Question 1.

Please see Makefile

Question 2.

make clean
gcc dotproduct.c -pg -o dotproduct
./dotproduct
gprof ./dotproduct gmon.out

Result:
Flat profile:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
100.79	0.01	0.01	3	3.36	3.36	init_matrix
0.00	0.01	0.00	1	0.00	0.00	dot_product

init_matrix takes most of the time.

Question 3.

Memory-bound. From the the flat profile in the question 2, we can see that init_matrix use 100.79 % time. It's larger than the % time of dot_product. init_matrix is to about the memory allocation, so it's the memory-bound.

Here are more about computer-bound and memory-bound. Computer-bound means the rate at which process progresses is limited by the speed of the CPU. If the calculations take most of the time, it's likely to be computer-bound. Memory-bound means the rate at which a process progress is limited by the amount memory available and the speed of that memory access.

Reference:
http://stackoverflow.com/questions/868568/what-do-the-terms-cpu-bound-and-i-o-bound-mean

Question 4.

Step 1: make clean
Step 2: make
Step 3: time ./dotproduct

real 0m0.020s
user 0m0.015s
sys 0m0.004s

- Explain the different values of this command.
Real shows the total turn around time for a process. User shows the execution time for user-defined instructions. Sys is for time for executing system calls.

- Why user is so different from real?
Real is time from start to finish of the call. It is all elapsed time including time slices used by other processes and time the process spend blocked. For example, it includes the waiting time for I/O. However user only includes time spent in user-mode code within the process.

- How can you fix this problem?
We can put the execution job in a higher priority. The time slices and the time the process spend blocked decrease. Another option is to use a multicore system. On a multicore system, the sum of the user and sys time can actually exceed the real time.

Reference:
http://stackoverflow.com/questions/556405/what-do-real-user-and-sys-mean-in-the-output-of-time1

Question 5.

dotproduct_timer.c is the file that contains some timer to measure the execution time.

make dotproduct_timer

```
  clock_t start_time, end_time;
  float elapsed_time;

  start_time = clock();

  for (k=0;k<N;k++) {
        c[i][j] += a[i][k] * b[k][j] ;
```

```
  }

  end_time = clock();

  elapsed_time = ((float)(end_time - start_time)) / (CLOCKS_PER_SEC);

  printf("          N: %d\n", N);
  printf("elapsed time: %.9f second\n", elapsed_time);
```

Question 6 - 1.
We measure dot_product, rather than init_matrix.

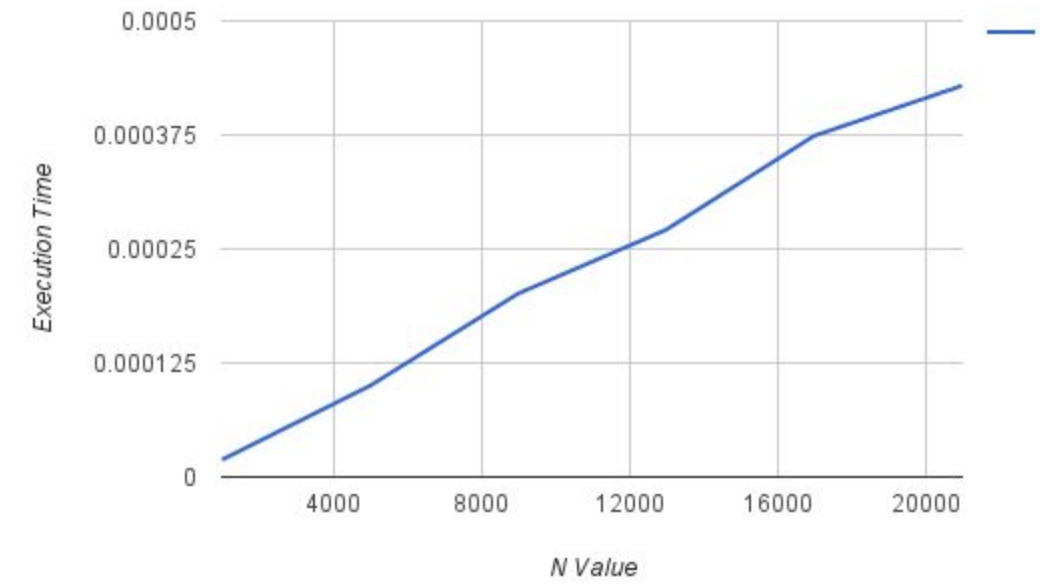
a). Native code
Normalized execution time = Execution time / (N*2)

We do two datasets. One is from 1000 - 21000. Another one is from 7500 - 8600.

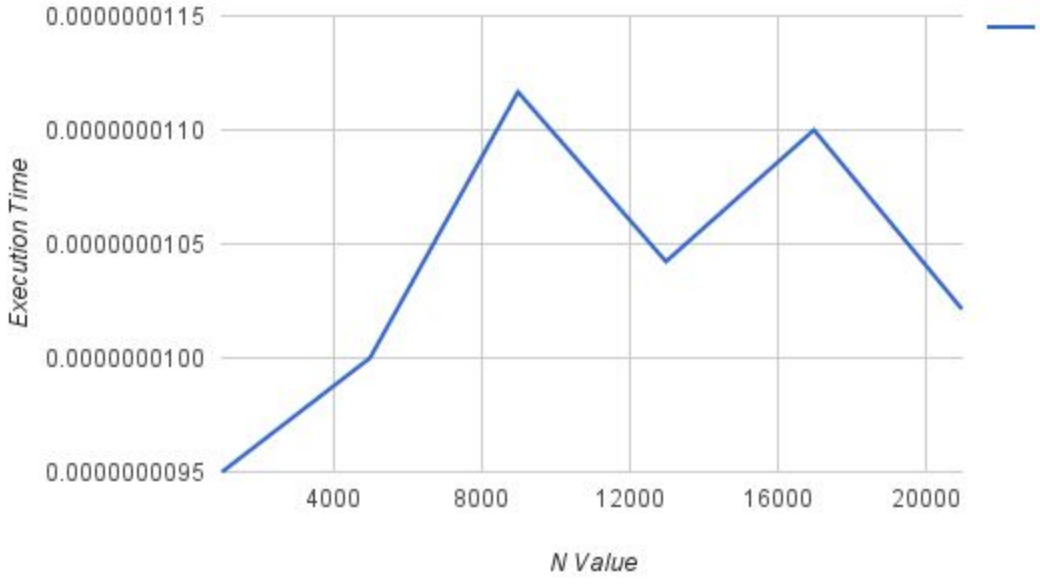
- Dataset 1
Execution Time v.s N Value and Normalized Execution Time v.s N Value Table

N	1000	5000	9000	13000	17000	21000
Elapsed time	0.000019	0.0001	0.000201	0.000271	0.000374	0.000429
Normalized time	0.0000000095	0.0000000100	0.0000000112	0.0000000104	0.0000000110	0.0000000102

Execution Time v.s N Value Plot



Normalized Execution Time v.s N Value Plot

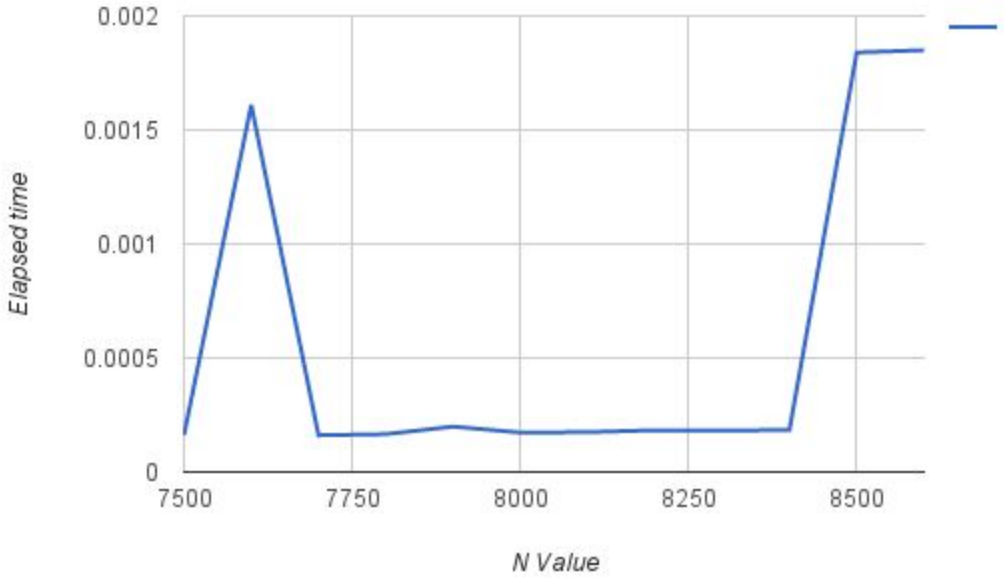


- Dataset 2

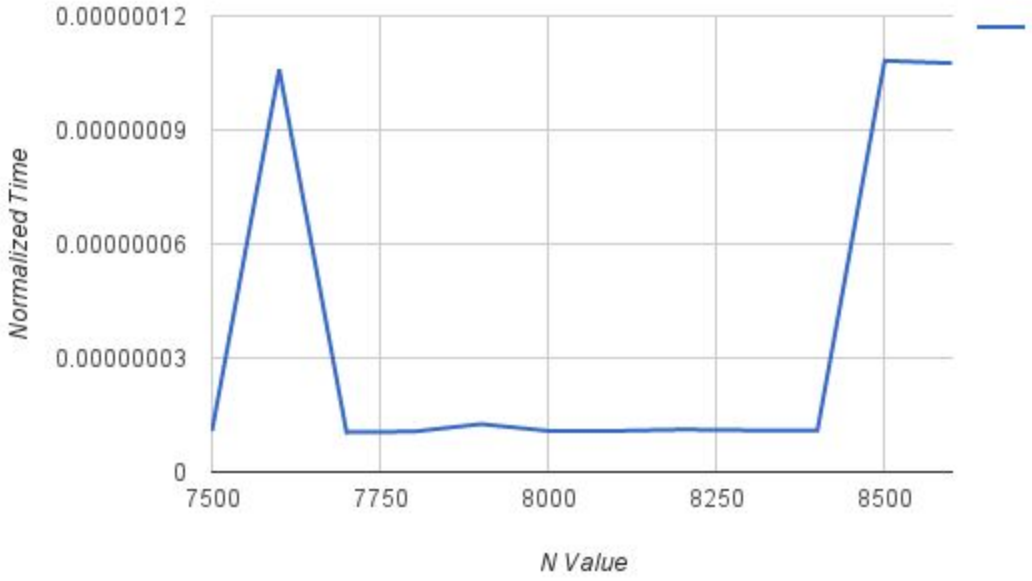
Execution Time v.s N Value and Normalized Execution Time v.s N Value Table

N	7500	7600	7700	7800	7900	8000	8100	8200	8300	8400	8500	8600
Elapsed time	0.000162	0.00161	0.000161	0.000166	0.000199	0.000173	0.000175	0.000183	0.000182	0.000184	0.0018 4	0.0018 5
Normalized time	0.000001080	0.000010592	0.000001045	0.000001064	0.000001259	0.000001081	0.000001080	0.000001116	0.000001096	0.000001095	0.000010824	0.000010756

Normalized Execution Time v.s N Value Plot



Normalized Execution Time v.s N Value Plot



b). Can you deduce anything?

In the dataset - 1, there is no plateau. In the dataset -2, we can see plateaus.

First, we check CPU details on the machine:

Architecture:	x86_64
CPU op-mode(s):	32-bit, 64-bit
Byte Order:	Little Endian
CPU(s):	8
On-line CPU(s) list:	0-7
Thread(s) per core:	2
Core(s) per socket:	4
Socket(s):	1
NUMA node(s):	1
Vendor ID:	GenuineIntel
CPU family:	6
Model:	60
Stepping:	3
CPU MHz:	3200.000
BogoMIPS:	6784.63

Virtualization: VT-x
L1d cache: 32K
L1i cache: 32K
L2 cache: 256K
L3 cache: 8192K
NUMA node0 CPU(s): 0-7

L1d cache and L1i cache are 32K, which is 32768 bytes. The int is 4 bytes. So each cache can hold 8192 int. It's possible that one vector is stored in two caches. If one vector is stored in two different caches, the normalized execution increases.

The reason is that it's cache-unfriendly. Elements that are adjacent in a row are supported to be adjacent in memory. So accessing them in sequence means accessing them in ascending memory order. This is called cache-friendly, since the cache tends to prefetch contiguous blocks of memory. However, if the elements that are adjacent in a row are not adjacent in memory, the cache fails to prefetch contiguous blocks of memory. Elements are distant in memory from each other. It jumps around in memory, potentially wasting the effort of the caches of retrieving the elements.

Reference: http://stackoverflow.com/questions/16699247/what-is-cache-friendly-code

c). Changing the alignment of the arrays.

I know we should alignment the arrays, but I don't know how to do it.

Question 6 - 2.
a). Write a loop around the location of your code to have more measurement sample
We can build a loop around the location of the code, as shown in the dotproduct_timer_better.c. In this case, we repeat the code for 1000 times, then we get the average of elapsed time.

In Makefile:
```
dotproduct_timer_better:
    gcc dotproduct_timer_better.c -o $@ -O0
```

```
  int repeated_time = 1000;
  int n;

  start_time = clock();
```

```
  for (n = 0; n < repeated_time; n++) {
        for (k=0;k<N;k++) {
        c[i][j] += a[i][k] * b[k][j] ;
        }
  }

  end_time = clock();

  elapsed_time = ((float)(end_time - start_time)) / (CLOCKS_PER_SEC * repeated_time);
```

b). Optimize your code.
Yes. I can optimize the code.

In Makefile:
```
dotproduct_timer:
   gcc dotproduct_timer.c -o $@ -O0
dotproduct_timer_optimized:
   gcc dotproduct_timer.c -o $@ -O3
```
dotproduct_timer_optimized is to use -O3

Here are optimize options which gcc provides. More details are in the reference.

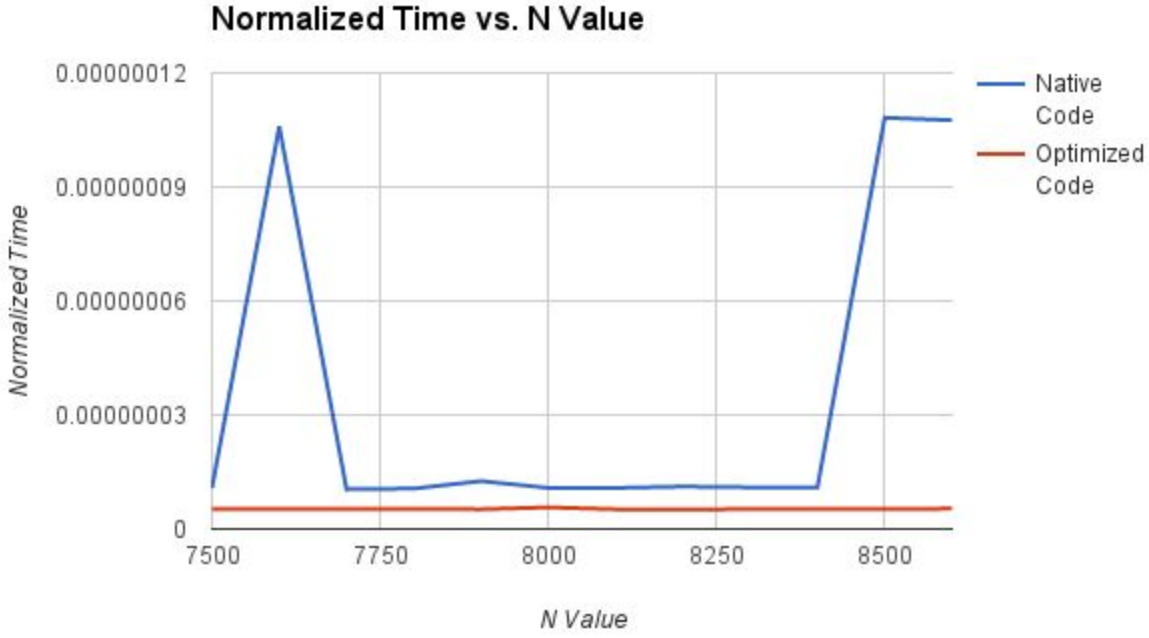
- -O0 (do no optimization, the default if no optimization level is specified)
- -O1 (optimize minimally)
- -O2 (optimize more)
- -O3 (optimize even more)
- -Ofast (optimize very aggressively to the point of breaking standard compliance)
- -Os: Optimize for code size. This can actually improve speed in some cases, due to better I-cache behavior.

The different between the 2 assembly codes is that the one with -O3 is optimized. -O3 includes a lot of methods, which can be found in the reference.

Reference:
http://stackoverflow.com/questions/1778538/how-many-gcc-optimization-levels-are-there
https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

c). The graphs with the optimized code vs native code.


Native Code vs. N Value
0.002
0.0015
0.001
0.0005
0
Elapsed time
7500
7750
8000
8250
8500
N Value
Native Code
Optimized Code


Normalized Time vs. N Value
0.00000012
0.00000009
0.00000006
0.00000003
0
Normalized Time
7500
7750
8000
8250
8500
N Value
Native Code
Optimized Code