CS-301
High Performance Computing
Assignment 1

Amarnath Karthi (201501005)
Chahak Mehta (201501422)

August 19, 2017

1. Probe the hardware information of your computing systems.

Vendor ID:	GenuineIntel
Model name:	Intel(R) Core(TM) i5-6300HQ CPU 2.30GHz
CPU MHz:	888.751
CPU max MHz:	3200.0000
CPU min MHz:	800.0000
L1d cache:	32K
L1i cache:	32K
L2 cache:	256K
L3 cache:	6144K
Architecture:	x86_64
CPU op-mode(s):	32-bit, 64-bit
Byte Order:	Little Endian
Socket(s):	1
Core(s) per socket:	4
Thread(s) per core:	1
CPU(s):	4
On-line CPU(s) list:	0-3

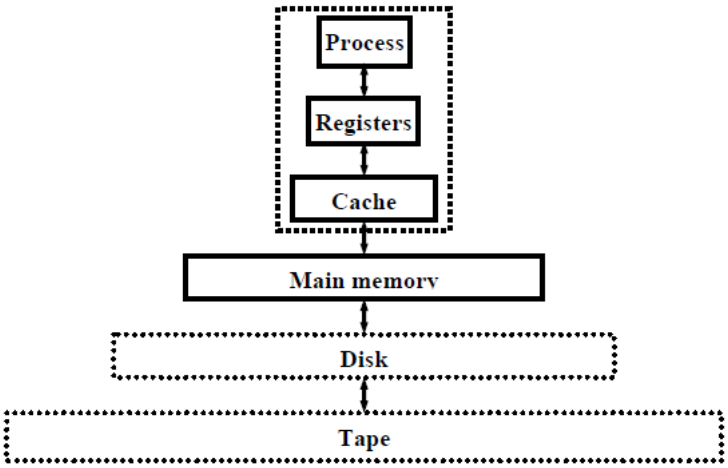

Figure 1: Memory hierarchy in a Computer

2. **Vector-triad benchmarking:**

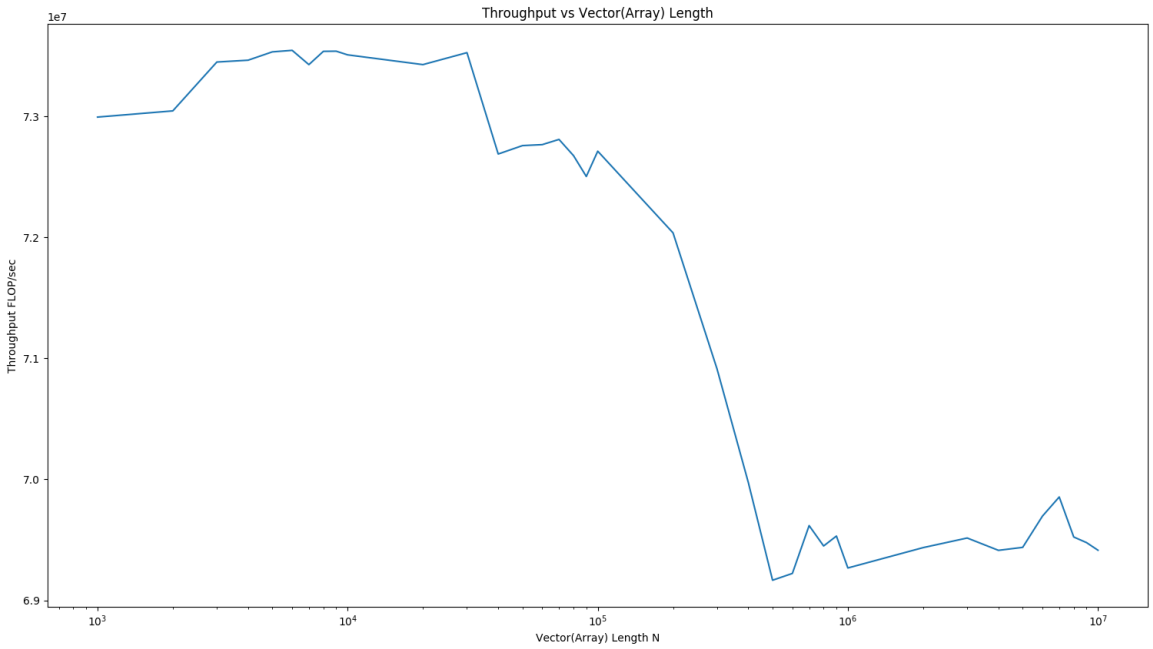

Figure 2: GFLOPS/sec vs Iterations

The Figure 2 was obtained by performing the Vector-Triad benchmarking 100 times and then taking the average of the runs. As can be seen in the Figure 2 we see a dip in the graph after 10^3 iterations. This is because the arrays used are stored in the L2 cache (256K) rather than L1 cache (32K) because of them being of a larger size then what the L1 cache can store. This increases the time taken to access the arrays for manipulation and hence leads to a lower throughput. Further, for more number of iterations, we can also see the dips in performance when the arrays will be shifted from L2 cache to L3 cache(6144K) and thereafter to the memory.

3. Take any of your code (with some reasonable no. of functions and subroutines) and do the profiling as discussed in the class (you can use gprof already installed in the machines). Understand the output.

Output:
Flat Profile:

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
100.89	218.49	1	218.49	218.49	func2	
0.86	220.37	1.88	1	1.88	1.88	func1

The flat profile gives the percentage of total running time each function takes, the number of times the functions are called and the time taken by each function to.

Call Graph:

index	%time	self	children	called	name ‘⟨spontaneous⟩’
[1]	100.0	0.00	220.37		main[1]
		218.49	0.00	1/1	func2 [2]
		1.88	0.00	1/1	func1 [3]
		218.49	0.00	1/1	main [1]
[2]	99.1	218.49	0.00	1	func2 [2]
		1.88	0.00	1/1	main [1]
[3]	0.9	1.88	0.00	1	func1 [3]

This table describes the call tree of the program, and was sorted by the total amount of time spent in each function and its children.

The line with the index number at the left hand margin lists the current function. The lines above it list the functions that called this function, and the lines below it list the functions this one called.

The output also contains information about the total amount of time spent in each function and its children, and the number of calls made to these functions. Also, if the parents of the function cannot be determined, the word ‘⟨*spontaneous*⟩’ is printed in the ‘name’ field, and all the other fields are blank.

4. . Implement some of the important optimization strategies discussed during the lecture today and measure difference in improvement while using the best strategy (look into the attached slide).

(a) **Case A:**

```
for(int i=0;i<N;i++)
  if(A[i] > 100){
    flag = true;
    break;
  }
```

Case B:

```
for(int i=0;i<N;i++)
  if(A[i] > 100){
    flag = true;
  }
```

Observation: When $N = 1 \times 10^9$ in the above cases, and we search in an already instantiated array, the flag is turned *true* when the value of $A[i]$ is greater than 100 which in this case was kept at $i = 500000000$. Theoretically, case B does all the iterations while case A has a break condition if the if condition returns *true*. Case A takes 0.989066 seconds while case B takes 1.956732 seconds.

Optimization Strategy: *Trying to perform the least number of operations if the output remains same.*

(b) **Squaring a number:**
A = A + B**2
A,B are float.
Case A:
A = A + B**2.0
Case B:
A = A + B*B

Observation: For large floating point numbers of the order 1×10^{25}, Case A takes 0.000019 seconds while case B takes 0.000000000001 seconds.

Optimization Strategy: *Try and avoid more resource consuming operations. In the first case, power does multiplication O(log(n)) times while in the second case, multiplication is done only one time.*

(c) **Case A:**

```
int s = 1, r=2;
for(int i=0;i<N;i++){
    A[i] = A[i] + s + r*sin(0.523598776);
  }
```

Operation $A(i) = A(i) + s + r * sin(x)$ is done 1×10^8 times.

Case B:

```
int s = 1, r=2;
double tmp = s + r*sin(0.523598776);
for(int i=0;i<N;i++){
    A[i] = A[i] + tmp;
  }
```

Operation $A(i) = A(i) + tmp$ is done 1×10^8 times.

Observation: As we can see in the code, the operation $s + r * sin(x)$ takes place 10^8 times in case A while it takes place only once in case B which makes it faster. Case A takes 3.584542 seconds, Case B takes 2.890220 seconds.

Optimization Strategy: *Reduce the number of registers used to store data, the number of comparisons made or the number of redundant operations*

(d) **Case A:**
Branched standard matrix multiplication (according to slide) where $N = 1 \times 10^4$.
Case B:
Unbranched standard matrix multiplication (according to slide) where $N = 1 \times 10^4$.

Observation: Case A takes 0.401576 seconds while case B takes 0.194602 seconds. The chances of a branch miss is higher in A then in B because of more number of branches in A than in B. This increases the latency and hence takes more time.

Optimization Strategy: *To avoid branching whenever possible.*