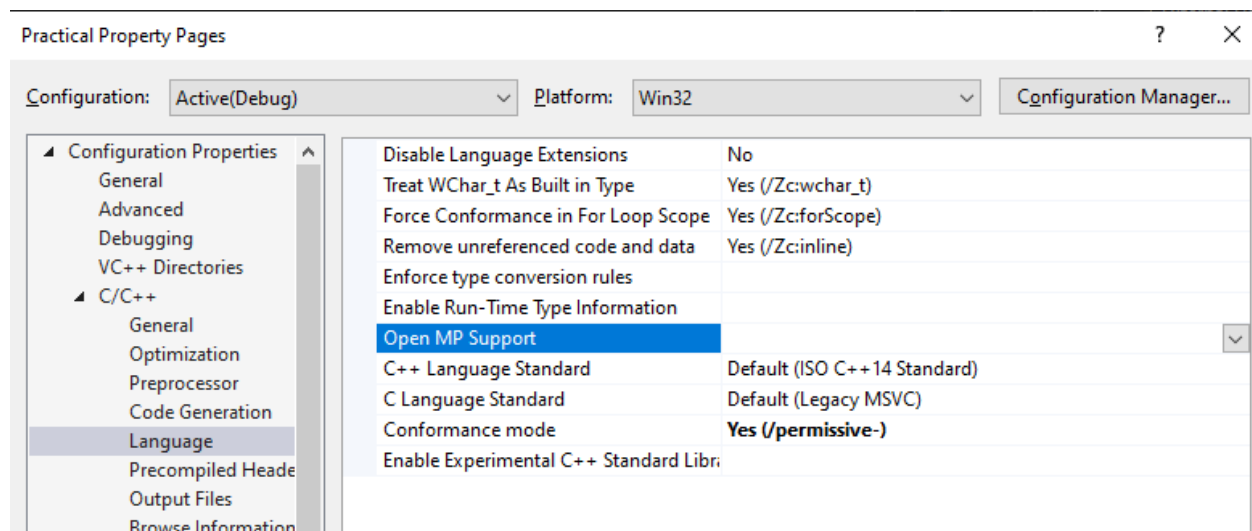


Resources: [OpenMP - Official Website](#)

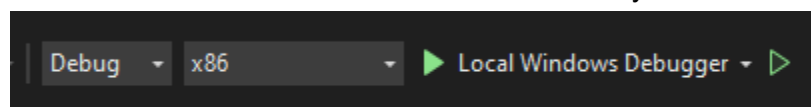
Tutorial from OpenMP: [OpenMP 101 \(ACF Spring Training Workshop\)](#)

Official OpenMP in Visual C++ Documentation: [MSDN Documentation on OpenMP](#)

Pre-configuration in Visual Studio, in your Project Properties > Configuration Properties > C/C++ > Language > Open MP Support > select **Yes (/openmp)**



Make sure the selected Platform is the same as your Build Solution selected Platform:



Hints: **Win32** is the same as **x86**

Question 1:

Write a program, named P3Q1.cpp that displays Hello World with OpenMP. This is to check whether you have OpenMP configured.

```
#include "omp.h"

int main()
{
    #pragma omp parallel
    {
        int ID = omp_get_thread_num();
        printf("hello(%d)", ID);
        printf("world(%d)\n", ID);
    }
    return 0;
}
```

Output:

```
hello(1)world(1)
hello(0)world(0)
hello(2)world(2)
hello(5)world(5)
hello(3)world(3)
hello(4)world(4)
hello(7)world(7)
hello(6)world(6)
```

OpenMP topic: Controlling thread data

In a parallel region there are two types of data: private and shared. In this section, we will see the various ways you can control what category your data falls under; for private data items we also discuss how their values relate to shared data.

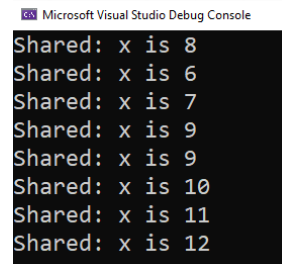
Shared Data

In a parallel region, any data declared outside it will be shared: any thread using a variable `x` will access the same memory location associated with that variable.

Example:

```
int x = 5;
#pragma omp parallel
{
    x = x+1;
    printf("shared: x is %d\n",x);
}
```

Output:



Microsoft Visual Studio Debug Console

```
Shared: x is 8
Shared: x is 6
Shared: x is 7
Shared: x is 9
Shared: x is 9
Shared: x is 10
Shared: x is 11
Shared: x is 12
```

All threads increment the same variable, so after the loop it will have a value of five plus the number of threads; or maybe less because of the data races involved.

Sometimes this global update is what you want; in other cases the variable is intended only for intermediate results in a computation. In that case there are various ways of creating data that is local to a thread, and therefore invisible to other threads.

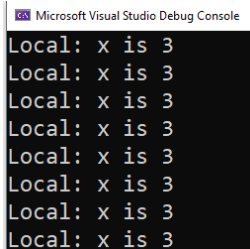
Private Data

In the C/C++ language it is possible to declare variables inside a *lexical scope* ; roughly: inside curly braces. This concept extends to OpenMP parallel regions and directives: any variable declared in a block following an OpenMP directive will be local to the executing thread.

Example:

```
int x = 5;
#pragma omp parallel
{
    int x; x = 3;
    printf("local: x is %d\n",x);
}
```

Output:



Microsoft Visual Studio Debug Console

```
Local: x is 3
Local: x is 3
Local: x is 3
Local: x is 3
Local: x is 3
Local: x is 3
Local: x is 3
Local: x is 3
Local: x is 3
```

After the parallel region the outer variable `x` will still have the value `5` : there is no *storage association* between the private variable and global one.

The private directive declares data to have a separate copy in the memory of each thread. Such private variables are initialized as they would be in a main program. Any computed value goes away at the end of the parallel region. (However, see below.) Thus, you should not rely on any initial value, or on the value of the outer variable after the region.

Example:

```
int x = 5;
#pragma omp parallel
{
    x = x+1; // dangerous
    printf("private: x is %d\n",x);
}
printf("after: x is %d\n",x); // also dangerous
```

Output:

```
Microsoft Visual Studio Debug Console
private: x is 6
private: x is 13
private: x is 9
private: x is 7
private: x is 10
private: x is 11
private: x is 12
private: x is 8
after: x is 13
```

Question 2:

Write a program that computes the numerical integration below:

Mathematically, we know that,

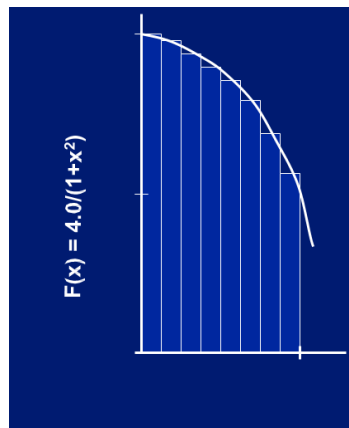
$$F(x) = \int_0^1 \frac{4.0}{(1+x^2)} dx = \pi$$

We can approximate the sum of rectangles

$$\sum_{i=0}^N \Delta x F(x_i) \approx \pi$$

Where each rectangle has width Δx and height $F(x_i)$ at the middle of the interval i as

shown in the figure.



The serial code is shown as below and obtained from here [P3Q2](#):

```
#include "stdafx.h"
#include "omp.h"

static long num_steps = 100000;
double step;

int main()
{
    int i; double x, pi, sum = 0.0;
    step = 1.0 / (double)num_steps;

    for (i = 0; i < num_steps; i++) {
        x = (i + 0.5)*step;
        sum = sum + 4.0 / (1.0 + x * x);
    }
    pi = step * sum;
    printf("%f", pi);
    return 0;
}
```

- a) Create a parallel version of the pi program using a parallel construct.

Use runtime library routines such as the followings:

`int omp_get_num_threads();`

Number of threads in the team

`int omp_get_thread_num();`

Thread ID or rank

`double omp_get_wtime();`

Time in seconds since a fixed point in the past

- b) While the code in a) is correct, it may be inefficient because of a phenomenon called **false sharing**. Even though the threads write to separate variables, those variables are likely to be on the same cache line. This means that the cores will be wasting a lot of time and bandwidth updating each other's copy of this cache line.

False sharing can be prevented by giving each thread its own cache line, modify the code using padding as shown in the example below:

`sum[i][pad]`

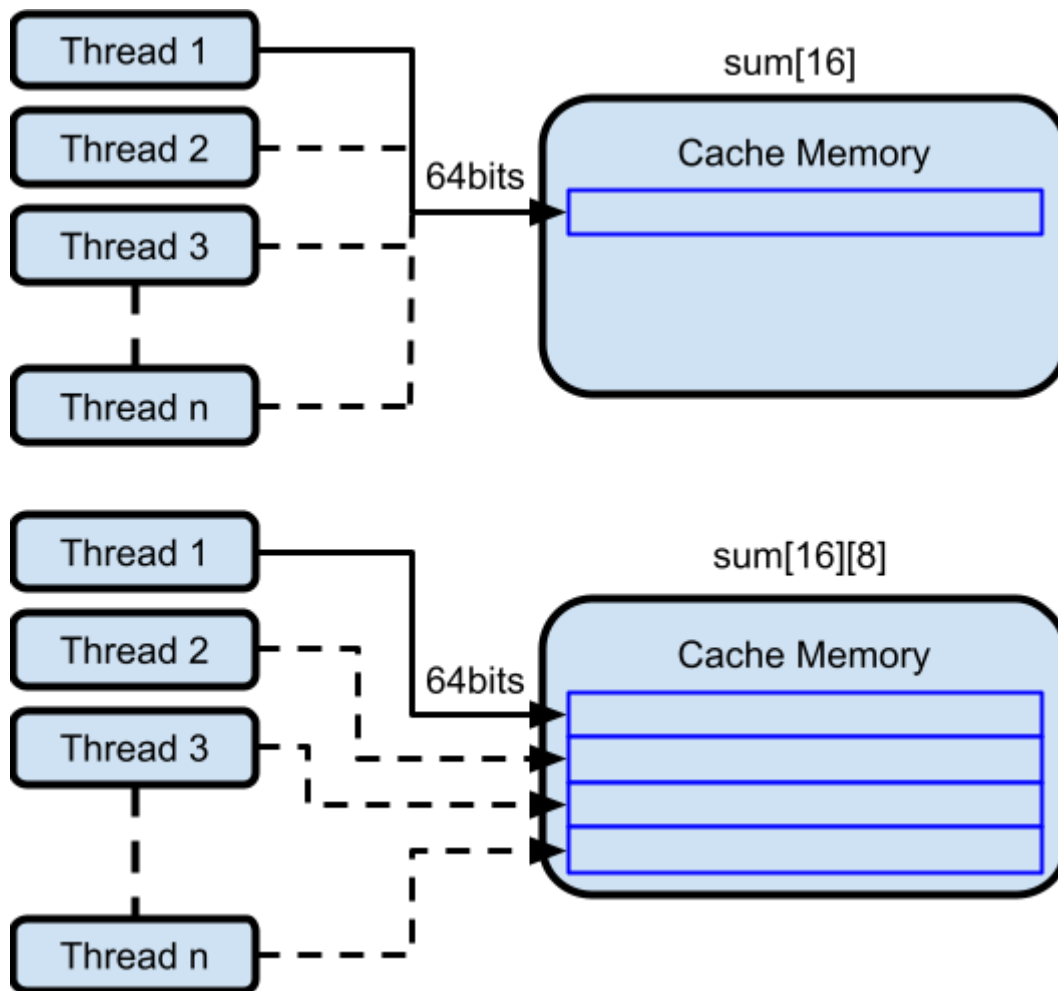


Illustration on Cache line(s)

* Compare the results between non-PAD and PAD, with
thread number = 8 and num_steps = 10000000 .

- c) The OpenMP reduction clause lets you specify one or more thread-private variables that are subject to a reduction operation at the end of the parallel region. OpenMP pre defines a set of reduction operators. Each reduction variable must be a scalar (for example, int, long, and float). OpenMP also defines several restrictions on how reduction variables are used in a parallel region.

Modify the code using `#pragma omp for reduction(+:sum)`.

Reference:

<https://docs.microsoft.com/en-us/cpp/parallel/concrt/convert-an-omp-loop-that-uses-a-reduction-variable?view=msvc-170>

Output:

```
C:\WINDOWS\system32\cmd.exe
num_threads = 1
pi is 3.141593 in 0.907559 seconds and 1 threads
num_threads = 2
pi is 3.142932 in 0.636558 seconds and 2 threads
num_threads = 3
pi is 3.142014 in 0.625026 seconds and 3 threads
num_threads = 4
pi is 3.141898 in 0.621494 seconds and 4 threads
num_threads = 5
pi is 3.142028 in 0.770418 seconds and 5 threads
num_threads = 6
pi is 3.142065 in 0.718954 seconds and 6 threads
num_threads = 7
pi is 3.142139 in 0.715128 seconds and 7 threads
num_threads = 8
pi is 3.142018 in 0.648263 seconds and 8 threads
Press any key to continue . . .
```

* Discuss why the pi value is different with different numbers of thread.

Question 3.

Parallelize the matrix multiplication program in the file [P3Q3.c](#). Then evaluate the time required to execute the program.

(Hints: Use `#pragma omp parallel for private(.....)`)

Reference:

<https://docs.microsoft.com/en-us/cpp/parallel/openmp/reference/openmp-clauses?view=msvc-170>

Example answer

```
Microsoft Visual Studio Debug Console
Order 1000 multiplication in 0.491261 seconds
Order 1000 multiplication at 4071.151515 mflops

Hey, it worked
all done
```

* Compare the results with no having `#pragma omp parallel for private(.....)`

Compute Unified Device Architecture (CUDA)

* If CUDA is not installed properly, may need to download the latest version from here:

<https://developer.nvidia.com/cuda-downloads>

* For latest Visual Studio 2022 (Community Version), download from here

<https://visualstudio.microsoft.com/vs/>

[OPTIONAL] Setting up CUDA project in Visual Studio 2019 *only if you are using VS2019

<https://medium.com/@aviatorx/c-and-cuda-project-visual-studio-d07c6ad771e3>

Memory Allocation in CUDA

To compute on the GPU, you need to allocate memory accessible by the GPU. Unified Memory in CUDA makes this easy by providing a single memory space accessible by all GPUs and CPUs in your system. To allocate data in unified memory, call `cudaMallocManaged()`, which returns a pointer that you can access from host (CPU) code or device (GPU) code. To free the data, just pass the pointer to `cudaFree()`.

Launch the `functionCall()` kernel, which invokes it on the GPU. CUDA kernel launches are specified using the triple angle bracket syntax `<< < > >>`.

Call `cudaDeviceSynchronize()` before doing the final error checking on the CPU.

Save this code in a file called `FileName.cu` and compile it with `nvcc`, the CUDA C++ compiler.

In summary, the steps to allocate memory, implement program parallelization and free memory are as follows:

- allocate device memory
- copy host memory to device
- initialize thread block and kernel grid dimensions
- invoke CUDA kernel
- copy results from device to host
- deallocate device memory

Question 4:

Perform the above steps for Vector Addition, then use *nvprof* (Nvidia Profiler) to evaluate the performance for the codes below:

- (i) [Serial Vector Addition](#)
- (ii) [Parallel Vector Addition](#)

* To run *nvprof* and *nvcc* in command prompt, we may need to include the following two paths in Run (or Windows+R), type *sysdm.cpl* and press Enter,

System Properties > Advanced > Environment Variable> System Variable > Paths > Add New

D:\Program Files (x86)\Microsoft Visual
Studio\2019\Professional\VC\Tools\MSVC\14.29.30133\bin\Hostx64\x86

- To get *cl.exe* when running *nvcc*

C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v11.7\extras\CUPTI\lib64

- To get *cupti64_XXXX.dll* when running *nvprof*

Output:

In powershell, type:

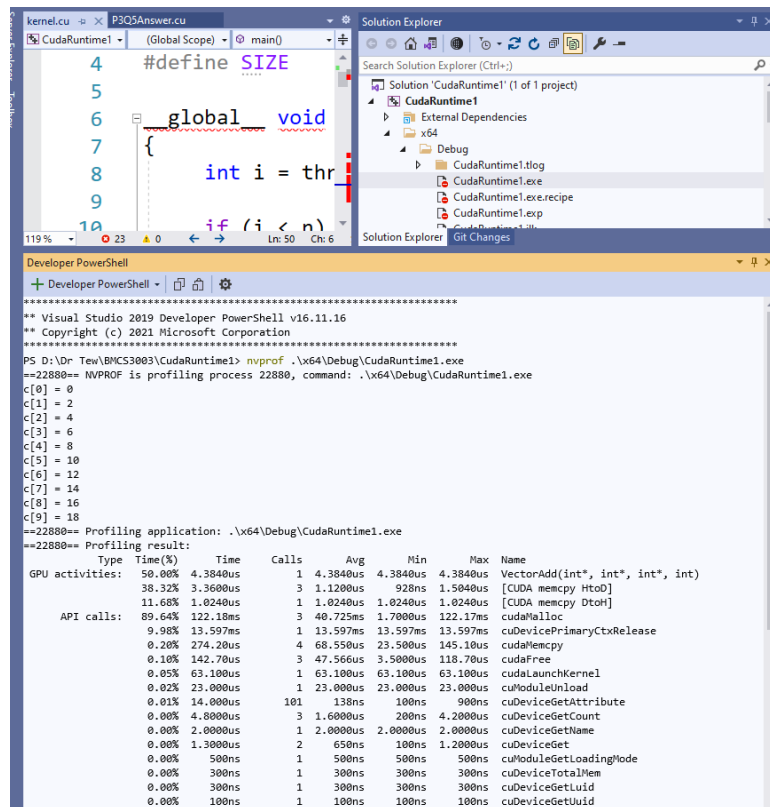
`nvprof YOUR_COMPILED_PROJECT_EXE_PATH.exe`

```
==6100== NVPROF is profiling process 6100, command: VectorAdd
c[0] = 0
c[1] = 2
c[2] = 4
c[3] = 6
c[4] = 8
c[5] = 10
c[6] = 12
c[7] = 14
c[8] = 16
c[9] = 18
==6100== Profiling application: VectorAdd
==6100== Profiling result:
   Type      Time          Calls      Avg      Min      Max      Name
GPU activities: 46.63% 5.1200us      3 1.7060us 1.5680us 1.9200us [CUDA memcpy HtoD]
                44.02% 4.8320us      1 4.8320us 4.8320us 4.8320us VectorAdd(int*, int*, int, int)
                9.33% 1.0240us      1 1.0240us 1.0240us 1.0240us [CUDA memcpy DtoH]
API calls: 81.54% 260.76ms      3 86.919ms 3.9000us 260.74ms cudaMalloc
                17.82% 56.983ms      1 56.983ms 56.983ms 56.983ms cuDevicePrimaryCtxRelease
                0.32% 1.0088ms      1 1.0088ms 1.0088ms 1.0088ms cuModuleLoad
                0.09% 293.00us      3 97.666us 5.5000us 262.80us cudaFree
                0.09% 291.40us      4 72.850us 52.900us 87.300us cudaMemcpy
                0.08% 269.80us      1 269.80us 269.80us 269.80us cuDeviceGetPCIBusId
                0.02% 63.900us      1 63.900us 63.900us 63.900us cudaLaunchKernel
                0.02% 55.300us      1 55.300us 55.300us 55.300us cuDeviceTotalMem
                0.01% 41.700us      97 429ms      100ns 18.100us cuDeviceGetAttribute
                0.00% 4.2000us      3 1.4000us      500ns 3.2000us cuDeviceGetCount
                0.00% 2.3000us      1 2.3000us      200ns 2.3000us cuDeviceGetName
                0.00% 1.9000us      2 950ns      200ns 1.7000us cuDeviceGet
                0.00% 400ns      1 400ns      400ns 400ns cuDeviceGetLuid
                0.00% 300ns      1 300ns      300ns 300ns cuDeviceGetLuid
```

OR

In visual studio terminal, type:

`nvprof .\x64\Debug\YOUR_PROJECT_NAME.exe`



The screenshot shows the Visual Studio IDE with a CUDA kernel file open. The kernel code is as follows:

```
kernel.cu
1 #define SIZE 10
2
3
4 global void
5 {
6     int i = thr
7     if (i < n)
8     {
9         // ...
10    }
```

The Solution Explorer on the right shows the project structure for 'CudaRuntime1'. The Developer PowerShell window at the bottom displays the output of the nvprof command, showing the profiling results for the application.

```
PS D:\Dr_Tew\BMCS3003\CudaRuntime1> nvprof .\x64\Debug\CudaRuntime1.exe
==22880== Nvprof is profiling process 22880, command: .\x64\Debug\CudaRuntime1.exe
c[0] = 0
c[1] = 2
c[2] = 4
c[3] = 6
c[4] = 8
c[5] = 10
c[6] = 12
c[7] = 14
c[8] = 16
c[9] = 18
==22880== Profiling application: .\x64\Debug\CudaRuntime1.exe
==22880== Profiling result:
Type Time(%) Time Calls Avg Min Max Name
GPU activities: 50.00% 4.3840us 1 4.3840us 4.3840us 4.3840us VectorAdd(int*, int*, int*, int)
38.32% 3.3600us 3 1.1200us 928ns 1.5040us [CUDA memcpy HtoD]
11.68% 1.0240us 1 1.0240us 1.0240us 1.0240us [CUDA memcpy DtoH]
API calls: 89.64% 122.18ms 3 40.725ms 1.7000ms 122.17ms cudaMalloc
9.98% 13.597ms 1 13.597ms 13.597ms 13.597ms cuDevicePrimaryCtxRelease
0.20% 274.20us 4 68.550us 23.500us 145.10us cudaMemcpy
0.10% 142.70us 3 47.566us 3.5000us 118.70us cudaFree
0.05% 63.100us 1 63.100us 63.100us 63.100us cudaLaunchKernel
0.02% 23.000us 1 23.000us 23.000us 23.000us cuModuleUnload
0.01% 14.000us 101 138ns 100ns 900ns cuDeviceGetAttribute
0.00% 4.8000us 3 1.6000us 200ns 4.2000us cuDeviceGetCount
0.00% 2.0000us 1 2.0000us 2.0000us 2.0000us cuDeviceGetName
0.00% 1.3000us 2 650ns 100ns 1.2000us cuDeviceGet
0.00% 500ns 1 500ns 500ns 500ns cuModuleGetLoadingMode
0.00% 300ns 1 300ns 300ns 300ns cuDeviceTotalMem
0.00% 300ns 1 300ns 300ns 300ns cuDeviceGetLuid
0.00% 100ns 1 100ns 100ns 100ns cuDeviceGetUuid
```

Question 5:

Using the same steps, modify the code for Matrix Multiplication that can be obtained here [P3Q5](#).

- allocate device memory
`cudaMalloc((void **)&device_memory, MAX_BYTES);`
- copy host memory to device
`cudaMemcpy(device_memory, host_memory, MAX_BYTES, cudaMemcpyHostToDevice);`
- initialize thread block and kernel grid dimensions, and invoke CUDA kernel
`matrix_mul << < CEIL(MAX_SIZE, 1024), 1024 >> >(source, source, destination);`
- copy results from device to host
`cudaMemcpy(host_memory, device_memory, MAX_BYTES, cudaMemcpyDeviceToHost);`
- free the memories at the correct location

Note: Adapt the variables that are highlighted in RED.

Output:

The solution is correct