Лабораторная работа 9: SIMD: перспективы производительности

1. Введение

Дальнейшее повышение производительности вычислений возможно только при помощи средств распараллеливания операций. Одним из способов увеличения степени параллелизма является векторное процессирование. Процессоры, поддерживающие технологию SIMD, дают возможность переработки существующих алгоритмов и программ с целью их многократного ускорения за счёт векторизации вычислений.

2. Постановка задачи

2.1. Цель выполнения лабораторной работы

Лабораторная работа направлена на достижение следующих навыков:

- практика определения критичных по производительности участков кода;
- практика применения технологии векторной обработки SSE;
- практика использования языка ассемблера;
- практика употребления мобильного синтаксиса AT&T;
- практика измерения производительности по схеме K-best.

2.2. Описание задачи

У органов специального назначения имеются образцы речи некоторых граждан. Необходимо автоматически распознать принадлежность произнесённого слова тому или иному гражданину.

Для этого образцы речи переводятся из пространственной области в область параметров фильтра предсказания некоторого вокодера (voice coder). Каждый отсчёт в области параметров – вектор из 16 однобайтовых элементов.

Идентификация осуществляется путём свёртки слова на каждом фрагменте каждого образца и сравнения найденного минимума в ортогональной метрике с пороговым значением. При большой базе данных скорость работы идентификатора является превалирующей.

2.3. Исходное решение задачи

2.3.1. Спецификация функции Stage

```
int Stage (void * original, void * signature, int signature_leng);
// свёртка сигнатуры на фрагменте поисковых признаков
// original – указатель на фрагмент поисковых признаков (кратно 16)
// signature – указатель на сигнатуру (кратно 16)
// signature_leng – число 128-битных векторов в сигнатуре
// возвращаемое значение – величина расстояния
```

2.3.2. Реализация функции Stage на Си

```
int Stage (void * original, void * signature, int signature_leng)
{
   int i,j;
   int S=0;
   for (j=0; j<signature_leng; j++)
      for (i=0; i<16; i++)
         S += abs( (int)(*((unsigned char *) original  + 16 * j + i )) -
                   (int)(*((unsigned char *) signature + 16 * j + i ))   );
   return S;
}
```

2.3.3. Спецификация функции SignFind

```
int SignFind (void * original, int original_leng,
              void * signature, int signature_leng);
// поиск в признаках
// original              – указатель на поисковые признаки (кратно 16)
// original_leng         – число 128-битных векторов
// signature             – указатель на сигнатуру (кратно 16)
// signature_leng        – число 128-битных векторов
// возвращаемое значение – минимальная величина расстояния
//                         (или -1, если правдоподобие не найдено)
```

2.3.4. Реализация функции SignFind на Си

```
int SignFind (void * original, int original_leng,
              void * signature, int signature_leng)
{
   char * U;
   char * V = (char *)original + ((original_leng-signature_leng)<<4);
   int Smin = THRESHOLD;
   for (U=original; U<=V; U+=16)
   {
      int S = Stage(U, signature, signature_leng);
      if (S<Smin) Smin=S;
   }
   return Smin < THRESHOLD ? Smin : -1 ;
}
```

2.4. Задача ускорения существующего решения

Производительность приведённой функции SignFind неприемлема.

Необходимо:

- Определить критичный по производительности участок кода
- Определить возможность применения векторной обработки
- Разработать SIMD алгоритм
- Реализовать SIMD алгоритм для Pentium 4

- Сравнить производительность по схеме K-best

3. Решение задачи

3.1. Выявление возможности векторизации

Критичный по производительности участок кода – базовый блок инструкций (ББИ) – наиболее глубоко вложенный код, вызываемый наиболее часто – тело функции Stage.

Данный ББИ хорошо сопрягается с технологией SSE.

3.2. Векторная реализация функции Stage

Реализация функции Stage на ассемблере Intel Pentium 4 с использованием техники SSE в синтаксисе AT&T.

```
.text
.global Stage
Stage:
       enter  $0, $0
       push   %esi
       push   %edi

       mov    8 (%ebp), %edi      ; edi – указатель на фрагмент поисковых признаков
       mov    12 (%ebp), %esi     ; esi – указатель на сигнатуру
       mov    16 (%ebp), %ecx     ; ecx – число 128-битных векторов в сигнатуре

       PXOR   %xmm1, %xmm1        ; xmm1 – аккумулятор суммы разностей

1:     MOVDQA (%edi), %xmm0       ; загрузка вектора признаков
       add    $16, %edi
       PSADBW (%esi), %xmm0       ; загрузка вектора сигнатуры
                                  ; и вычисление суммы модулей разностей
       add    $16, %esi
       PADDD  %xmm0, %xmm1        ; пополнение аккумулятора
       loop   1b

       PREFETCHT2 (%edi)          ; предвыборка для ускорения
                                  ; последующих вызовов Stage

       PSHUFD $0xAA, %xmm1, %xmm0
       MOVD   %xmm1, %ecx         ; выгрузка первой полусуммы
       MOVD   %xmm0, %eax         ; выгрузка второй полусуммы
       add    %ecx, %eax          ; формирование результата

       pop    %edi
       pop    %esi
       leave
       ret
```

4. Заключение

Сопоставив результаты измерений производительности при помощи схемы K-best, можно увидеть, что использование векторного процессирования позволяет получить ускорение вычислений в 51 раз!

5. Литература

1. John L. Hennessy and David A. Patterson, Computer Architecture: A Quantitative Approach, 3rd Ed., Morgan Kaufmann, 2003
2. Randal E. Bryant and David R. O'Hallaron, Computer Systems: A Programmer's Perspective, Prentice Hall, 2003
3. Официальная документация Intel Pentium 4, www.intel.com