别使用 native，expert，sse，avx 以及手动修改的 avx 指令集执行相同的代码，其中 native 使用的是 00 优化，expert 使用的是 02 优化，其他的均为 03 优化。

可以看出，sse 和 avx 的性能比原来的指令集性能更好，其中在同样的优化强度下，avx 比 sse 性能更好，且手动修改后的 avx 与原来的 avx 性能相近。

```
PS E:\Desktop\vector> ./vector
native - 2751463424 - 15068971791 cycles
expert - 2751463424 - 4775683042 cycles
sse - 2751463424 - 3590233178 cycles
avx-auto - 2751463424 - 2529880681 cycles
avx-manual - 2751463424 - 2544619490 cycles
```

问题及收获：

这里通过 c 语言实现了内联汇编，这里使用了 avx 指令集，实际运行时，avx-auto 性能是最好的，其中 avx 性能比 sse 指令集更好。

内联汇编还需要包含破坏列表，破坏列表可以表示出哪些寄存器修改过，如果不写，编译器会认为这些寄存器没有被修改过，会在其他代码使用到对应的寄存器时，会因为没有进行更新导致寄存器的值出现错误。如果破坏列表中没有声明内存，但修改过内存，这会造成内存中的值不会被更新，汇编代码中会加载内存中原来的值。最后还需要声明条件码，否则会认为条件码没有被修改，造成错误。