Описание задачи

Пусть у нас имеется некоторый вектор $x \in R^n$ и пусть x_i - $i-ая$ компонента вектора x. Необходимо вычислить

$$a = \prod_{i=1}^{n} x_i$$

Предварительные изменения исходного кода

- Увеличено число элементов вектора до 1 000 000 000
- Изменён способ генерации чисел, чтобы при их перемножении мы не выходили за пределы double или не получали в результате ноль

  v.data[i] = 1.0 + ((double)rand() - 0.5) / RAND_MAX / 1e9;
- Добавлены функции, измеряющие время и печатающие результат работы программы
- Сменены 32 битные целые числа на 64 битные

Результат работы исходного кода

CMakeLists.txt

```
project(optimization)
cmake_minimum_required(VERSION 3.9)

find_package(OpenMP)
if (OPENMP_FOUND)
    set (CMAKE_C_FLAGS "${CMAKE_C_FLAGS} ${OpenMP_C_FLAGS}")
    set (CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} ${OpenMP_CXX_FLAGS}")
    set (CMAKE_EXE_LINKER_FLAGS "${CMAKE_EXE_LINKER_FLAGS} ${OpenMP_EXE_LINKER_FLAGS}")
endif()

add_executable(test_multiply main.c)
```

```
moskalenkoviktor@moskalenkoviktor-Aspire-E5-571G:~/Документы/Study/6_1/optimization/build$ cmake ..
-- The C compiler identification is GNU 7.2.0
-- The CXX compiler identification is GNU 7.2.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Found OpenMP_C: -fopenmp (found version "4.5")
-- Found OpenMP_CXX: -fopenmp (found version "4.5")
-- Configuring done
-- Generating done
-- Build files have been written to: /home/moskalenkoviktor/Документы/Study/6_1/optimization/build
```

Result — результат перемножения элементов вектора
Time — время выполнения умножения

```
Result - 1.648726
Time - 19.559099 sec
```

Оптимизация 1 (смена компилятора)

Меняем компилятор с **GNU** на **Intel**

Результаты

```
moskalenkoviktor@moskalenkoviktor-Aspire-E5-571G:~/Документы/Study/6_1/optimization/build$ cmake -DCMAKE_BUILD_TYPE=RelWithDebInfo -DCMAKE_C_C
OMPILER=icc -DCMAKE_CXX_COMPILER=icpc ..
-- The C compiler identification is Intel 19.0.0.20181018
-- The CXX compiler identification is Intel 19.0.0.20181018
-- Check for working C compiler: /home/moskalenkoviktor/intel/bin/icc
-- Check for working C compiler: /home/moskalenkoviktor/intel/bin/icc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /home/moskalenkoviktor/intel/bin/icpc
-- Check for working CXX compiler: /home/moskalenkoviktor/intel/bin/icpc -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Found OpenMP_C: -qopenmp
-- Found OpenMP_CXX: -qopenmp
-- Configuring done
-- Generating done
-- Build files have been written to: /home/moskalenkoviktor/Документы/Study/6_1/optimization/build
```

```
Result - 1.648726
Time - 9.777383 sec
```

Попытка оптимизации 1 (проверка векторизации)

Проверяем векторизацию основного цикла программы

Поиск проблемы

Заходим в микроархитектурный анализ **Intel VTune Amplifier** и смотрим на исходный код функции **multiply**.

17	void multiply(my_vector *v, double *res)				
18	{				
19	int64_t i;				
20	double tmp_res = 1;				
21	//#pragma omp parallel for reduction(* : tmp_res)				
22	for (i = 0; i < my_vector_len(v); ++i) {	6,579,000,000	4,981,000,000	1.321	1.2%
23	tmp_res *= v->data[i];	19,697,900,000	3,784,200,000	5.205	6.1%
24	}				
25	*res = tmp_res;				
26	}				

Результаты

0x400e3b		**Block 10:**	
0x400e3b	23	mulsdq (%rsi,%r8,8), %xmm1	
0x400e41	22	inc %r8	
0x400e44	22	cmp %r10, %r8	
0x400e47	22	jb 0x400e3b <Block 10>	
0x400e49		**Block 11:**	
0x400e49	20	unpcklpd %xmm0, %xmm1	
0x400e4d		**Block 12:**	
0x400e4d	23	mulpdx (%rsi,%r10,8), %xmm1	0
0x400e53	23	mulpdx 0x10(%rsi,%r10,8), %xmm1	7,146,800,000
0x400e5a	23	mulpdx 0x20(%rsi,%r10,8), %xmm1	6,291,700,000
0x400e61	23	mulpdx 0x30(%rsi,%r10,8), %xmm1	6,259,400,000
0x400e68	22	add $0x8, %r10	6,473,600,000
0x400e6c	22	cmp %r9, %r10	105,400,000
0x400e6f	22	jb 0x400e4d <Block 12>	
0x400e71		**Block 13:**	
0x400e71	20	movaps %xmm1, %xmm3	
0x400e74	20	unpckhpd %xmm1, %xmm3	
0x400e78	20	mulsd %xmm3, %xmm1	
0x400e7c		**Block 14:**	
0x400e7c	22	cmp %rdi, %r9	
0x400e7f	22	jnb 0x400e8f	
0x400e81		**Block 15:**	

Смотрим во что скомпилировалась наша функцию, видим что уже присутствуют векторные инструкции **mulpdx**. Компилятор сам справился с векторизацией.

Попытка оптимизации 2 (проверка в VTune)

Поиск проблемы

Открываем **Intel VTune Amplifier**, выбираем наш исполняемый файл. Запускаем микроархитектурный анализ, смотрим на функцию **multiply**:

Retiring:	**7.4%**	**of Pipeline Slots**
Front-End Bound:	**0.8%**	**of Pipeline Slots**
Bad Speculation:	**0.5%**	**of Pipeline Slots**
Back-End Bound:	**91.4%**	**of Pipeline Slots**
Memory Bound:	**65.8%**	**of Pipeline Slots**
L1 Bound:	**11.9%**	**of Clockticks**
DTLB Overhead:	3.2%	of Clockticks
Loads Blocked by Store Forwarding:	0.0%	of Clockticks
Lock Latency:	0.0%	of Clockticks
Split Loads:	0.0%	of Clockticks
4K Aliasing:	0.0%	of Clockticks
FB Full:	5.4%	of Clockticks
L2 Bound:	10.5%	of Clockticks
L3 Bound:	**7.8%**	**of Clockticks**
Contested Accesses:	0.0%	of Clockticks
Data Sharing:	0.0%	of Clockticks
L3 Latency:	100.0%	of Clockticks
SQ Full:	3.2%	of Clockticks
DRAM Bound:	**35.0%**	**of Clockticks**
Memory Bandwidth:	32.1%	of Clockticks
Memory Latency:	**53.3%**	**of Clockticks**
LLC Miss:	74.5%	of Clockticks
Store Bound:	**0.0%**	**of Clockticks**
Core Bound:	**25.7%**	**of Pipeline Slots**

Спускаемся через самые большие цифры к **Memory Latency**. Проверим пропускную способность DRAM в бенчмарке:

```
moskalenkoviktor@moskalenkoviktor-Aspire-E5-571G:~$ mbw 4096 -t1
Long uses 8 bytes. Allocating 2*536870912 elements = 8589934592 bytes of memory.
Getting down to business... Doing 10 runs per test.
0       Method: DUMB    Elapsed: 0.52710        MiB: 4096.00000 Copy: 7770.777 MiB/s
1       Method: DUMB    Elapsed: 0.52906        MiB: 4096.00000 Copy: 7741.975 MiB/s
2       Method: DUMB    Elapsed: 0.52811        MiB: 4096.00000 Copy: 7755.901 MiB/s
3       Method: DUMB    Elapsed: 0.52565        MiB: 4096.00000 Copy: 7792.198 MiB/s
4       Method: DUMB    Elapsed: 0.55435        MiB: 4096.00000 Copy: 7388.874 MiB/s
5       Method: DUMB    Elapsed: 0.52971        MiB: 4096.00000 Copy: 7732.547 MiB/s
6       Method: DUMB    Elapsed: 0.52949        MiB: 4096.00000 Copy: 7735.687 MiB/s
7       Method: DUMB    Elapsed: 0.52661        MiB: 4096.00000 Copy: 7778.052 MiB/s
8       Method: DUMB    Elapsed: 0.52905        MiB: 4096.00000 Copy: 7742.223 MiB/s
9       Method: DUMB    Elapsed: 0.52677        MiB: 4096.00000 Copy: 7775.631 MiB/s
AVG     Method: DUMB    Elapsed: 0.53059        MiB: 4096.00000 Copy: 7719.685 MiB/s
```

Скорость передачи данных в нашей задаче:

$$\frac{8 * 1000000000\,байт}{0.975949\,сек} \approx 7817.411\,\frac{Мбайт}{сек}$$

Результаты

Мы упёрлись в пропускную способность оперативной памяти

Оптимизация 2 (параллелизм)

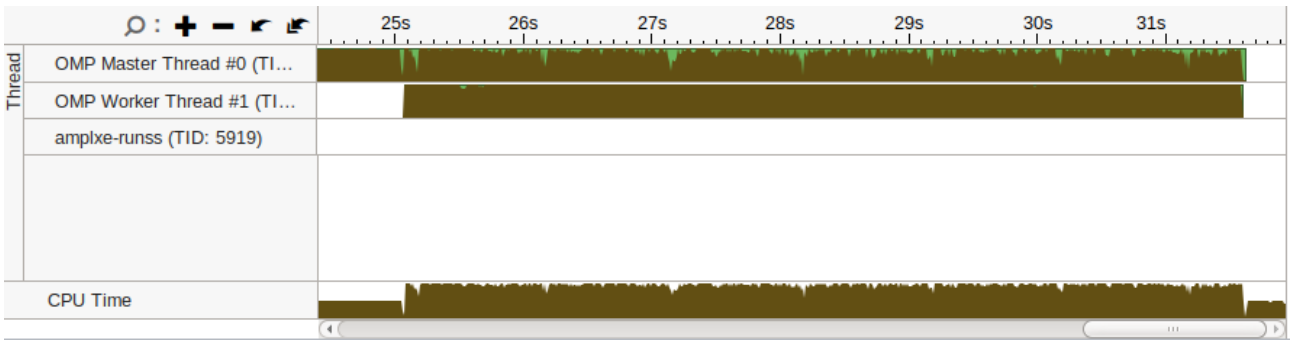
Добавим прагму перед циклом и запустим программу в два потока

```c
void multiply(my_vector *v, double *res)
{
    long i;
    double tmp_res = 1;
    #pragma omp parallel for reduction(* : tmp_res)
    for (i = 0; i < my_vector_len(v); i++) {
        tmp_res *= v->data[i];
    }
    *res = tmp_res;
}
```

Результаты

```
Result - 1.648720
Time - 5.954473 sec
```

Получили ускорение в 1.64 раз


25s
26s
27s
28s
29s
30s
31s
Thread
OMP Master Thread #0 (TI…
OMP Worker Thread #1 (TI…
amplxe-runss (TID: 5919)
CPU Time