

Київський національний університет імені Тараса Шевченка
Факультет радіофізики, електроніки та комп'ютерних систем

Лабораторна робота № 3
з курсу «Комп'ютерні системи»

Роботу виконав
студент 3 курсу
Комп'ютерної інженерії(СА)
Мураховський Владислав

Хід:
Туторіал:

```
GNU: :s2 [tb225 code]$ ml icc
```

Виконуємо програму

```
GNU: :s2 [tb225 src]$ icc -O1 -std=c99 Multiply.c Driver.c -o MatVector
GNU: :s2 [tb225 src]$
```

Після виконання бачим результат(Час виконання).

```
Execution time is 11.952 seconds
GigaFlops = 1.706976
Sum of result = 195853.999899
```

Тепер зробимо з оптимізацією -O2 та порівняємо часи.

```
GNU: :s2 [tb225 src]$ icc -std=c99 -O2 -D NOFUNCCALL -qopt-report=1 -qopt-report-phase=vec Multiply.
c Driver.c -o MatVector
icc: remark #10397: optimization reports are generated in *.optrpt files in the output location
GNU: :s2 [tb225 src]$
icc: remark #10397: optimization reports are
GNU: :s2 [tb225 src]$ icc -std=c99 -O2 -D NO
GNU: :s2 [tb225 src]$ ./MatVector

ROW:101 COL: 101
Execution time is 4.121 seconds
GigaFlops = 4.951174
```

З результату можна зробити висновок,що час зменшився майже 3рази.

Дивимось на файли .optrpt. Бачим що 3 вложений цикл був векторизован.

Перекомпілюємо

```
GNU: :s2 [tb225 src]$ icc -std=c99 -O2 -D NOFUNCCALL -qopt-report-phase=vec,loop -qopt-report=2 Mult
GNU: :s2 [tb225 src]$ ./MatVector

ROW:101 COL: 101
Execution time is 4.126 seconds
GigaFlops = 4.944545
Sum of result = 195853.999899
```

Бачим детальніший звіт.І бачим причини чого не були векторизовані цикли.

```

:NU: :s2 [tb225 src]$ cat Multiply.optrpt
:ntel(R) Advisor can now assist with vectorization and show optimization
report messages with your source code.
:ee "https://software.intel.com/en-us/intel-advisor-xe" for details.

begin optimization report for: matvec(int, int, double (*)[*], double *, double *)
  Report from: Loop nest & Vector optimizations [loop, vec]

.OOP BEGIN at Multiply.c(37,5)
  remark #15541: outer loop was not auto-vectorized: consider using SIMD directive

  LOOP BEGIN at Multiply.c(49,9)
    remark #15344: loop was not vectorized: vector dependence prevents vectorization. First dependence is shown below. Use level 5 report for details
    remark #15346: vector dependence: assumed FLOW dependence between b[i] (50:13) and b[i] (50:13)
    remark #25439: unrolled with remainder by 2
  LOOP END

  LOOP BEGIN at Multiply.c(49,9)
    <Remainder>
  LOOP END
.OOP END

```

Бачим детальніший звіт. І бачим причини чого не були векторизовані цикли.

Тепер скопілюємо програму без макроса NOFUNCCALL, через нього не викликався цикл.

```

    for (k = 0; k < REPEATTIMES; k++) {
#ifdef NOFUNCCALL
        int i, j;
        for (i = 0; i < size1; i++) {
            b[i] = 0;
            for (j = 0; j < size2; j++) {
                b[i] += a[i][j] * x[j];
            }
        }
#else
        matvec(size1, size2, a, b, x);
#endif
    }
}

```

але з макросом NOALIAS, що забезпечить використання restrict для 2-го аргумента matvec.

```

=====
KNU: :s2 [tb225 src]$ icc -std=c99 -qopt-report=2 -qopt-report-phase=vec -D NOALIAS Multiply.c Driver.c -o MatVector
icc: remark #10397: optimization reports are generated in *.optrpt files in the output location
KNU: :s2 [tb225 src]$ ./MatVector

```

```

  LOOP BEGIN at Multiply.c(49,9)
    remark #15300: LOOP WAS VECTORIZED
  LOOP END

```

Як бачимо тепер цикл був векторизован завдяки тому що ми прибрали одну з залежностей аліаса. (Два вказівника вказували на одну область). Компілятор вважає, що елементи, що отримуються за рахунок вказівника, що дозволяє бути впевненими що кожна ітерація вважається незалежною.

Достігнем оптимізації за рахунок вирівнювання даних.

```

Begin optimization report for: matvec(int, int, double (*)[*], double *__restrict__, double *)
Report from: Vector optimizations [vec]

LOOP BEGIN at Multiply.c(37,5)
remark #15542: loop was not vectorized: inner loop was already vectorized

LOOP BEGIN at Multiply.c(49,9)
remark #15388: vectorization support: reference a[i][j] has aligned access [ Multiply.c(50,21) ]
remark #15388: vectorization support: reference x[j] has aligned access [ Multiply.c(50,31) ]
remark #15305: vectorization support: vector length 2
remark #15399: vectorization support: unroll factor set to 4
remark #15309: vectorization support: normalized vectorization overhead 0.594
remark #15355: vectorization support: *(b+i*8) is double type reduction [ Multiply.c(50,13) ]
remark #15300: LOOP WAS VECTORIZED
remark #15448: unmasked aligned unit stride loads: 2
remark #15475: --- begin vector cost summary ---
remark #15476: scalar cost: 10
remark #15477: vector cost: 4.000
remark #15478: estimated potential speedup: 2.410
remark #15488: --- end vector cost summary ---
LOOP END

LOOP BEGIN at Multiply.c(49,9)
<Remainder loop for vectorization>
remark #15388: vectorization support: reference a[i][j] has aligned access [ Multiply.c(50,21) ]
remark #15388: vectorization support: reference x[j] has aligned access [ Multiply.c(50,31) ]
remark #15335: remainder loop was not vectorized: vectorization possible but seems inefficient. Use vector always directive or -vec-threshold0 to override
remark #15305: vectorization support: vector length 2
remark #15309: vectorization support: normalized vectorization overhead 2.417
remark #15355: vectorization support: *(b+i*8) is double type reduction [ Multiply.c(50,13) ]
remark #15448: unmasked aligned unit stride loads: 2
remark #15475: --- begin vector cost summary ---
remark #15476: scalar cost: 10
remark #15477: vector cost: 4.000
remark #15478: estimated potential speedup: 2.410
remark #15488: --- end vector cost summary ---
LOOP END
LOOP END

```

icc: remark #10397: optimization reports are generated in *.oprpt files in the output location

```

GNU: :s2 [tb225 src]$ ./MatVector

```

```

ROW:101 COL: 102
Execution time is 4.269 seconds
GigaFlops = 4.779146
Sum of result = 195853.999899

```

Як бачимо при порівнянні з минулими результатами, швидкодії збільшилась, але не набагато.

Тепер зробимо міжпроцедурні оптимізації:

```

=====
GNU: :s2 [tb225 src]$ icc -std=c99 -qopt-report=2 -qopt-report-phase=vec -D NOALIAS -D ALIGNED -ipo Multiply.c Driver.c -o MatVector
icc: remark #10397: optimization reports are generated in *.oprpt files in the output location
GNU: :s2 [tb225 src]$ ./MatVector

```

ROW:101 COL: 102

```

ROW:101 COL: 102
Execution time is 4.043 seconds
GigaFlops = 5.046549
Sum of result = 195853.999899

```

Як бачимо швидкодія теж зростає.

1. Напишіть сценарій, що:
 - a. Компілює програму з різними оптимізаціями (-O) та виміряйте час її роботи. Якщо час досить малий - вимірюйте час роботи 1000 (чи 1000000) запусків алгоритму в циклі. Час роботи можна виміряти утилітою time.
 - b. Отримує перелік всіх розширень процесору що підтримуються
 - c. Для кожного розширення компілює Intel-компілятором окремий варіант оптимізованого коду (наприклад -x SSE2)
 - d. Вимірює час виконання кожного варіанта оптимізованої програми

Для прикладу я написав програму C++. Її суть є два масиви, 1 масив заповнюється рандомно, 2 за формулою. Результат множення матриць (ціх двох масивів). Двохвимірні масиви на 500 елементів.

код: самої програми можна побачити на Git.

Для виміру часу оптимізації був написан такий скрипт:

```
#!/bin/bash
cd /home/grid/testbed/tb225/code
ml icc
g++ myCode.cpp -o example0
time ./example0
icc -O1 myCode.cpp -o example1
time ./example1
icc -O2 -qopt-report=2 -qopt-report-phase=vec myCode.cpp -o example2
time ./example2
icc -qopt-report=2 -qopt-report-phase=vec myCode.cpp -o example3
time ./example3
icc -qopt-report=4 -qopt-report-phase=vec myCode.cpp -o example4
time ./example4
icc -qopt-report=2 -qopt-report-phase=vec -ipo myCode.cpp -o example5
time ./example5
```

Результат виконання даного скрипта:

```
real    0m10.833s
user    0m10.815s
sys     0m0.017s

real    0m5.414s
user    0m5.390s
sys     0m0.022s
icc: remark #10397: optimization reports are generated in *.optrpt files in the output location

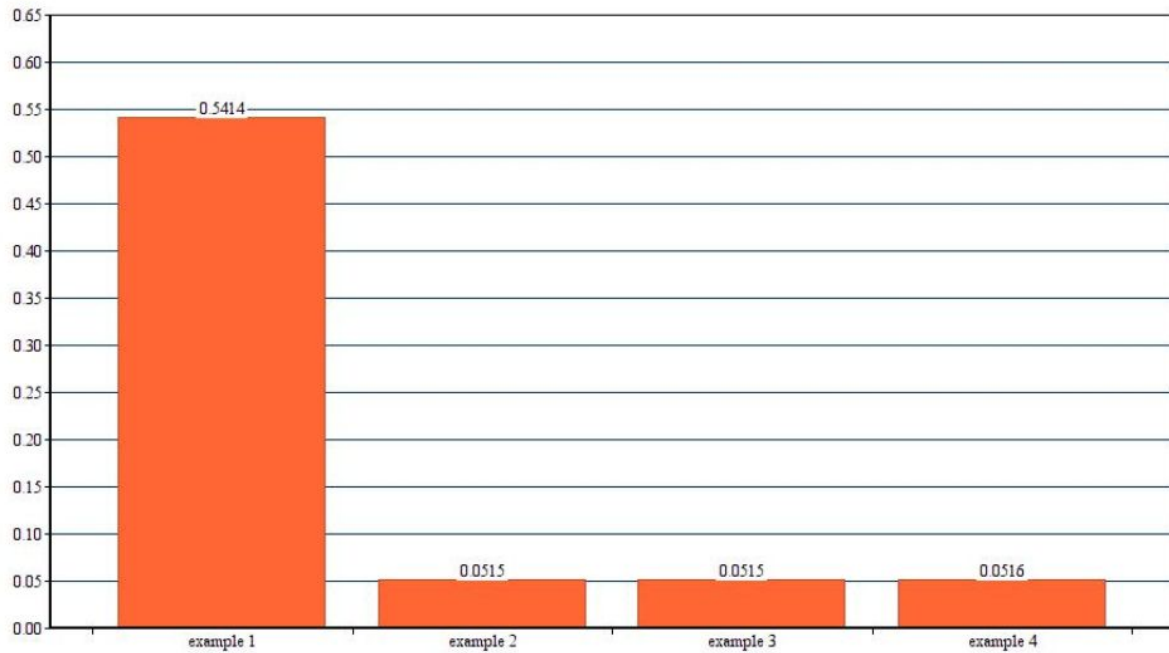
real    0m0.515s
user    0m0.502s
sys     0m0.012s
icc: remark #10397: optimization reports are generated in *.optrpt files in the output location

real    0m0.515s
user    0m0.500s
sys     0m0.014s
icc: remark #10397: optimization reports are generated in *.optrpt files in the output location

real    0m0.516s
user    0m0.503s
sys     0m0.012s
icc: remark #10397: optimization reports are generated in *.optrpt files in the output location
```

Як бачимо найкращий результат показав з опцією -O2. (Примітка при кожному запуску скрипта є маленька похибка, але вона ніяк не впливає на загальний результат.) Наступні 3 результати відрізняються лише трохи.

Діаграма:



Розширення. Подивимось як буде оптимізація з опцією -O2 на різних розширеннях.

Скрипт:

```
#!/bin/bash
exts=`cat /proc/cpuinfo | grep flags | cut -d: -f2 | uniq`
for i in $exts
do
    icc -O2 -qopt-report=2 -qopt-report-phase=vec myCode.cpp -o example6 -x$i 2>/dev/null
    if [ $? -eq 0 ]
    then
        printf "\nmode: $i"
        time ./example6
    fi
done
```

Результат:

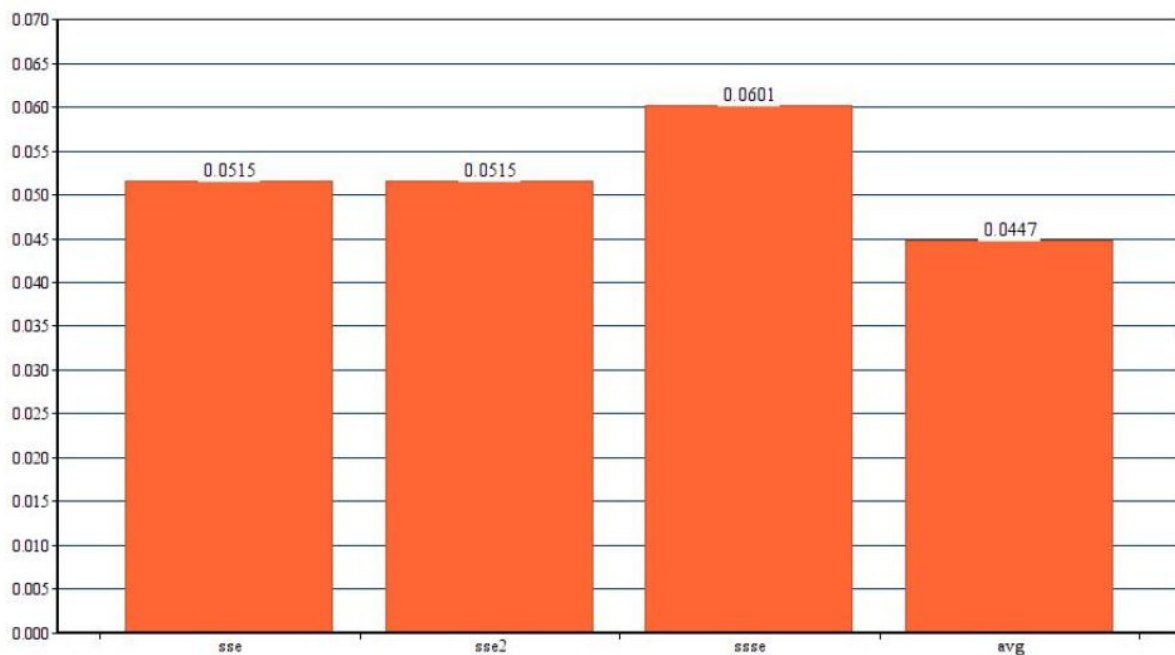

```
mode: sse
real    0m0.515s
user    0m0.503s
sys     0m0.011s
```

```
mode: sse2
real    0m0.515s
user    0m0.502s
sys     0m0.012s
```

```
mode: ssse3
real    0m0.601s
user    0m0.590s
sys     0m0.010s
```

```
mode: avx
real    0m0.447s
user    0m0.433s
sys     0m0.013s
```

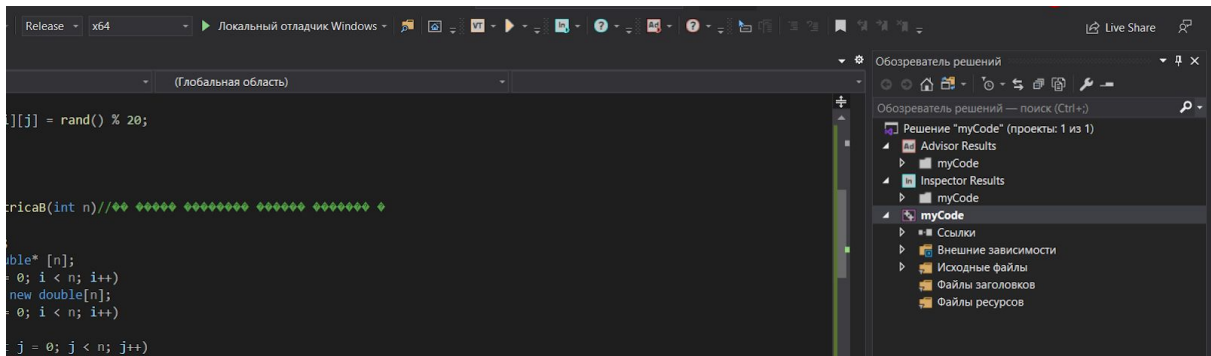
Діаграма



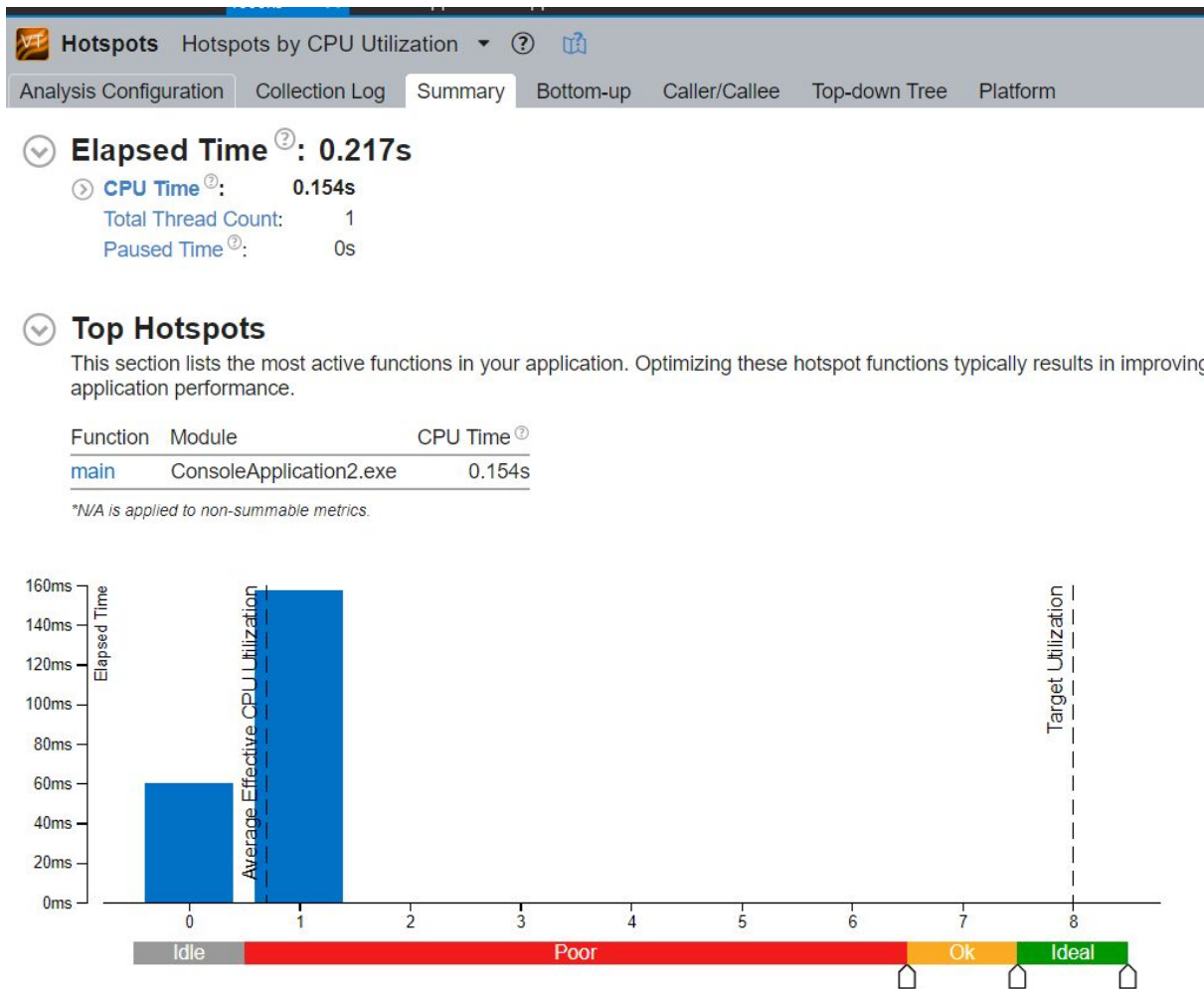
Як бачимо швидкодія трохи збільшилась,особливо avx.

Parallel studio:

Спочатку скомпілюємо програму звичайно:



Запустимо за допомогою утиліт аналітику по часу виконання програми:



Дивимось де найбільше по часу виконувалось

double** q = MultiplyMatrica(number, c, z);	100.0%	153.778ms
---------------------------------------------	--------	-----------

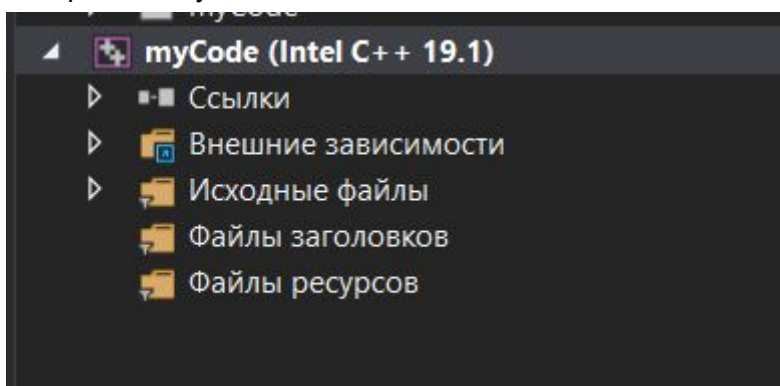
Як бачимо цей метод виконує множення матриць розміром 700 на 700.

Параметри для оптимізації

Оптимизация	Максимальная оптимизация (приоритет скорости) (/O2)
Развертывание подставляемых функций	По умолчанию
Включить подставляемые функции	Да (/Oi)
Предпочитать размер или скорость	Никакой
Опустить указатели на фреймы	Да (/Oy)
Включить волоконно-безопасные операции	Нет
Оптимизация всей программы	Да (/GL)

```
double** MultiplyMatrica(int n, double** c, double** z)
{
    double** mA;
    mA = new double* [n];
    for (int i = 0; i < n; i++)
        mA[i] = new double[n];
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++) {
            mA[i][j] = 0;
            for (int k = 0; k < n; k++)
                mA[i][j] += z[i][k] * c[k][j];
        }
    return mA;
}
```

Використовуємо Intel C++



Function	Module	CPU Time [?]
main	ConsoleApplication2.exe	0.127s

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

Ось і бачим приріст швидкодії програми

Function / Call Stack	CPU Time ▼ ⓘ	Module	Function (Full)	Source File	Start Address
main	127.171ms	ConsoleApplication2.exe	main	ConsoleApplication2.cpp	0x401000

Але для вдалого результату я збільшу розміри масивів до 1000 елементів.



Для intel c++:



Function	CPU Time: Total ▼ ⓘ	CPU Time: Self ⓘ	Module	Function (Full)	
BaseThreadInitThunk	100.0%	0s	kernel32.dll	BaseThreadInitThunk	
func@0x4b2e7b29	100.0%	0s	ntdll.dll	func@0x4b2e7b29	
_scrt_common_main_seh	100.0%	0s	ConsoleApplication2.exe	_scrt_common_main_seh	exe
func@0x4b2e7b45	100.0%	0s	ntdll.dll	func@0x4b2e7b45	
main	100.0%	1.783s	ConsoleApplication2.exe	main	Cor

Ось і бачимо різницю в часі під час різного запуску одної програми.

Висновок:

При виконанні лабораторної роботи мною були досліджені різні варіанти оптимізації. Компіляція відбувалась на університетському кластері з лише консольним інтерфейсом. Спочатку я пройшов по Tutorial, який прилагався к л.р. Потім я дослідив та побудував графіки з різними флагами компіляції программ. На останньому етапі: я оптимизував програму за допомогою програми Paralell Studio.

<https://github.com/amurahovski/LABCS3>