Matrix multiplication by using multi-threading

I. Methodology

The main point to improve the performance is not only increasing the number of threads but also making sure tasks are equality assigned into each thread. With mapping equally, it prevents main thread from waiting long time for one of thread still executing when others have completed their own tasks. Therefore, it leads in a lower CPU utilization and higher executing time.

The work allocates tasks to each thread by the total number of elements which depend on the size of matrix. In addition, to observe the multi-threading performance more flexible, dynamic memory allocation is adopted in my program to create the matrixes and threads by difference value of input (Fig 1).

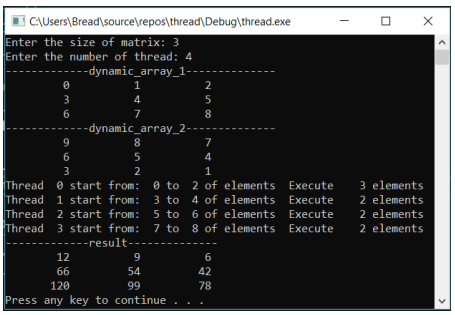


Fig. 1

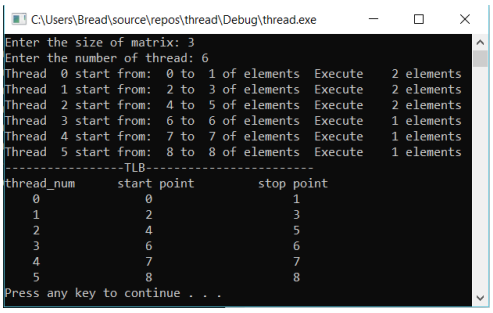


Fig. 2

Firstly, the program reads input parameters including matrix size and the number of threads to create global dynamic arrays and threads by using "`new`" instruction. Following that, the main program calculates the total number of elements of result array and assigns them to each thread. To minimize the executing time in each thread, main program calculates the start point and end point for every thread before thread start processing matrix multiplication, and put them into global array called TLB (Fig. 2). In this program, the work uses `CreateThread()` defined in `windows.h` to build threads. When creating threads, all of them are pointed to the same executing function but passed a unique number for each thread by using a parameter in `CreateThread()`. Therefore, in the thread function, by receiving the different thread number, each thread reads a different start point and stop point from TLB to get which part of element the thread is allocated to achieve computing separately and parallelly. In addition, the program can create multiple thread more flexible due to point threads to the same function. Furthermore, by using this method, the program will not happen critical section anymore due to every data in each thread is independent.

In this program, `Clock()` which is defined in `Time.h, and` adopted to calculate the time executing matrix multiplication. To estimate the running more accurately, the timer is only triggered during thread executing matrix multiplication. After checking threads and assigning tasks, main program starts the timer while all of threads are set to enable, until all the threads have been marked into finish state. As the result, the executing time can be more accurate and not affected by the other processes in main program.

II. Performance Analysis

To eliminate the tolerance of performance estimation, the work takes tests executing time for 100 times and take average value as the result.

According to the trend (Fig. 3), when executing nxn matrix multiplication by single thread, the using time remains stable with only approximately 1.5ms when matrix size is below 300. However, when the matrix size is over 300x300, the time increases significantly by growing around 1.7 time for per 100 n, due to the calculating becomes heavier to affect the performance. Therefore, it is inefficient for single thread to execute a heavy loading processing.

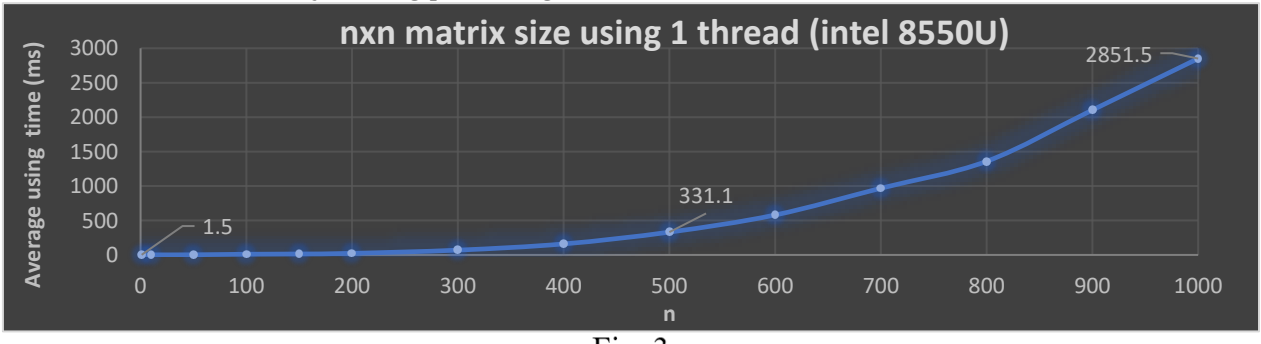


Fig. 3

● **Observe the O(n) performance by using threads**

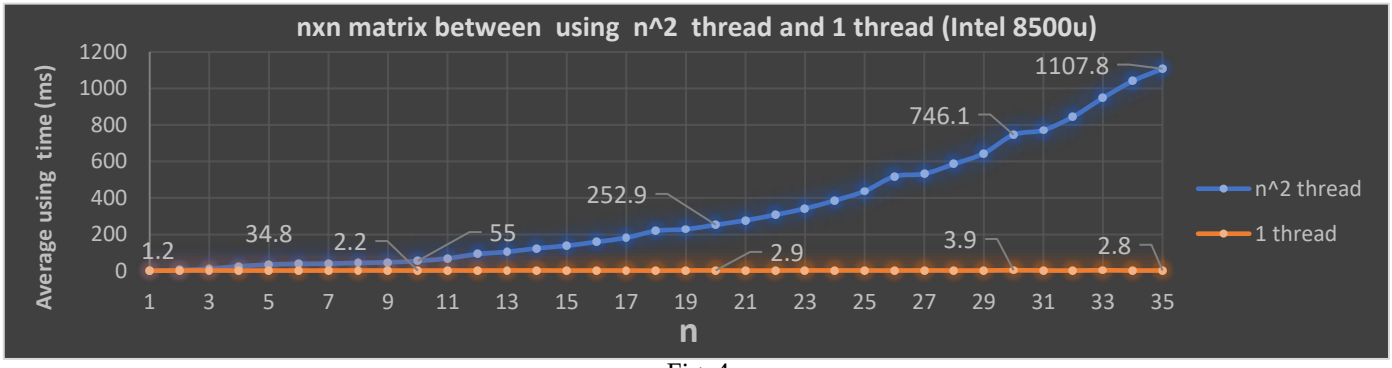


Fig. 4

The O(n) notation means the limiting behavior of the function which is dominated by n, and it implies that the equation is linear function. When executing 10x10 matrix, the program creates 100 threads to execute 100 elements, and each thread spends a significant time to sleeping and synchronize by scheduler. Base on the phenomenon, the time that executes matrix multiplication by n^2 threads increases as a polynomial function and is approximately described as $f(n)=1.2403n^2 - n + 65.654$ by Excel (Fig. 4). As a result, it is evident that the performance is not a linear function and limiting behavior becomes $O(n^2)$.

● **The implication for performance when varying the number of threads**

Compare to another trend in Fig. 4, when increasing the matrix size, the executing time by single thread is significantly lower than using multi-thread. The reason for this phenomenon is that, if the total number of threads is lower 8 of the logical cores (using Intel 8550u), all threads complete their own work parallelly without any thread waiting. However, if the number of threads is over than 8, which means some of threads need to spend time waiting to be executed, every thread consumes a considerable executing time on synchronization as well(Fig. 5). Therefore, when the program executing a lower loading processing, the efficiency of using multi-thread is significantly lower than using single thread.

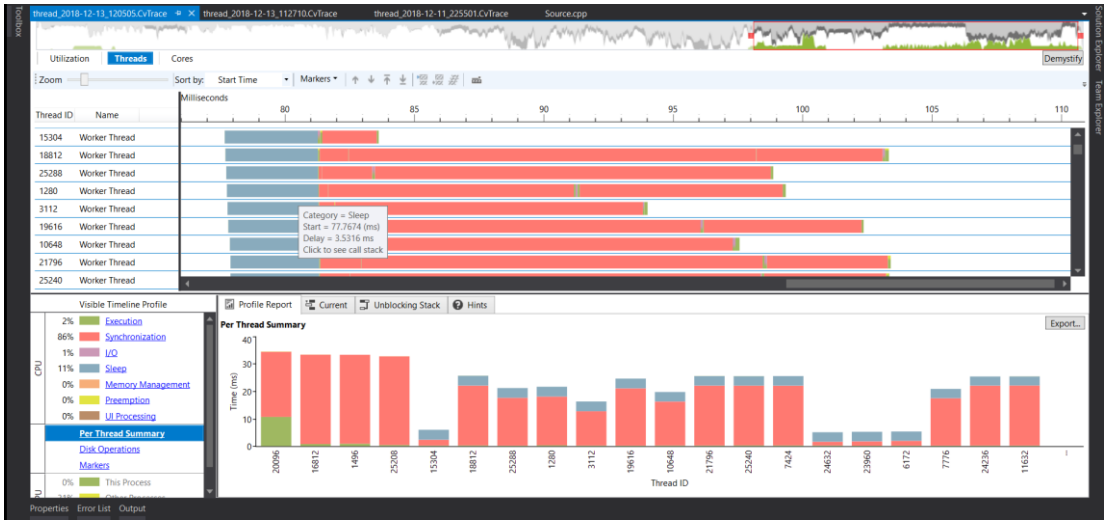


Fig. 5 multi-thread working distribution
(40x40 matrix multiplication by using 1600 thread)

● **Is it possible to achieve roughly same performance by using smaller threads ($< n^2$)**

As the work mention before, the using time increasing dramatically when executing heavy loading by single thread. To alleviate the issue, multi-thread plays an imperative role to enhance the performance. According to fig. 6, in intel 8550u, it cost around 2.85 second for single thread to calculate 1000x1000 matrix multiplication. However, the time drops to 1.4 second when adopting 2 thread to

calculate and stays constant with only 1.1sec by using 50 threads. As the result, due to heavy loading, the characteristic of thread system can be observed easily that with multi-threading , tasks are executed parallelly and each thread calculates their own task in the same machine cycle. It implies that the running time can be reduce efficiently by using multi-thread. However, due to the synchronization, even adopting 50 threads, the time remains at a constant and grows up again while each thread starts to consume time on synchronization.

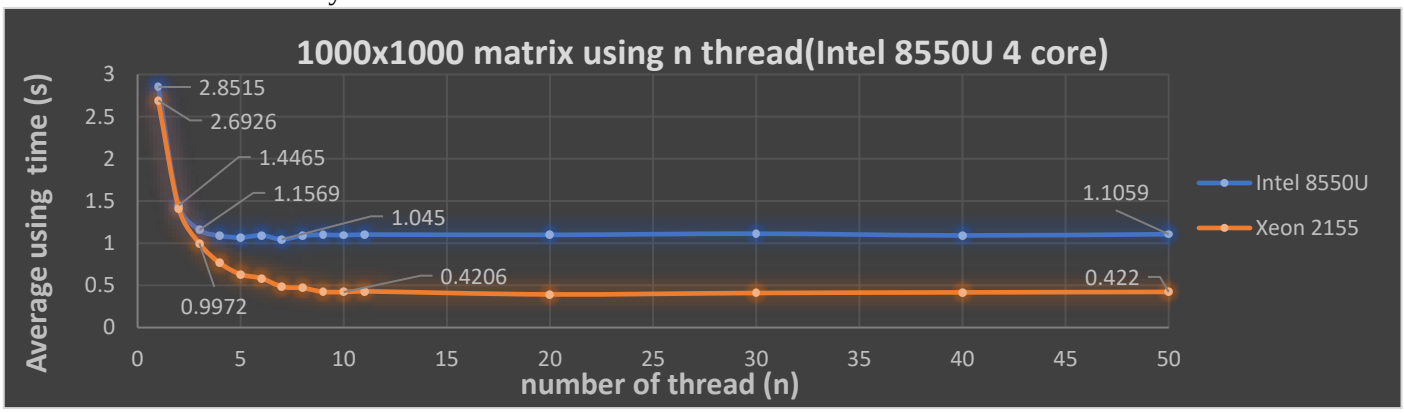


Fig. 6

● Implication by physical cores and cache size

According to fig. 6, the orange figure shows the using time when executing 1000x1000 matrix using intel Xeon 2155 CPU (10 cores 2 physical threads in each core). It is evident that due to the number of physical cores is more than intel 8550u (4 cores 2 physical threads in each core), the executing times can be decreased considerably with only 0.42 second by using 50 threads. When both of computer using double threads, the time depends on the frequency of CPU. With a large number of logical cores, it implies more and more threads are allowed to be allocated to each core in the same machine cycle. In addition, if cache memory is bigger, it suggests each thread would not spends too much time on I/O due to more and more data can be stored in cache at a time.

● Discussion

Initially, in my program, the work assigns the task to threads by dividing and conquering matrix to calculate multiplication such as Strassen's method. However, mapping tasks with this method will become more complicated when increases the number of threads. Alternatively, to make algorithm more intuitive and understandable, the work allocates tasks by dividing total number of elements to threads.

Due to compiler only can allocate 2GB address spaces for program, there is a limitation about number of creating multi-thread. Even a thread had no code and data and the entire memory space are occupied stacks, the program can create at most 2048 threads. Base on the restriction, the work uses `Check_error()` to check whether a thread can be created successfully after executing `CreateThread()`. If the check function occurs error, it returns 1 to let main program know it hits the limitation number of threads. After confirming all threads, main program reassigned the tasks to prevent some of elements from missing or pointing to an empty data address.

III. CONCLUSION

In conclusion, to evaluate whether thread system is a good technology, the answer depends on the process loading. When executing nonheavy load processing, it is worth to adopt single thread. Nevertheless, for multi-thread, due to each thread spending a significant time on synchronization, it is inefficient to use nxn threads to execute matrix multiplication. However, if the number of threads is below number of logical cores, it contributes the highest performance and utilization due to each thread execute parallelly without consuming lots of time on synchronization.

● **Code**

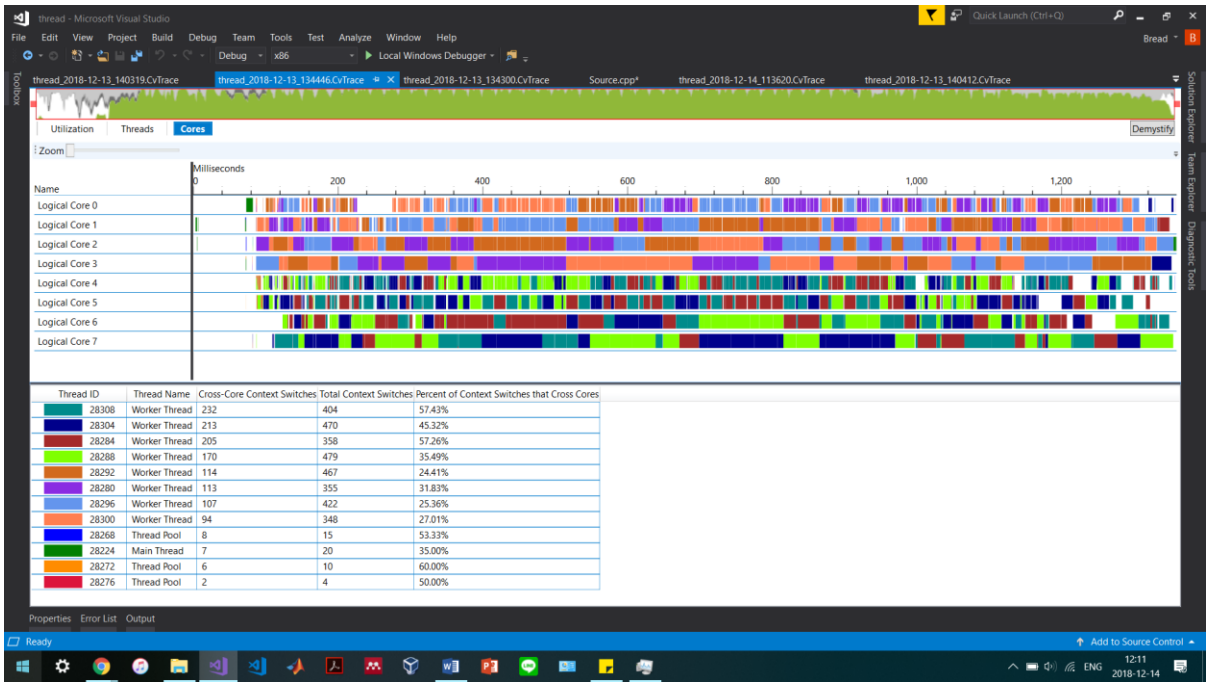
```
#include<stdio.h>
#include<stdlib.h>
#include<windows.h>
#include <time.h>
#pragma warning( disable : 4996 )
int *dynamic_array_1, *dynamic_array_2,*result,*TLB, size, thread_num;

//-----Thread Function-----------------//
int start_opint_calculate(int threadID)
{
	int start_opint = 0;
	for (int id = 0; id < TLB[3 * threadID]; id++)                // TLB[2*id+1]=
The number of elements this thread will execute
		start_opint = start_opint + TLB[3 * id + 1];           //start_opint=
before this thread how many element has been execute ------> thread_3's start point is
thread_1's number of element  + thread_2's number of element
	//printf("Thread %d start point is: %d\n ", threadID, start_opint);
	return start_opint;
}
int execute_element_number(int order_of_element)
{
	int temp = 0, array_1_row, array_2_col;
	array_1_row = order_of_element / size;
	array_2_col = order_of_element % size;

	for (int element = 0; element < size; element++)
		temp = temp + dynamic_array_1[size*array_1_row + element] * dynamic_ar-
ray_2[size*element + array_2_col];
	return temp;
}

DWORD WINAPI Thread_dynamic(LPVOID lpParam)
{
	int id = (int)lpParam,temp;

	//printf("Thread %3d start point is %d  execute to %d\n",id, TLB[3 * id + 2],
TLB[3 * id + 2] +TLB[3 * id+1]-1); //Test Each thread can get their own variable

	for (int element = TLB[3 * id + 2]; element < TLB[3 * id + 2] + TLB[3 * id+1];
element++)
	{
		temp= execute_element_number(element);
		result[element] = temp;
	}
	return 0;
}
//------Dynamic creating array Function-------//
void number_of_each_thread_calculated_Table()
{
	for (int i = 0; i < thread_num; i++)
		printf("Thread %2d start from: %2d to %2d of elements  Execute %4d ele-
ments\n", i, TLB[3 * i + 2], TLB[3 * i + 1]+ TLB[3 * i + 2]-1, TLB[3 * i + 1]);
}
bool Check_error(HANDLE thread1,  DWORD thread_id,int id)
{
	if (thread1 == NULL)
```

```
	{
		printf("Thread creation occur Error, Error num:   ");
		printf("%d\n", GetLastError());
		return 1;
	}
	//printf("Thread creation successful\n");
	//printf("Thread%3d ID: %d\n", id,thread_id);
	return 0;
}
void print_matrix(int *boj ,int n)
{
	for (int row = 0; row < n; row++)
	{
		for (int col = 0; col < n; col++)
			printf("%10d ", *(boj+col+n*row));
		printf("\n");
	}
}
void dynamic_assign_array_data(void)
{
	int square;
	printf("Enter the size of matrix: ");
	scanf("%d", &size);
	square = size * size;
	dynamic_array_1 = new int[square];
	dynamic_array_2 = new int[square];
	result = new int[square];
	for (int i = 0; i < square; i++)
	{
		dynamic_array_1[i] = i;
		dynamic_array_2[i] = square - i;
	}
}
void thread_input(void)
{
	printf("Enter the number of thread: ");
	scanf("%d", &thread_num);
	TLB = new int[3 * thread_num];		// calculate  number of element  for
per thread
}
void assign()
{
	//---------------Assign the number of element per thread will execute----------
--------------------//
/*			TLB:   ID		Time   start point
			 Thread  0		 10		0
			 Thread  1		 10		10
			 Thread  2		 10		20
			 Thread  3		 10		30
*/
	int  remain;
	remain = (size*size) % thread_num;
	for (int i = 0; i < thread_num; i++)
	{
		TLB[3*i] = i;
		TLB[3*i+1] = (size*size) / thread_num;
	}
	for (int i = remain; i > 0; i--)
		TLB[3*(remain - i)+ 1] = TLB[3*(remain - i)+ 1] + 1;
```

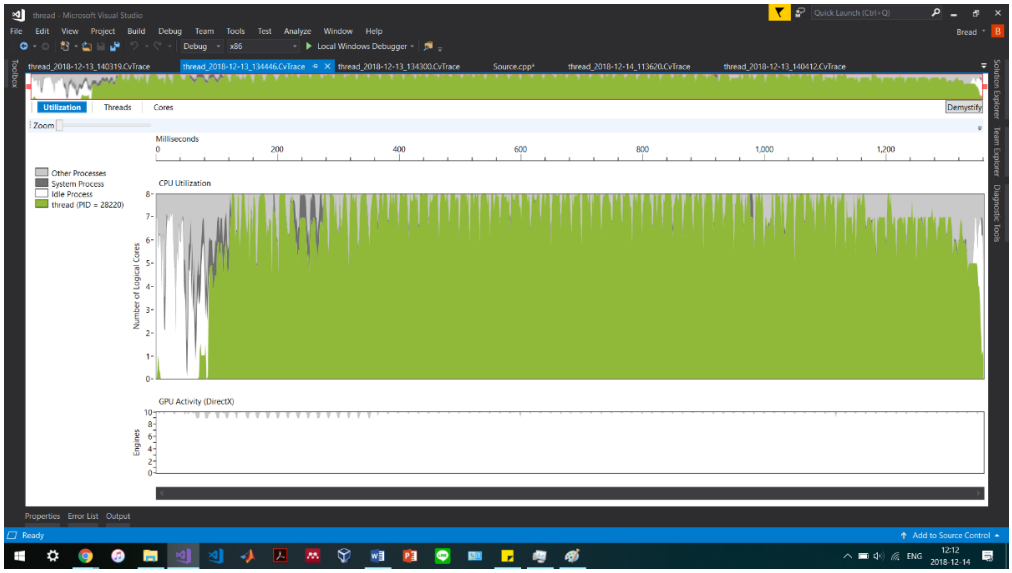
```
	for (int i = 0; i < thread_num; i++)
		TLB[3 * i + 2] = start_opint_calculate(i);
}
void print_input()
{
	printf("-------------dynamic_array_1-------------\n");
	print_matrix(dynamic_array_1, size);
	printf("-------------dynamic_array_2-------------\n");
	print_matrix(dynamic_array_2, size);
}
int main(void)
{
	//---------Create array-----------------------------//
	dynamic_assign_array_data();
	thread_input();
	assign();
	print_input();
	number_of_each_thread_calculated_Table();

//--Threading program & confirm every thread can create  within the limitation---//

	HANDLE *thread = new HANDLE[thread_num];
	DWORD  *thread_id = new DWORD[thread_num];
	for (int i = 0; i < thread_num; i++)
	{
		thread[i] = CreateThread(NULL, 0, Thread_dynamic, (void*)TLB[3 * i],
0x00000004, &thread_id[i]); //(void*) to make variable compactable to the function
		if (Check_error(thread[i], thread_id[i], i))
	//if one of thread occurs error change the total number of thread
		{
			thread_num = i - 1;
			printf("Only can create %d of Threads\n", thread_num);
			assign();		//	reassigned
			break;
		}
	}
	//--------Enable every Thread to start execute matrix moultiplication----------
-----//
	clock_t time = clock();
	for (int i = 0; i < thread_num; i++)
		ResumeThread(thread[i]);

	//------------Wait for all thread to complete & Print the execute time---------
----//
	for (int i = 0; i < thread_num; i++)
		WaitForSingleObject(thread[i], INFINITE);
	time = clock() - time;
	printf("Exetution time: %4f Sec \n", ((double)time) / CLOCKS_PER_SEC);

	//---------- Release the Thread---------------------//
	for (int i = 0; i < thread_num; i++)
	{
		CloseHandle(thread[i]);
		TerminateThread(thread[i], NULL);
	}

	//-----------Print result-----------------------------//
		printf("-------------result-------------\n");
		print_matrix(result, size);
```

```
        //-----clear all Register & release the memory-------//
        free(dynamic_array_1);
        free(dynamic_array_2);
        free(TLB);
        free(thread);
        free(thread_id);
        free(result);

        system("pause");
        return 0;
}
```

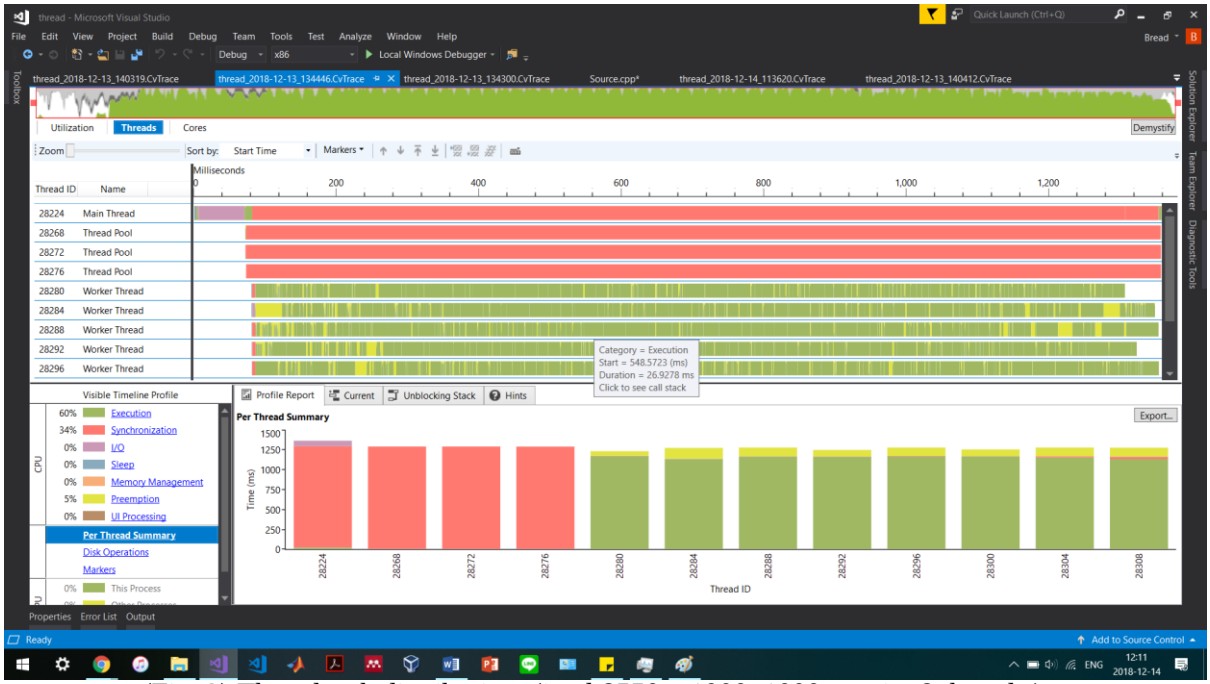
● Appendix



(Fig. 1)Threads distribution in physical cores (intel 8550u, 1000x1000 matrix, 8 threads)



(Fig. 2) CPU utilization (intel 8550u, 1000x1000 matrix , 8 threads)



(Fig. 2) Thread task distribution (intel 8550u, 1000x1000 matrix, 8 threads)