# Parallel Computing Platform

⇒ Environment that allow multi-core CPUs, GPUs work together on a computational problem

communication channel

Example

1) OMP (Open MP - Open Multi-Processing) ⇒ Shared memory (difficult to do distributed system)

2) MPI (Message Passing Interface) ⇒ Message Passing

3) CUDA (must use NVIDIA GPU to run CUDA)

1 CPU have multiple core, 1 GPU have a lot core

EG: Normal CPU have around 16/32 core

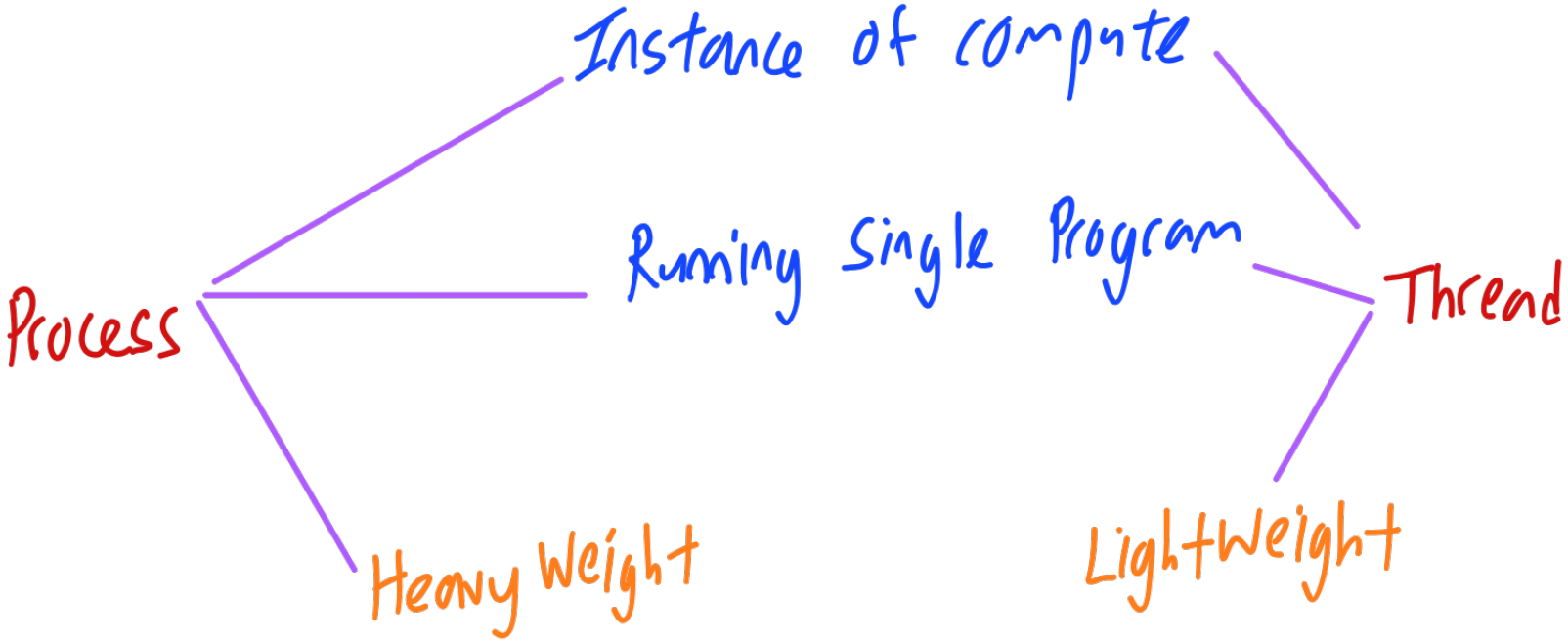
But GPU can have 2048 core

(only can utilize fully with lower level programming)

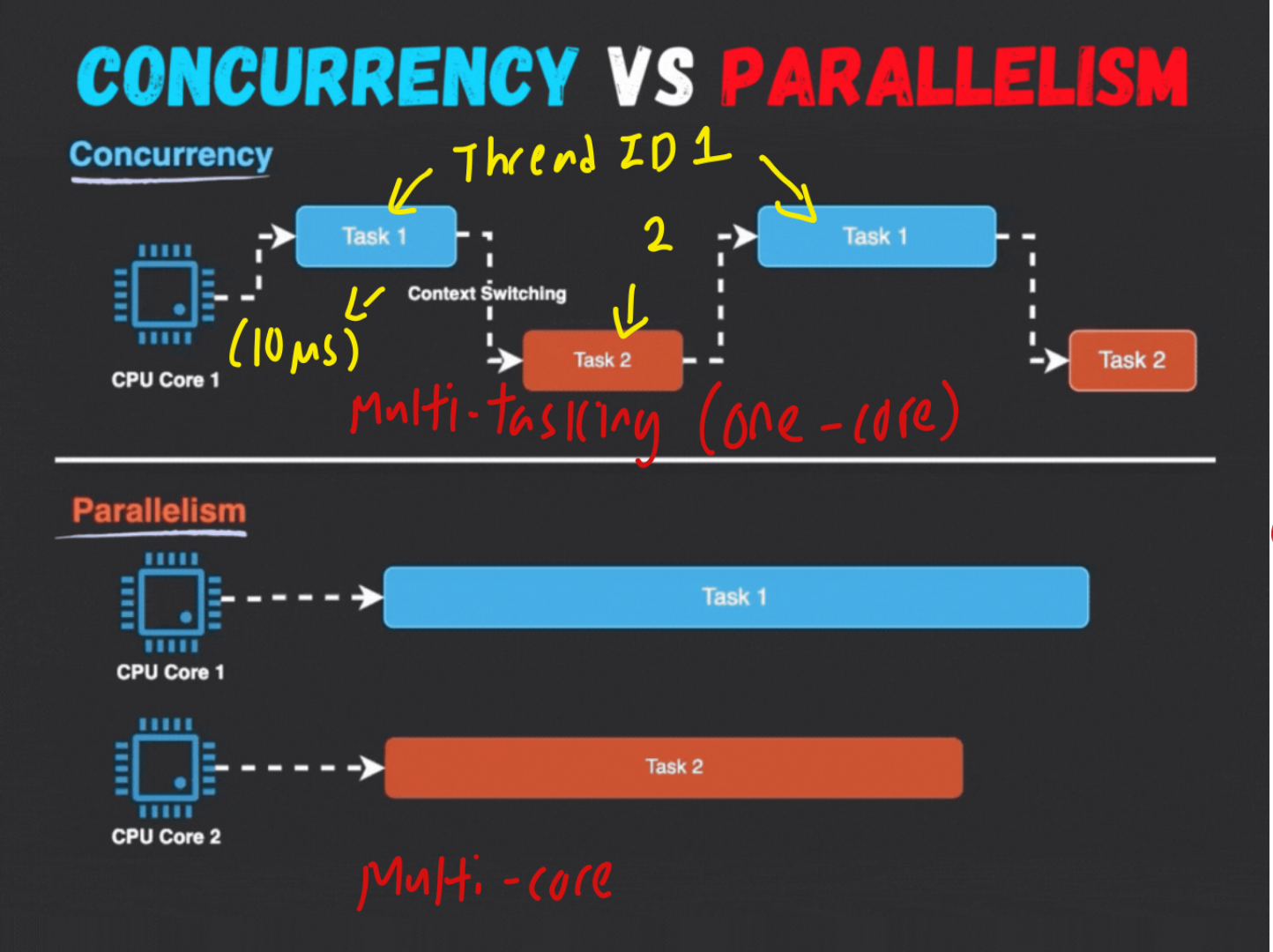
However, CUDA can utilize around 1024 core

(a lot!!! compare to CPU)

# Process VS Thread



# CONCURRENCY VS PARALLELISM




multiple threads run on single core (switch one to another thread)

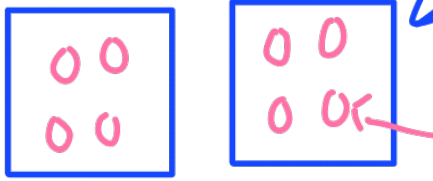
---

each thread/process run on diff core at the same time

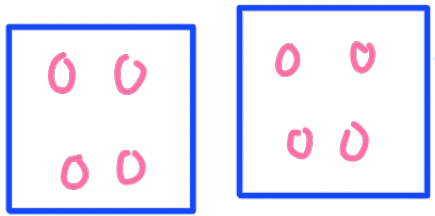
1 core ← used by 1 Thread (best case)

Not Enough Core to used by a lot of Thread?

EG: 4 cores, 16 threads ☆ Parallel 4 core ⇒ 4x improve speed




Each core run 4 thread



parallel with concurrency at the same time (other case)

# Race Condition - multiple thread compete to each other race

```cpp
#include <iostream>
#include <omp.h>

void ordinary_demo() {
    std::cout << "Normal Hello World!" << std::endl;
}

// Running parallel process, os will assign a thread id to each process
void omp_demo() {
    #pragma omp parallel
    { // this bracket must be on the next line, pragma does not understand {
        int id = omp_get_thread_num();
        std::cout << "Hello World!" << id << std::endl;
    }
}

int main() {
    ordinary_demo();
    omp_demo();
    return 0;
}
```

① create a team of thread, each thread execute the code inside {} independently

② unique Id (thread Id) assign to each thread on the team

a) All threads are trying to write std::out at the same time (concurrent access to std::out)

b) std::out is not thread safe, meaning multiple threads writing to it at the same time, causing output unpredictable

```
Normal Hello World!
Hello World!Hello World!Hello World!Hello World!02
13
```

What happens under the Hood?

① Thread 0 write "Hello World!"

② Before it can print the thread ID (0),
Thread 1 writes "Hello World!"

③ The output mixed out, causing unpredictable

# Cout vs Printf

```cpp
#include <iostream>
#include <omp.h>

void omp_demo_cout() {
    #pragma omp parallel
    {
        int id = omp_get_thread_num();
        std::cout << "Cout: Hello World!" << id << std::endl;
    }
}

void omp_demo_printf() {
    #pragma omp parallel
    {
        int id = omp_get_thread_num();
        printf("Printf: Hello World! %d\n",id);
    }
}

int main() {
    omp_demo_cout(); //Not Thread Safe
    omp_demo_printf(); //Thread Safe
    return 0;
}
```

```
Cout: Hello World!Cout: Hello World!23

Cout: Hello World!1
Cout: Hello World!0
Printf: Hello World! 2
Printf: Hello World! 1
Printf: Hello World! 0
Printf: Hello World! 3
```

if multiple threads are write to same output stream, the order of line printed is not guarantee

Generally Thread Safe

(printf in diff threads should not interfere each other)

(entire output of printf as a single uninterrupted text)

printf use internal locking mechanism (mutex) ensure output is atomic

# Shared Data

Sharing data is good, but there is a downside

↓ can cause

Race condition

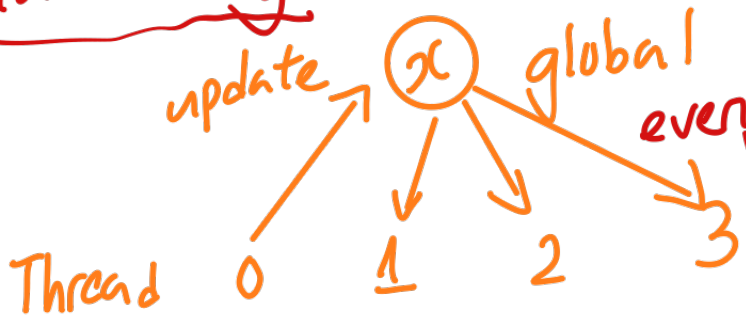
If data can share ⇒ make it public

If data no sharing ⇒ make it private

```cpp
#include <iostream>
#include <omp.h>

void q1_data_sharing() {
    int x = 5; //global

    #pragma omp parallel
    {
        x = x + 1; //accessing global
        printf("shared: x is %d\n", x);
    }
}

// Everybody will have their own copy of x
void q1_no_data_sharing() {
    int x = 5; //global

    #pragma omp parallel
    {
        int x; x = 3; //local
        x = x + 1; //accessing local
        printf("local: x is %d\n", x);
    }
    printf("global: x is %d\n", x);

}

int main() {
    q1_data_sharing();
    q1_no_data_sharing();
    return 0;
}
```

```
shared: x is 7
shared: x is 6
shared: x is 8
shared: x is 9
local: x is 4
local: x is 4
local: x is 4
local: x is 4
global: x is 5
```

Data sharing




everybody will notify that x have been updated

thread no guarantee execute in order (race condition)

# Private Data

```cpp
#include <iostream>
#include <omp.h>

void q1_no_data_sharing() {
    int x = 5; //global

    #pragma omp parallel
    {
        int x; x = 3; //local
        x = x + 1; //accessing local
        printf("local: x is %d\n", x);
    }
    printf("global: x is %d\n", x);

}
```

Output

```
local: x is 4
local: x is 4
local: x is 4
local: x is 4
global: x is 5
```

```cpp
void q1_no_data_sharing_v2() {
    int x = 5; //global

    #pragma omp parallel private(x) // create local x
    {
        x = 3;
        x = x + 1;
        printf("local: x is %d\n", x);
    }
    printf("global: x is %d\n", x);

}

void q1_no_data_sharing_v3() {
    int x = 5;

    #pragma omp parallel firstprivate(x)
    {
        x = x + 1;
        printf("local: x is %d\n", x);
    }
    printf("global: x is %d\n", x);

}
```

local x

copy value

```
local: x is 4
local: x is 4
local: x is 4
local: x is 4
global: x is 5
```

```
local: x is 6
local: x is 6
local: x is 6
local: x is 6
global: x is 5
```

Private Data

(x) global

Thread 0 ↓ x  1 ↓ x  2 ↓ x  3 ↓ x

Each thread have it own copy of x

## Python scope

```
x = 3   ← global
if ... :
    x = 5   ← local
```

---

```
x = 3   ← global
if ... :
    global x   ← global
    x = 5   ← global will change
```

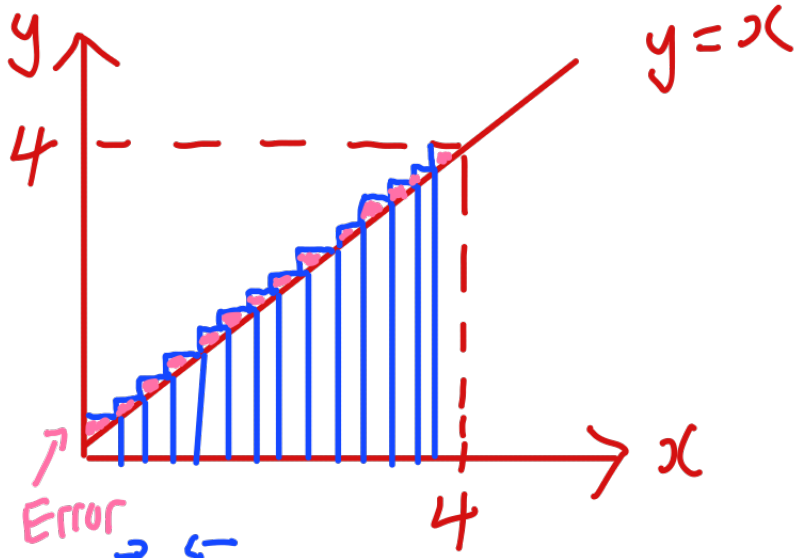
## C++ scope

```
int x=3 ← global
{
    int x; x=5 ← local
}
```

---

```
int x = 3
{
    x = 5 ← global
}
```

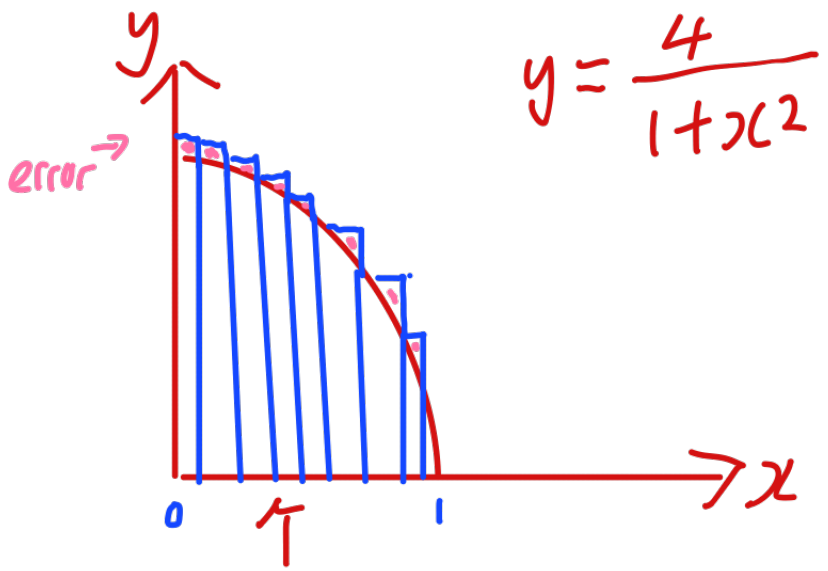
will change



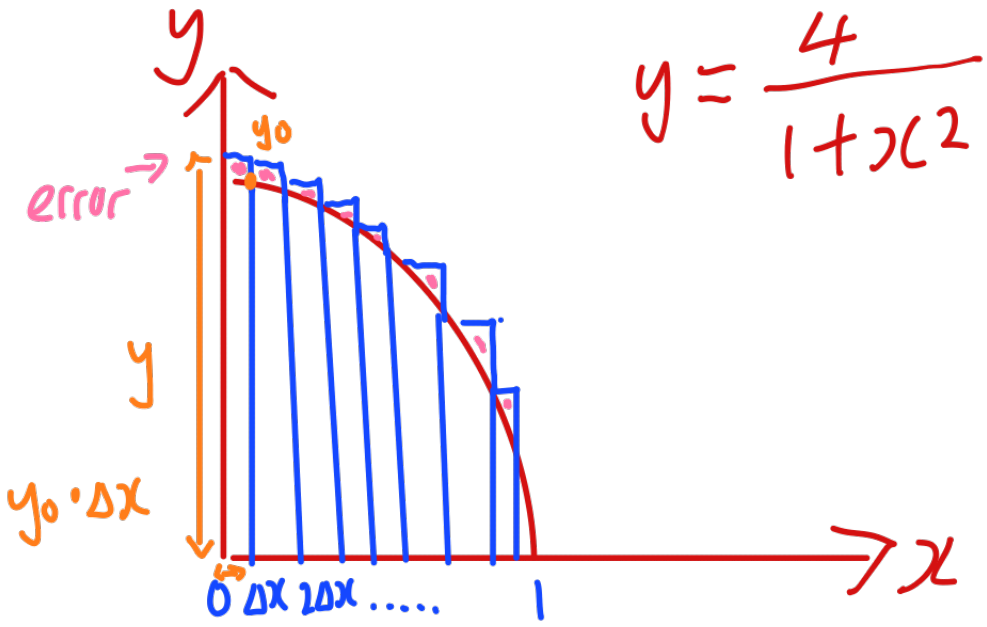

as $\Delta x \to 0$

tiny change of $x$ close to 0 $\Rightarrow$ better estimation

$$\text{Area} = \frac{1}{2} \times 4 \times 4 = 16$$




Hard to calculate area




$$y = \frac{4}{1+x^2}$$

$$A_1 = \frac{4}{1+\Delta x^2} \cdot \Delta x$$

$$A_2 = \frac{4}{1+(2\Delta x)^2} \cdot \Delta x$$

↓ General Formula

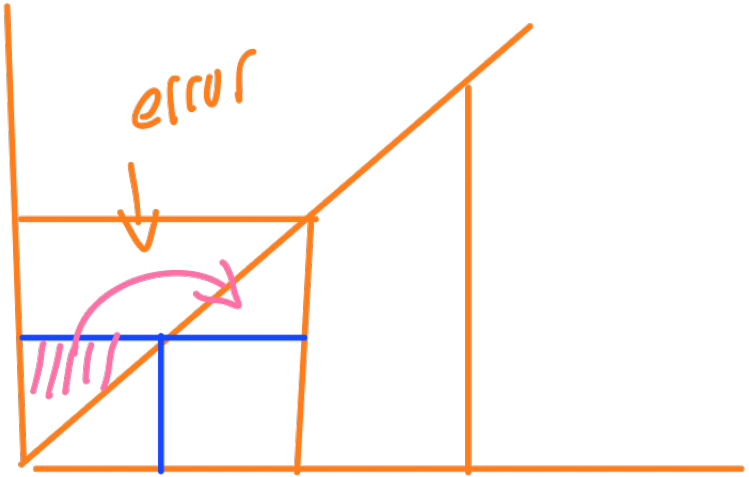
$$A_n = \frac{4}{1+(n \cdot \Delta x)^2}$$

i * step

$$\text{Step} = \Delta x = \frac{1-0}{100,000,000}$$ (chop down 1 million)

why $x = (i - 0.5) * step$

why -0.5




become the actual area

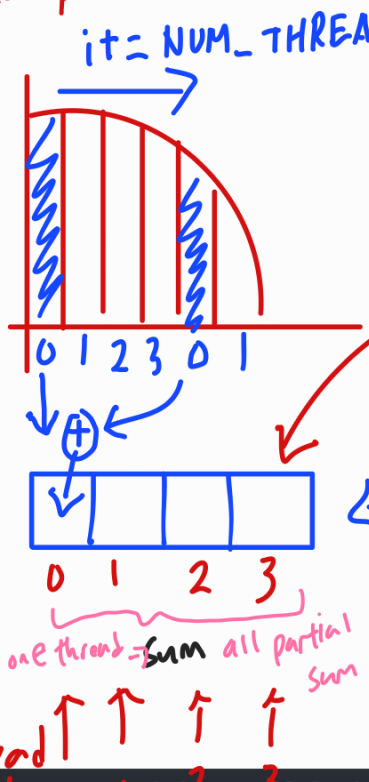
→ more accurate

Non-parallel

```cpp
#include <iostream>
#include <omp.h>

static long num_steps = 100000000;
double step;
double compute_pi()
{
    int i;
    double x, pi, sum = 0.0;
    double start_time, run_time;

    step = 1.0 / (double)num_steps;

    start_time = omp_get_wtime();

    for (i = 1; i <= num_steps; i++)
    {
        x = (i - 0.5) * step;
        sum = sum + 4.0 / (1.0 + x * x);
    }       You, 7 minutes ago • Add Practical3 Code

    pi = step * sum;
    run_time = omp_get_wtime() - start_time;
    printf("Non-Parallel: pi with %ld steps is %.16lf in %lf seconds\n ", num_steps, pi, run_time);

    return run_time;
}
```

# Parallel (with storing result to array)

## Example of 4 Thread




outside parallel thread, one thread will ← loop through this array and sum up

Variable ⇒ decide public/private

i ⇐ private
⇒ each thread have it own i (i not update randomly cause race condition

num_steps ⇐ public
⇒ same almost all threads and only retrieve only

x ⇐ private

sum ⇐ public ⇒ partial_sum[NUM_THREADS]

```
double compute_pi_in_parallel_v1()
{
    int i;
    double x, pi, sum = 0.0;
    double start_time, run_time;
    const int NUM_THREADS = 16;
    double partial_sum[NUM_THREADS];

    for (int i = 0; i < NUM_THREADS; i++)
    {
        partial_sum[i] = 0;
    }

    step = 1.0 / (double)num_steps;

    start_time = omp_get_wtime();

#pragma omp parallel num_threads(NUM_THREADS) private(x, i)
    {
        int id = omp_get_thread_num();
        for (i = id + 1; i <= num_steps; i += NUM_THREADS)
        {
            x = (i - 0.5) * step;
            partial_sum[id] += 4.0 / (1.0 + x * x);
        }
    }

    for (int i = 0; i < NUM_THREADS; i++)
    {
        sum += partial_sum[i];
    }

    pi = step * sum;
    run_time = omp_get_wtime() - start_time;
    printf("Version 1 (Parallel): pi with %ld steps is %.16lf in %lf seconds\n ", num_steps, pi, run_time);

    return run_time;
}
```

Solution

# ALU (Arimetric Logic Unit) + Memory

⇒ To process data, the ALU requires the data to be transfelled from external repositoty to register

1) Register (Very fast)

⇒ Common Registers: AX, BX, CX, DS

⇒ operate at the same clock speed as CPU

EG: a CPU running at 4GH$_2$, register also operate at 4GH$_2$

↓

1GH$_2$ = 1 nanoseconds (ns) per clock cycle

4GH$_2$ = 0.25 ns

1ns = $1 \times 10^{-9}$ sec

2) Cache Memory (Fast)

⇒ Access Time : 2.0-5.0 ns

⇒ Stores Frequent Access data and prefetch data from memory

⇒ Store entire pages of memory (Prefetch multiple bytes (512 bits - 1024 bits)) into cache line

⇒ Near to ALU (micrometer)

3) Memory (RAM) (Medium Speed)

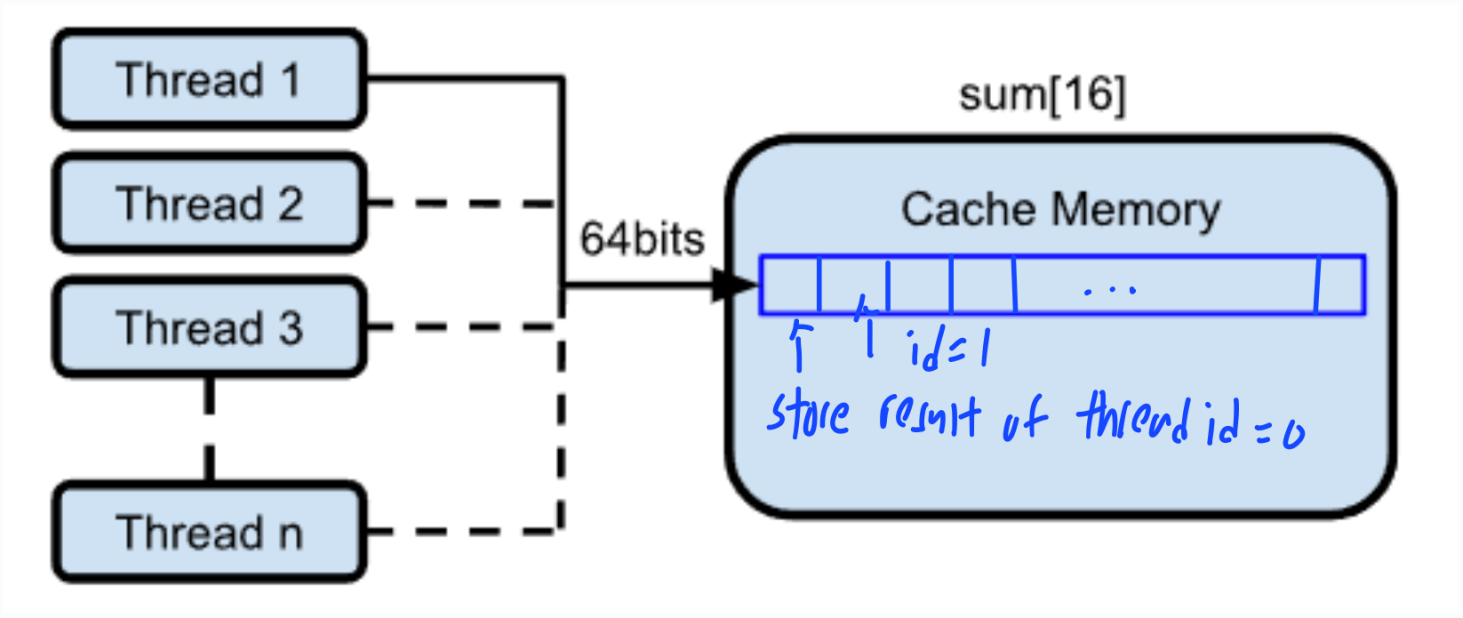
⇒ Access Time: 100-500 ns

⇒ Data Retrieval slower ⇒ electrical signal need to pass through across distance

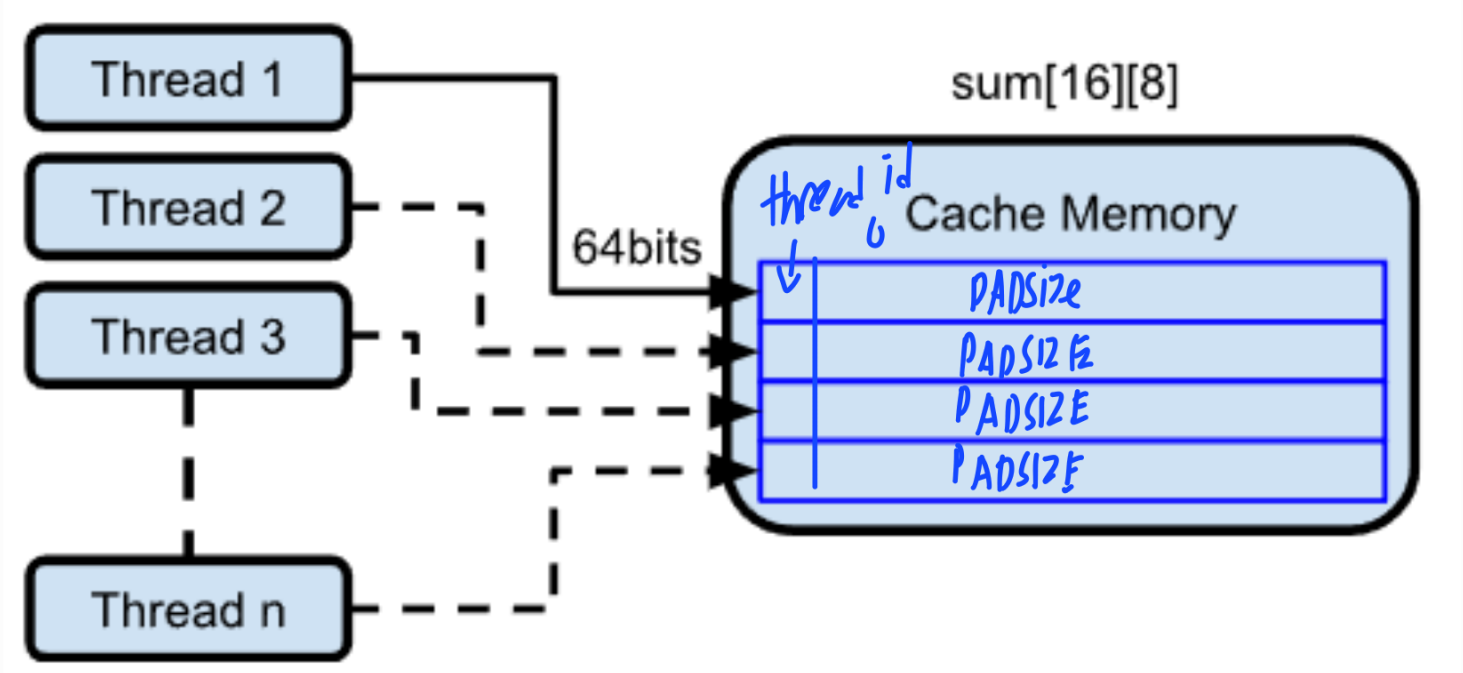
Why use cache memory?

⇒ Access memory take long time (100-500ns)

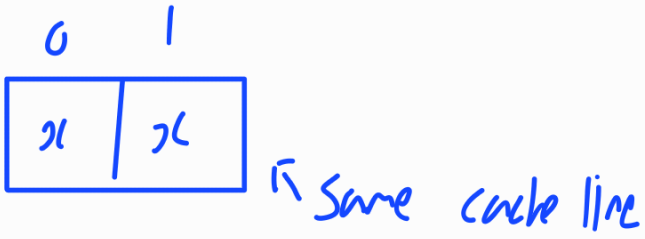
False sharing ⇒ occur when multiple core access diff variable at same cache line. Each time only 1 thread can access the cache line



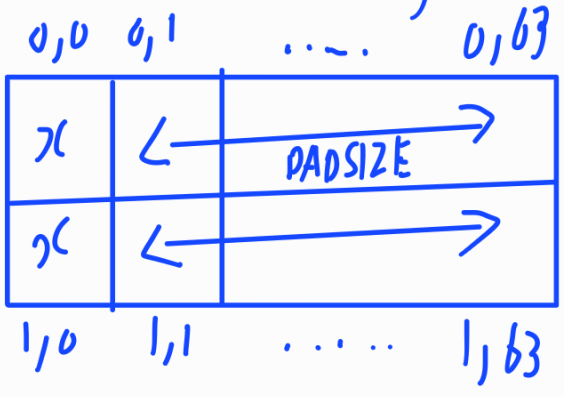

Solution adding PADsize

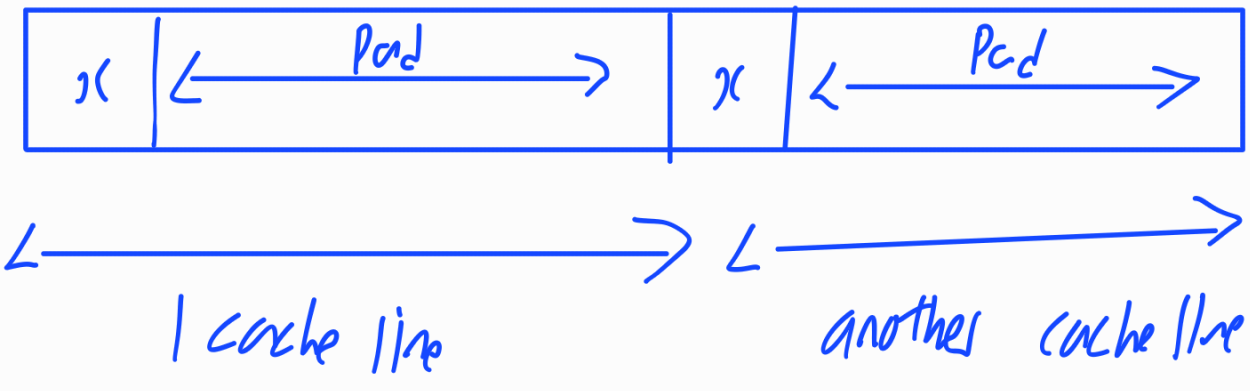


char a[2] → 2 bytes




char a[2][64];



PADS|2E

```
double compute_pi_in_parallel_v2()
{
    int i;
    double x, pi, sum = 0.0;
    double start_time, run_time;
    const int NUM_THREADS = 16;
    const int PAD_SIZES = 64;
    double partial_sum[NUM_THREADS][PAD_SIZES];

    for (int i = 0; i < NUM_THREADS; i++)
    {
        partial_sum[i][0] = 0;
    }

    step = 1.0 / (double)num_steps;

    start_time = omp_get_wtime();

#pragma omp parallel num_threads(NUM_THREADS) private(x, i)      You, 49 minutes ago • Add Practical3 C
    {
        int id = omp_get_thread_num();
        for (i = id + 1; i <= num_steps; i += NUM_THREADS)
        {
            x = (i - 0.5) * step;
            partial_sum[id][0] += 4.0 / (1.0 + x * x);
        }
    }

    for (int i = 0; i < NUM_THREADS; i++)
    {
        sum += partial_sum[i][0];
    }

    pi = step * sum;
    run_time = omp_get_wtime() - start_time;
    printf("Version 2 (Parallel - With Pad Size to Solve False Sharing): pi with %ld steps is %.16lf in
    %lf seconds\n ", num_steps, pi, run_time);

    return run_time;
}
```

Explaination on for keyword

```
#pragma omp parallel for
for (i = 0 ; i < MAX; i ++) {

}
```

turn into this (reduce boilerplate code)

```
# pragma omp parallel num_threads(NUM_THREADS)
{
    int id = omp_get_thread_num();
    for (i = id; i  < MAX; i += NUM_THREADS) {
        int i
    }
}
```

# For + Reduction(+:sum)

```
double compute_pi_in_parallel_v3()
{
    int i;
    double x, pi, sum = 0.0, step;
    double start_time, run_time;
    const int NUM_THREADS = 16;
    const int num_steps = 1000000; // Example number of steps

    step = 1.0 / (double)num_steps;

    start_time = omp_get_wtime();

// #pragma omp parallel for reduction(+:sum) private(x) num_threads(NUM_THREADS) // Set the number of
threads
#pragma omp parallel for reduction(+ : sum) private(x)
    for (i = 1; i <= num_steps; i++)
    {
        x = (i - 0.5) * step;
        sum += 4.0 / (1.0 + x * x); // Sum all thread's results
    }

    // Calculate pi after the parallel region
    pi = step * sum;
    run_time = omp_get_wtime() - start_time;

    printf("Version 3 (Parallel - For Reduction): pi with %d steps is %.16lf in %lf seconds\n ",
    num_steps, pi, run_time);

    return run_time;
}
```

cannot have { , must start with for y

+ => add, x => multiple

# Result

```
int main()
{
    double ori_rt, mod_rt_v1, mod_rt_v2, mod_rt_v3;
    ori_rt = compute_pi();
    mod_rt_v1 = compute_pi_in_parallel_v1();
    mod_rt_v2 = compute_pi_in_parallel_v2();
    mod_rt_v3 = compute_pi_in_parallel_v3();
    std::cout << "The performance gain between non-parallel and parallel is " << ori_rt / mod_rt_v1 <<
    std::endl;
    std::cout << "The performance gain between non-parallel and parallel (With Pad Size to Solve False
    Sharing) is " << ori_rt / mod_rt_v2 << std::endl;
    std::cout << "The performance gain between non-parallel and parallel(For + Reduction) is " <<
    ori_rt / mod_rt_v3 << std::endl;

    return 0;
}
```

```
Non-Parallel: pi with 100000000 steps is 3.1415926535904264 in 0.271470 seconds
 Version 1 (Parallel): pi with 100000000 steps is 3.1415926535897758 in 0.265444 seconds
 Version 2 (Parallel - With Pad Size to Solve False Sharing): pi with 100000000 steps is 3.14159265358
97758 in 0.070436 seconds
 Version 3 (Parallel - For Reduction): pi with 1000000 steps is 3.1415926535898717 in 0.000680 seconds
 The performance gain between non-parallel and parallel is 1.0227
The performance gain between non-parallel and parallel (With Pad Size to Solve False Sharing) is 3.854
15
The performance gain between non-parallel and parallel(For + Reduction) is 399.377
```

Why >TOL= 0.001 , not ==0

```
cval = Pdim * AVAL * BVAL;
errsq = 0.0;
for (i = 0; i < Ndim; i++) {
    for (j = 0; j < Mdim; j++) {
        err = *(C + i * Mdim + j) - cval;
        errsq += err * err; //square it to make the error not balance each other //always have positive error
    }
}

// if not error, the error is 0
if (errsq > TOL)
    printf("\n Errors in multiplication: %f", errsq);
else
    printf("\n Hey, it worked");

printf("\n all done \n");
```

## 1. Binary Representation of Decimal Numbers:

In digital computers, numbers are stored in **binary** (base 2) format, but humans generally use **decimal** (base 10). Not all decimal numbers can be exactly represented in binary with a finite number of bits. This leads to **rounding errors** when we try to store a decimal number in binary format.

**Example: 1.9 in Decimal to Binary Conversion**

Let's try to convert the decimal number `1.9` into binary:

1. **Integer part (1):**
   - The integer part `1` is straightforward. In binary, `1` is simply `1`.
2. **Fractional part (0.9):** To represent the fractional part `0.9`, we multiply it by 2 and take the integer part of the result:

$$0.9 \times 2 = 1.8 \quad \text{(integer part is 1, fractional part is 0.8)}$$

   Now take the fractional part `0.8` and multiply by 2:

$$0.8 \times 2 = 1.6 \quad \text{(integer part is 1, fractional part is 0.6)}$$

   Then, multiply `0.6` by 2:

$$0.6 \times 2 = 1.2 \quad \text{(integer part is 1, fractional part is 0.2)}$$

   Continuing this process, you keep getting:

$$0.2 \times 2 = 0.4, \quad 0.4 \times 2 = 0.8, \quad 0.8 \times 2 = 1.6, \quad \ldots$$

   As you can see, the fractional part keeps cycling through the same values (`0.9, 0.8, 0.6, 0.2, 0.4, 0.8, ...`), which means the binary representation of `0.9` is **repeating**.

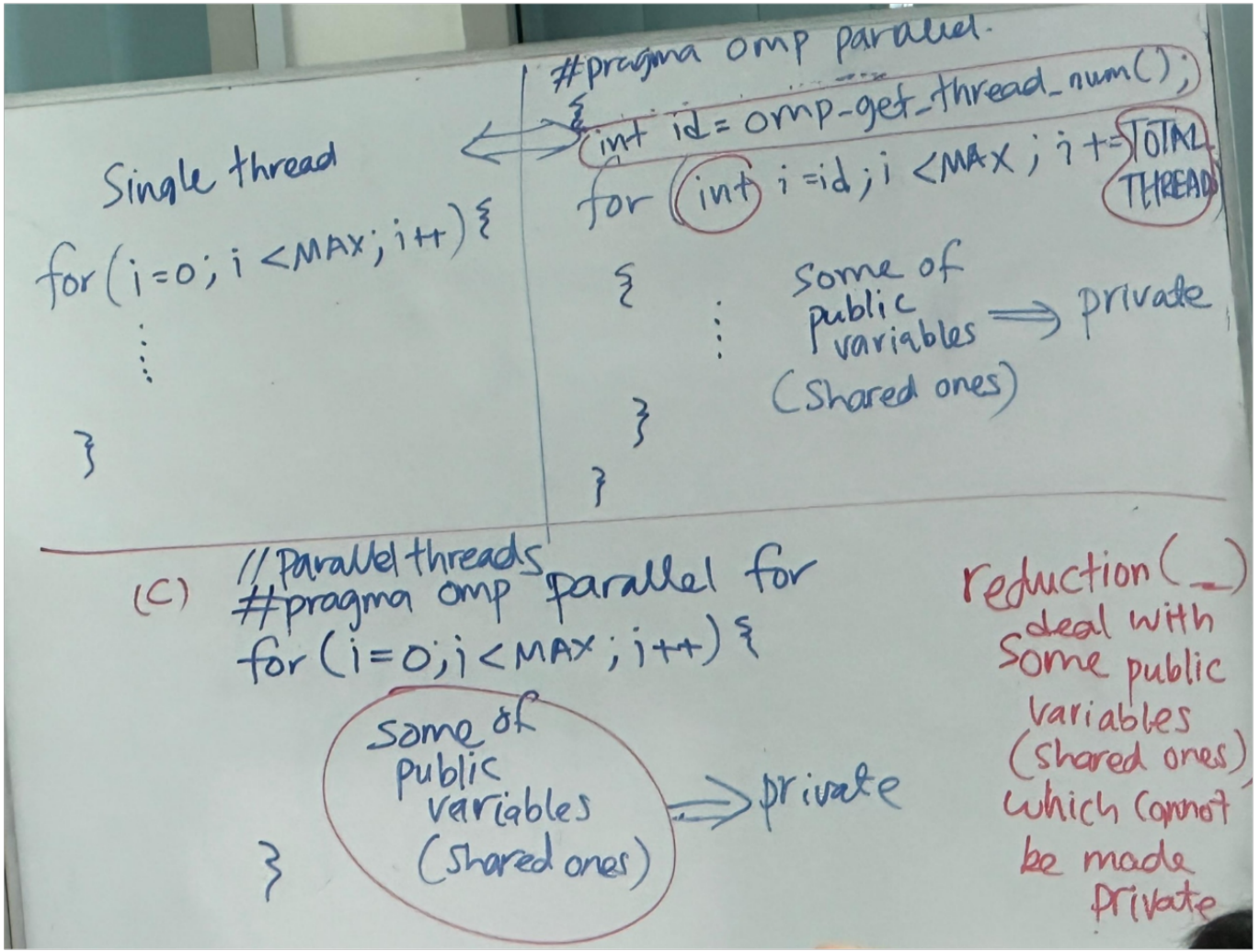
   The binary representation of `0.9` is approximately:

$$0.9 \approx 0.1110011001100110011\ldots_2 \quad \text{(repeating)}$$

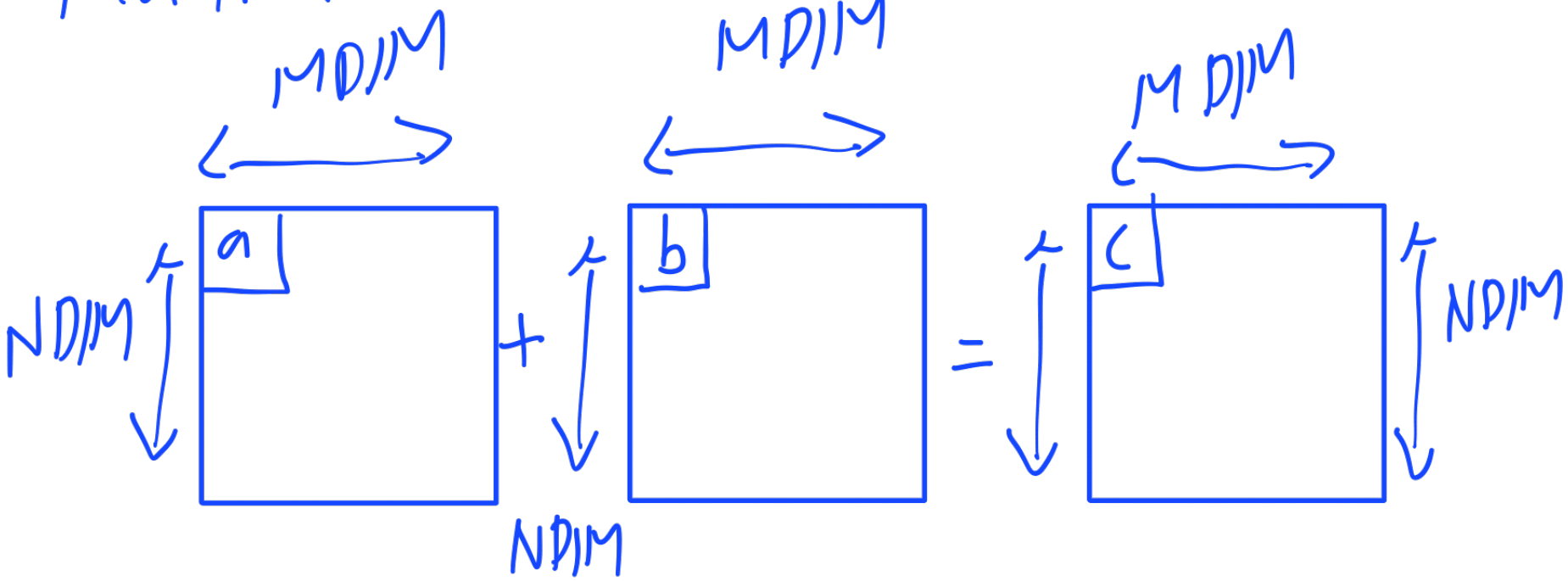
3. **Putting it together:** The full binary representation of `1.9` would be:

$$1.9 \approx 1.1110011001100110011\ldots_2$$

   Since the fractional part is repeating, it cannot be exactly represented with a finite number of bits.


Single thread
for (i=0; i<MAX; i++) {
}
#pragma omp parallel
{
int id = omp_get_thread_num();
for (int i=id; i<MAX; i+=TOTAL_THREADS)
{
Some of public variables ⟹ private (Shared ones)
}
}
(C) // Parallel threads
#pragma omp parallel for
for (i=0; i<MAX; i++) {
some of public variables (Shared ones) ⟹ private
}
reduction(_) deal with some public variables (shared ones) which cannot be made private

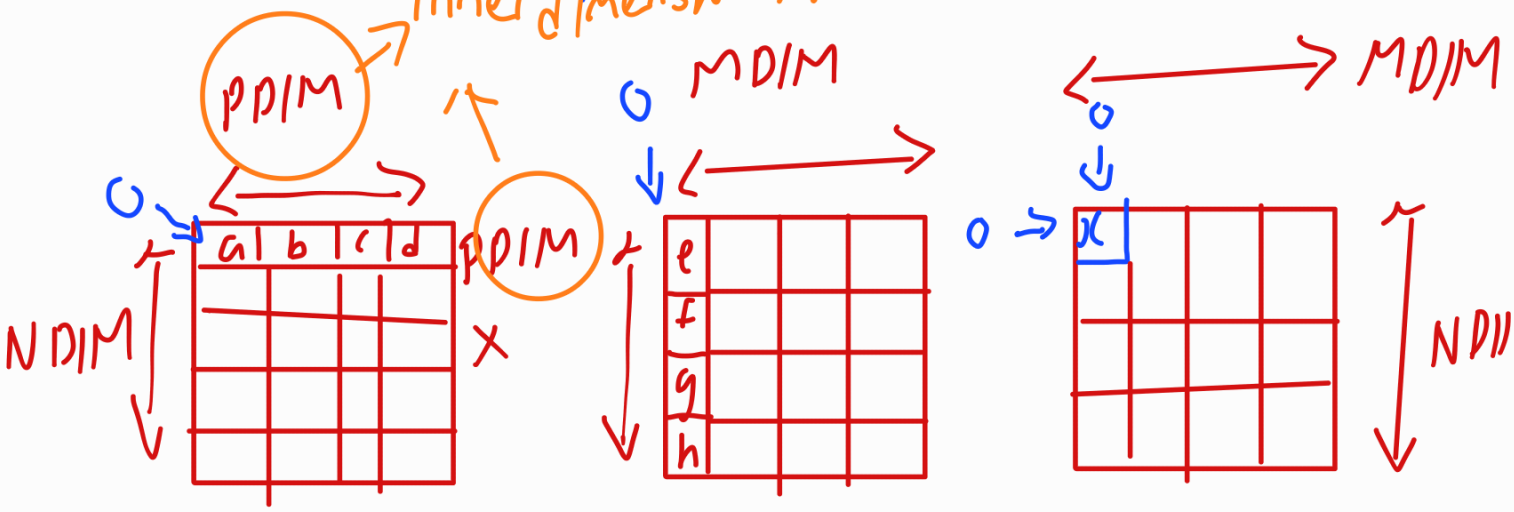
# Matrix Addition




$a + b = c$

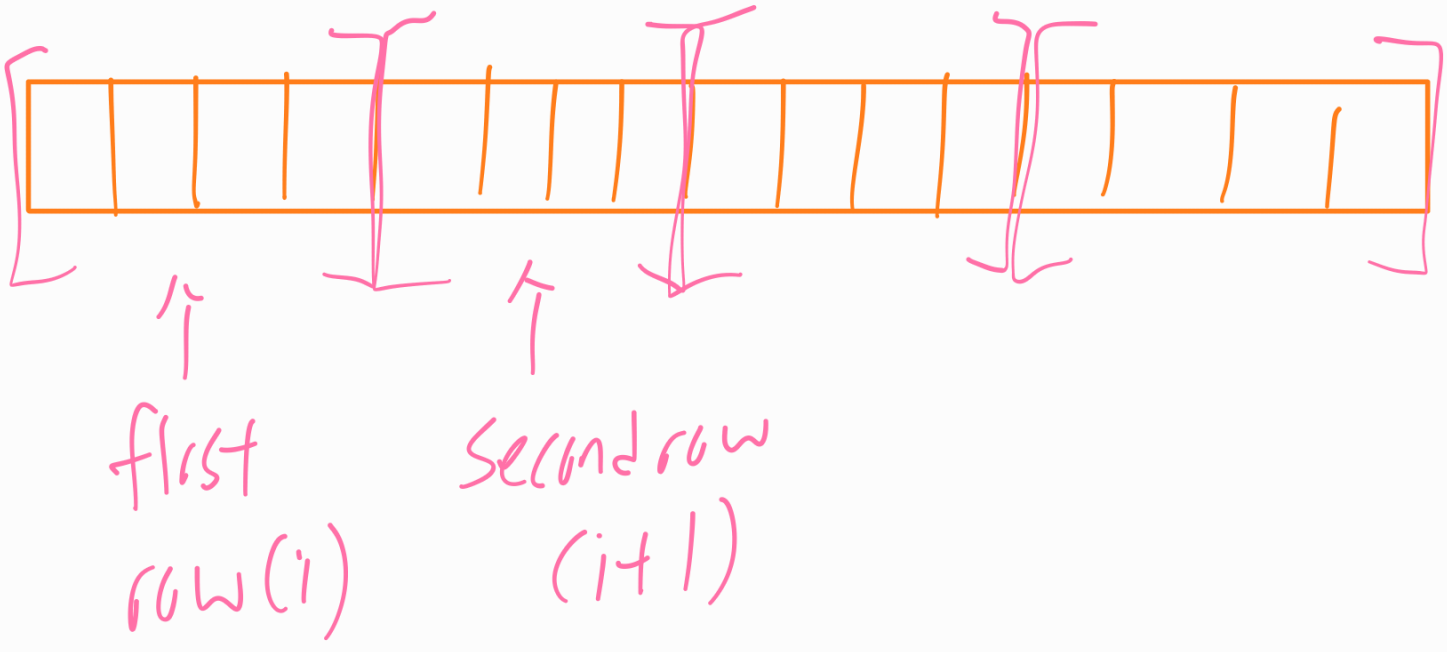
# Matrix Multiplication

inner dimension must match




$(a*e)+(b*f)+(c*g)+(d*h)=x$

matrix (imagination) to linear memory



```c
#define ORDER 1000
#define AVAL 3.0
#define BVAL 5.0
#define TOL  0.001 //tolerance

int main(int argc, char** argv) {
    int Ndim, Pdim, Mdim;   /* A[N][P], B[P][M], C[N][M] */
    int i, j, k;
    double* A, * B, * C, cval, tmp, err, errsq;
    double dN, mflops;
    double start_time, run_time;

    Ndim = ORDER;
    Pdim = ORDER;
    Mdim = ORDER;

    A = (double*)malloc(Ndim * Pdim * sizeof(double));
    B = (double*)malloc(Pdim * Mdim * sizeof(double));
    C = (double*)malloc(Ndim * Mdim * sizeof(double));
```

allocate num of bytes

dynamic memory

→ the memory only appear when this line code run and memory can be free

where static memory/ variable appear before program start

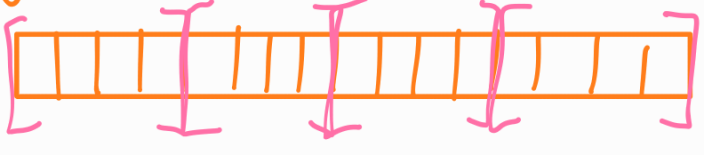
return address that need a pointer to points to the first byte allocated

8 bytes (in modern pc)

↓

but 4bytes in old 8086 PC

pointer



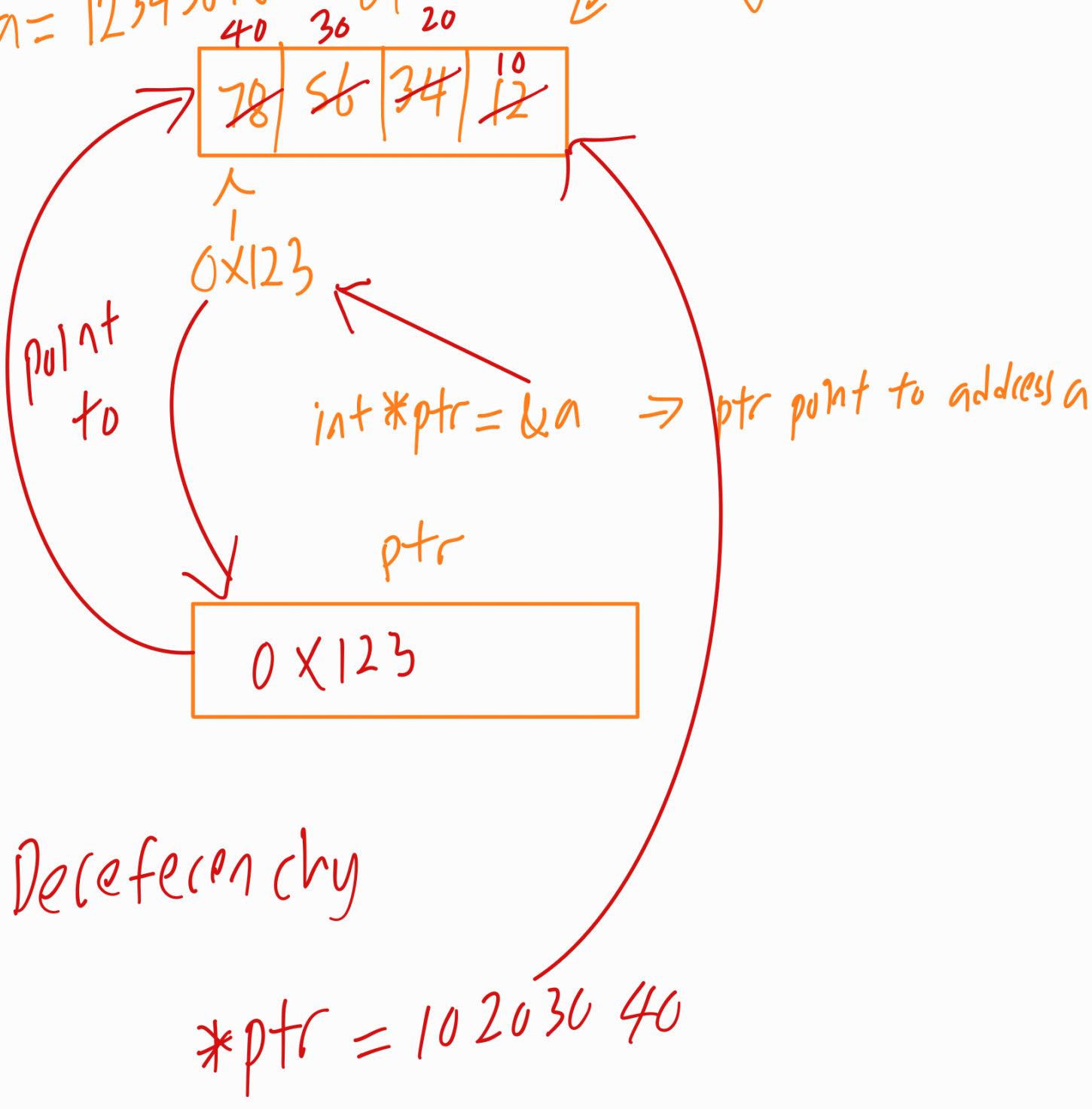
# Detail about pointer

int a;

a = 12345678

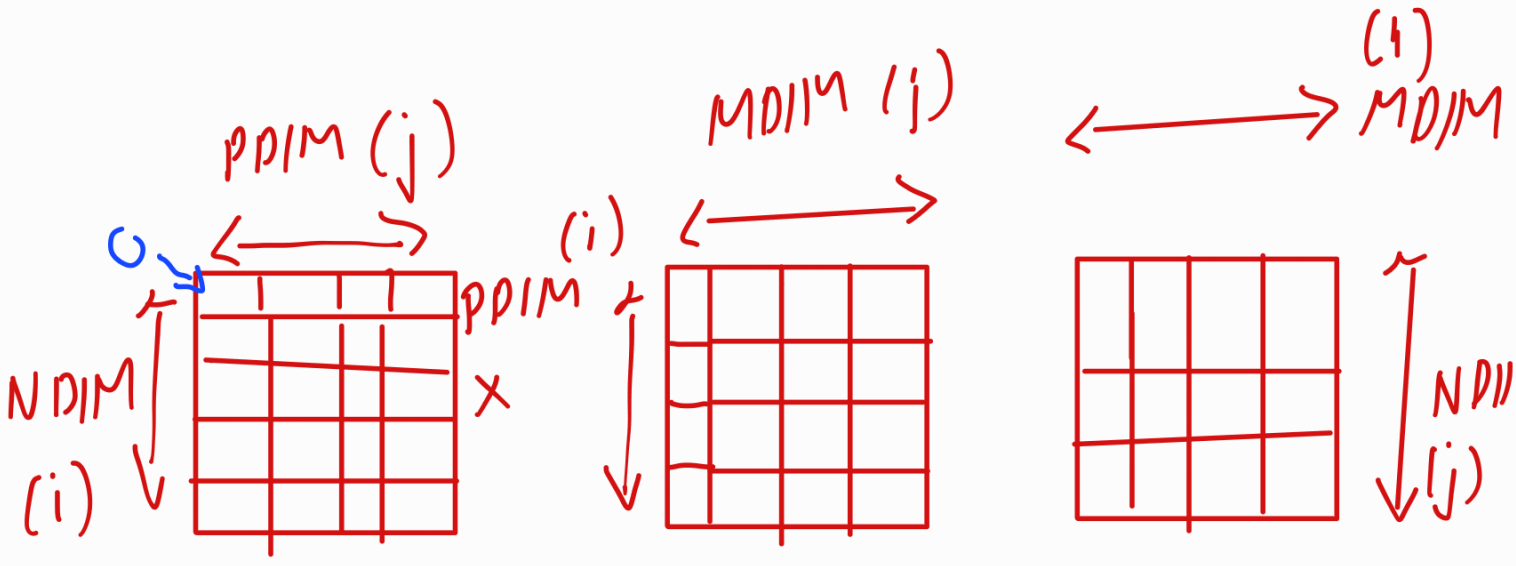
little indian

4 bytes

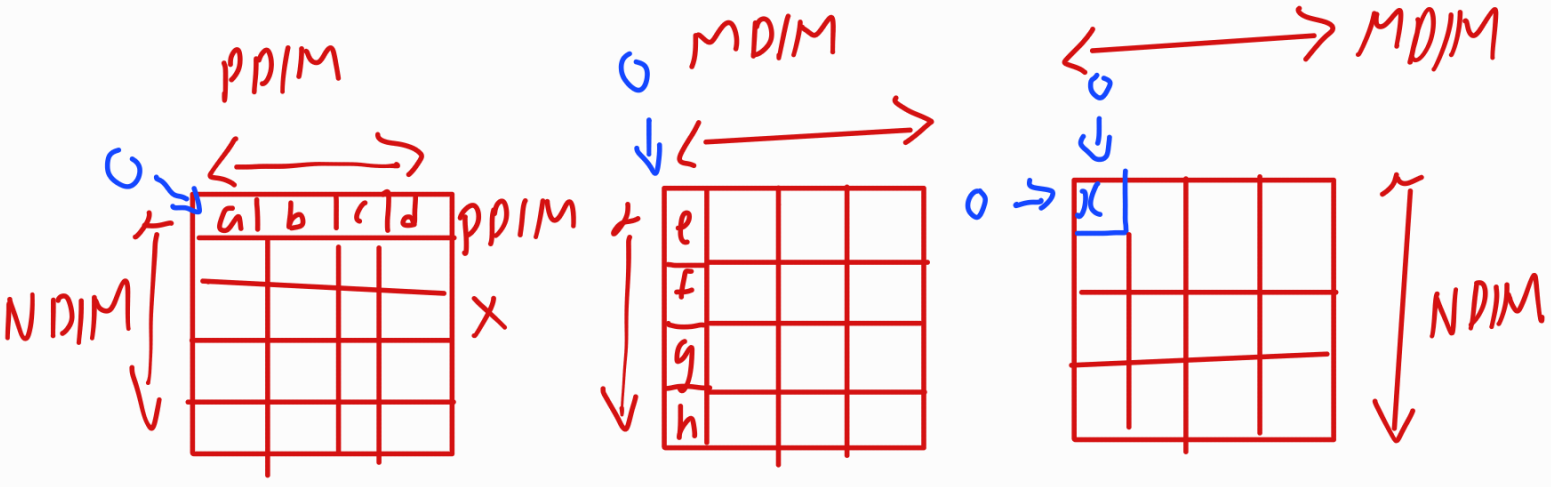



Decefercnchy

*ptr = 10 20 30 40




```
/* Initialize matrices */

for (i = 0; i < Ndim; i++)
    for (j = 0; j < Pdim; j++)
        *(A + (i * Pdim + j)) = AVAL;
    or //A[i * Pdim + j] = AVAL;

for (i = 0; i < Pdim; i++)
    for (j = 0; j < Mdim; j++)
        *(B + (i * Mdim + j)) = BVAL;
    ok for efficiency

for (i = 0; i < Ndim; i++)
    for (j = 0; j < Mdim; j++)
        *(C + (i * Mdim + j)) = 0.0;
```




(a*e) + (b*f) + (c*g) + (d*h) = x

```c
/* Do the matrix product */

start_time = omp_get_wtime();
for (i = 0; i < Ndim; i++) {
    for (j = 0; j < Mdim; j++) {
        tmp = 0.0;
        for (k = 0; k < Pdim; k++) {
            /* C(i,j) = sum(over k) A(i,k) * B(k,j) */
            tmp += *(A + (i * Pdim + k)) * *(B + (k * Mdim + j));
        }
        *(C + (i * Mdim + j)) = tmp;
    }
}
```

for row

for col

row * length of column + col

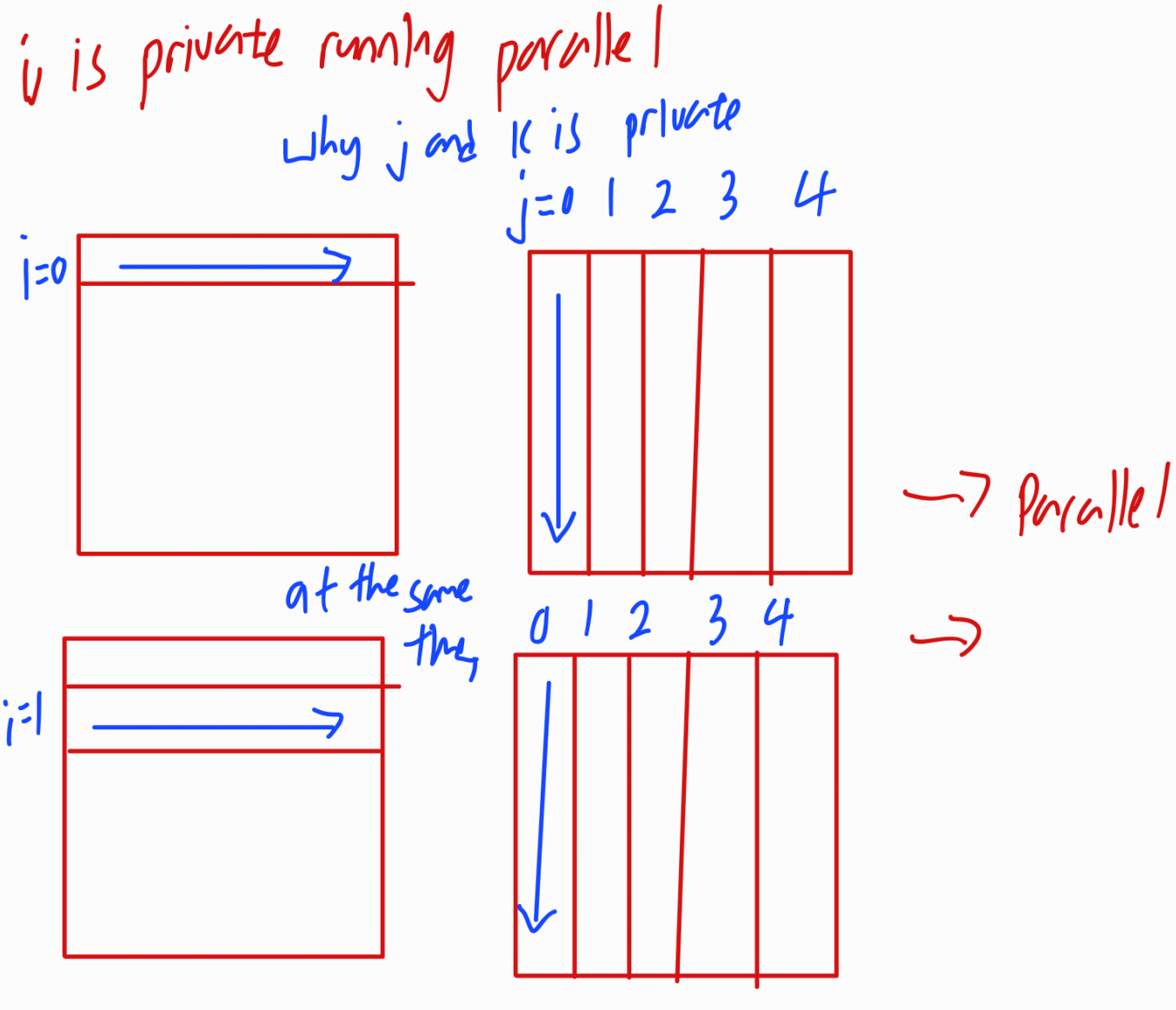
# OS_Compute_PI_dynamically_change_thread

```
    // Loop through different numbers of threads
    for (int num_threads = 1; num_threads <= NUM_THREADS; num_threads++)
    {
        sum = 0.0;                          // Reset sum for each number of threads
        omp_set_num_threads(num_threads); // Set the number of threads dynamically

// #pragma omp parallel for reduction(+:sum) private(x) num_threads(NUM_THREADS) // Set the number of threads
#pragma omp parallel for reduction(+ : sum) private(x)
        for (i = 1; i <= num_steps; i++)
        {
            x = (i - 0.5) * step;
            sum += 4.0 / (1.0 + x * x); // Sum all thread's results
        }

        // Calculate pi after the parallel region
        pi = step * sum;
        run_time = omp_get_wtime() - start_time;

        // Use a parallel block to retrieve the number of threads
        int actual_num_threads;
#pragma omp parallel
        {
            actual_num_threads = omp_get_num_threads();
        }

        printf("num_threads = %d\n", actual_num_threads);
        printf("Pi is %.16lf in %lf seconds and %d threads\n ", pi, run_time, actual_num_threads);
    }
```

```
num_threads = 1
Pi is 3.1415926535897643 in 0.002653 seconds and 1 threads
 num_threads = 2
Pi is 3.1415926535898993 in 0.004262 seconds and 2 threads
 num_threads = 3
Pi is 3.1415926535899041 in 0.005520 seconds and 3 threads
 num_threads = 4
Pi is 3.1415926535898753 in 0.006492 seconds and 4 threads
 num_threads = 5
Pi is 3.1415926535899117 in 0.008473 seconds and 5 threads
 num_threads = 6
Pi is 3.1415926535898828 in 0.010158 seconds and 6 threads
 num_threads = 7
Pi is 3.1415926535898944 in 0.014701 seconds and 7 threads
 num_threads = 8
Pi is 3.1415926535898713 in 0.016928 seconds and 8 threads
```

# multiply_matrices_in_parallel

```
// i is private , because the pragma omp for already make it private
// all other variables is public
// Mdim, PDim, Ndim -> public (because we never write into these variables - only reading, no race condition)
// j -> private (all the threads handle it own i row, so each i row need to multiple each j column
// tmp -> private
// k -> private
// A, B -> public (never change the content in A and B)
// C -> public
// #pragma omp parallel for private(j, k, tmp) shared(A, B, C, Ndim, Pdim, Mdim)
#pragma omp parallel private(j, k, tmp) num_threads(NUM_THREADS)
    {
#pragma omp for
        for (i = 0; i < Ndim; i++)
        {
            for (j = 0; j < Mdim; j++)
            {
                tmp = 0.0;
                for (k = 0; k < Pdim; k++)
                {
                    /* C(i,j) = sum(over k) A(i,k) * B(k,j) */
                    tmp += *(A + (i * Pdim + k)) * *(B + (k * Mdim + j));
                }
                *(C + (i * Mdim + j)) = tmp;
            }
        }
    }
```



if j is public, if i is 1, and it calculate faster, and it update j=1, but the i is 0 case haven't complete the calculation in j=0 row, but since j have been updated by i=1 row, i=0 now doby j=1, which is not good

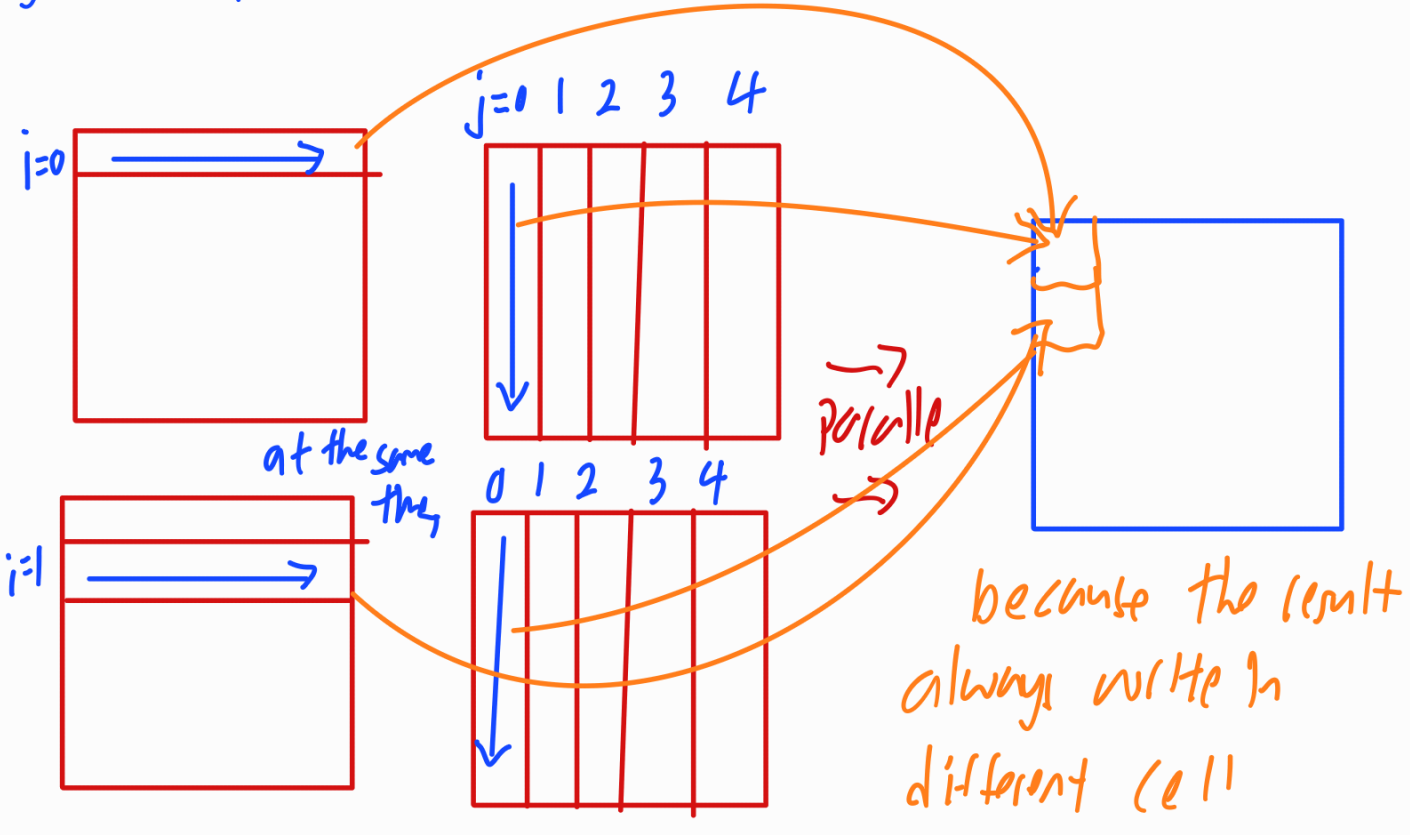
---

Why A, B, NDIM, PDIM is public

→ because we only readby, but not writhby the value

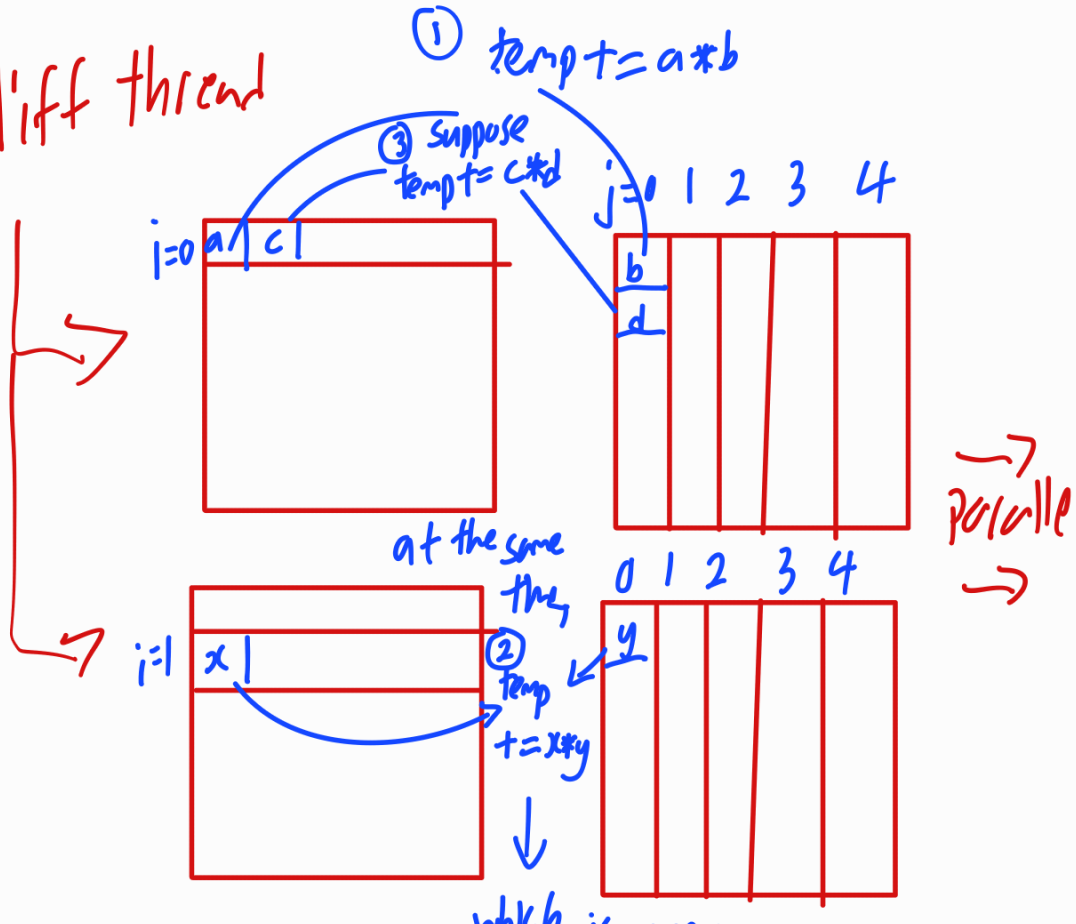
---

Why C is public



# why temp is private

diff thread



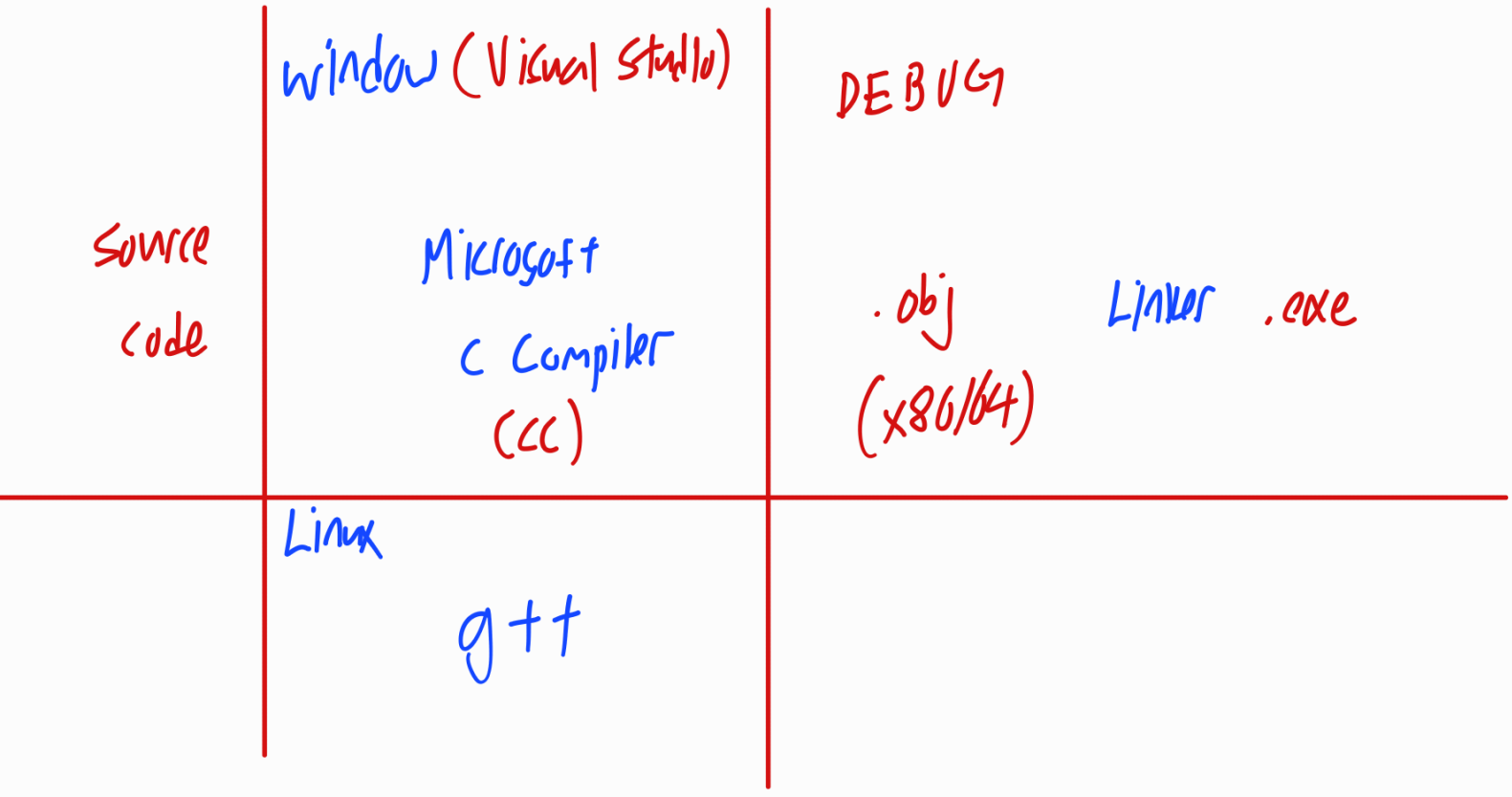

which is wrong

because we want i=0 row * j=0 row

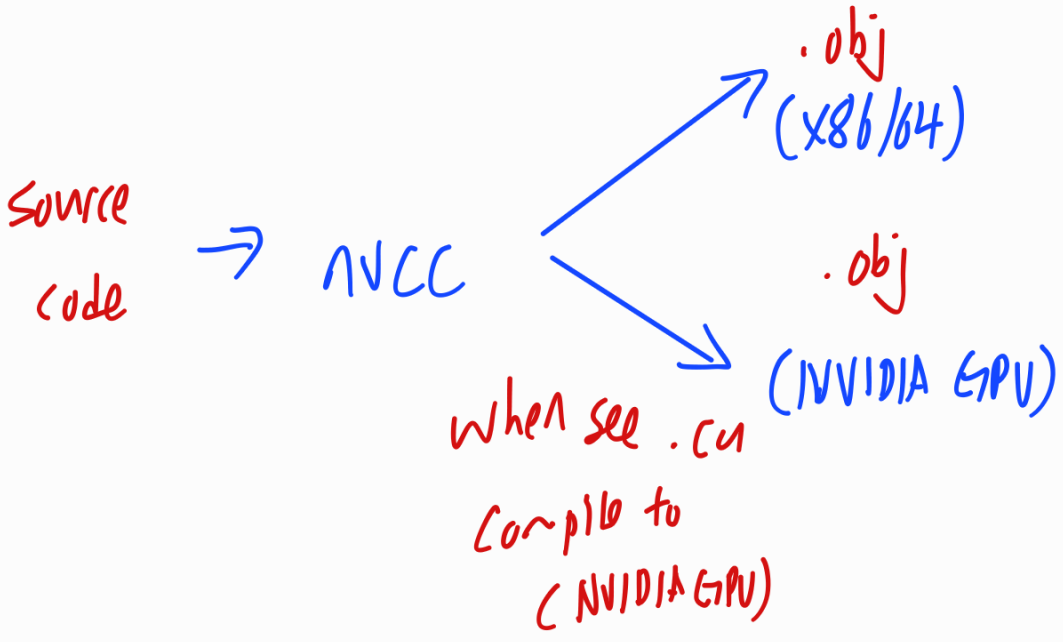
but now we get some value from

i=0 row * j=0 row +

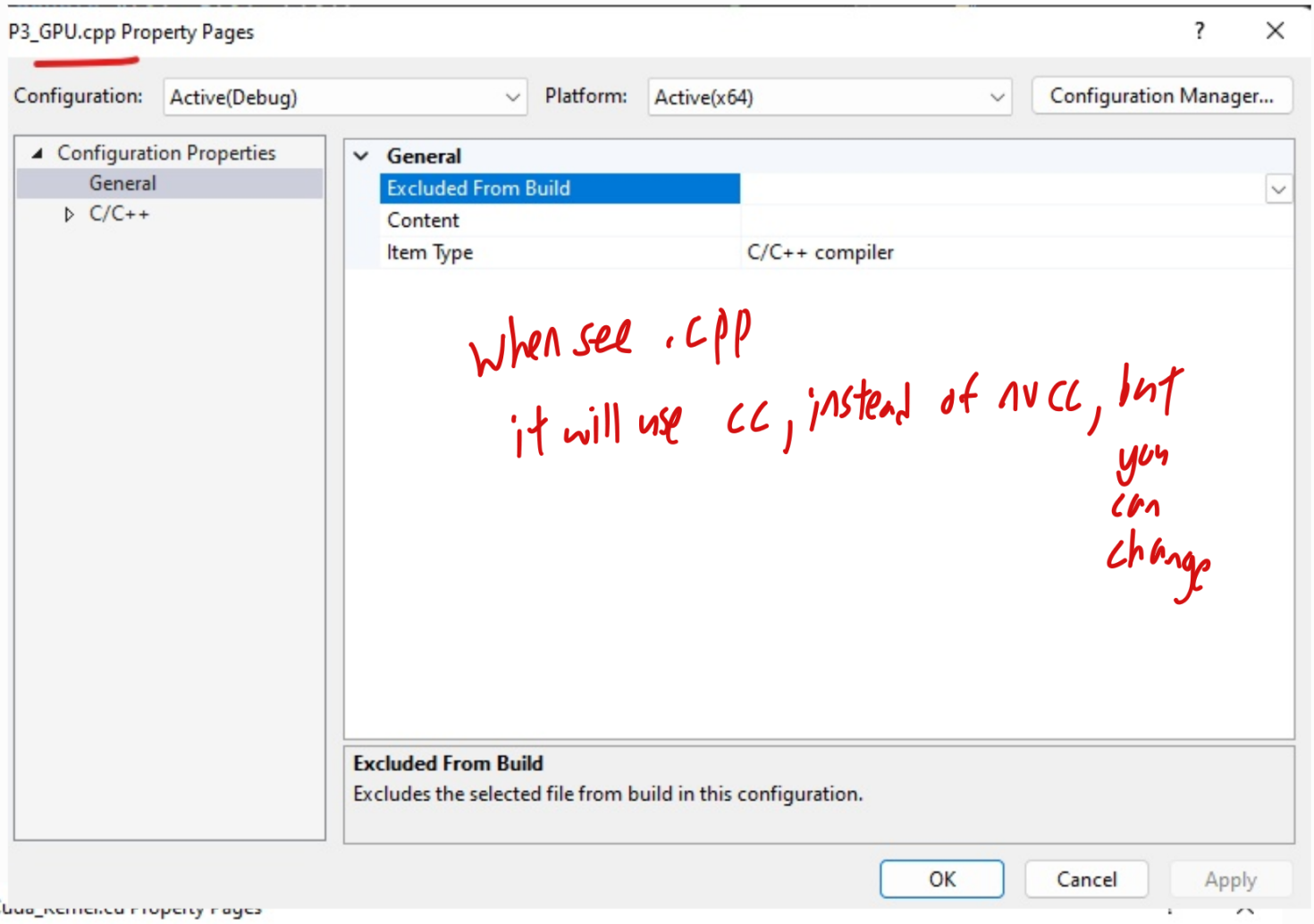
i=1 row * j=0 row.

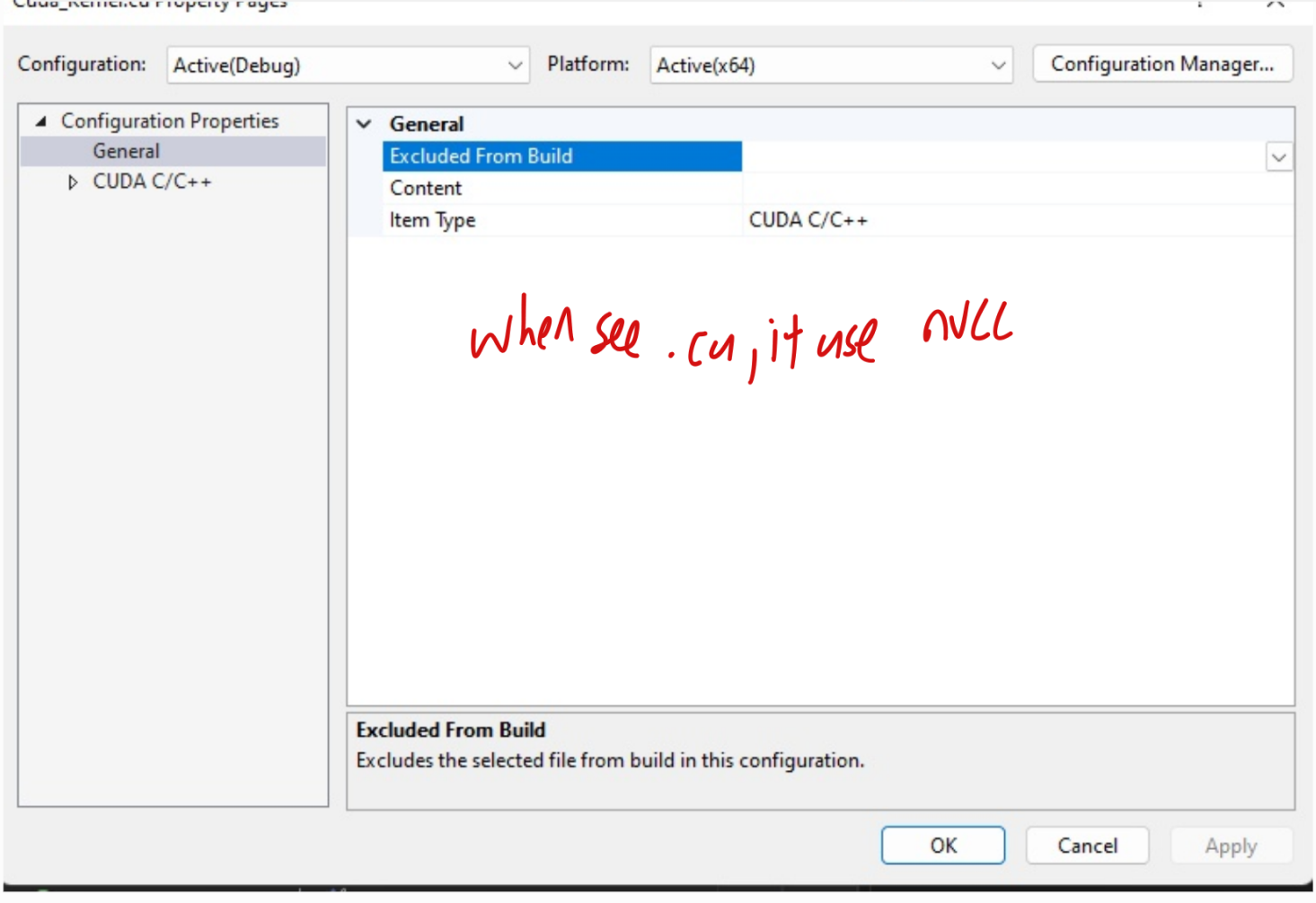

window (Visual Studio)
DEBUG
Source code
Microsoft C Compiler (cc)
.obj (x86/64)
Linker
.exe
Linux
gtt

Nvidia


Source code
NVCC
.obj (x86/64)
.obj (NVIDIA GPU)
when see .cu
Compile to (NVIDIA GPU)

In visual studio



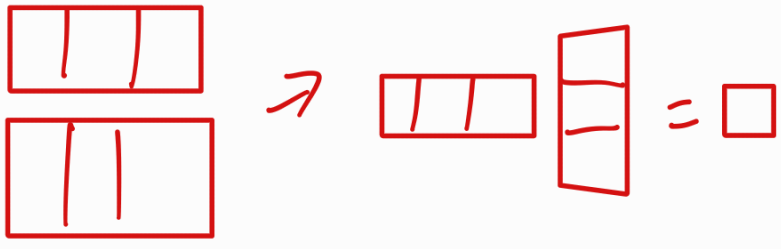



<<< >>> - only understand by nvcc, not cc/gtt, if using cl or gtt compile .cu code, it will have error

vector 1 dimension

Matrix 2 dimension

$ab^T$ = scalar value



```cpp
// __global__ tell nvcc that this code is kernel not a function, that code translate to nvidia gpu machine code to run
__global__ void vectorAdditionKernel(double *A, double *B, double *C, int arraySize)
{
    // Get thread ID.
    int threadID = blockDim.x * blockIdx.x + threadIdx.x;

    // Check if thread is within array bounds.
    if (threadID < arraySize)
    {
        // Add a and b.
        C[threadID] = A[threadID] + B[threadID];
    }
}
```

← Kernel (run in nvidia gpu)

```cpp
// for pc to run
void kernel(double *A, double *B, double *C, int arraySize)
{

    // Initialize device pointers.
    double *d_A, *d_B, *d_C;

    // Allocate device memory.
    cudaMalloc((void **)&d_A, arraySize * sizeof(double));
    cudaMalloc((void **)&d_B, arraySize * sizeof(double));
    cudaMalloc((void **)&d_C, arraySize * sizeof(double));

    // Transfer arrays a and b to device.
    cudaMemcpy(d_A, A, arraySize * sizeof(double), cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, B, arraySize * sizeof(double), cudaMemcpyHostToDevice);

    // Calculate blocksize and gridsize.
    dim3 blockSize(512, 1, 1);   // 512 thread ,x,y, z because gpu is design for 3d gaming
    dim3 gridSize(512 / arraySize + 1, 1); // grid two dimension
      You, 1 hour ago • Update
    // Launch CUDA kernel.
    vectorAdditionKernel<<<gridSize, blockSize>>>(d_A, d_B, d_C, arraySize);

    // Copy result array c back to host memory.
    cudaMemcpy(C, d_C, arraySize * sizeof(double), cudaMemcpyDeviceToHost);
}
```

(1) you write a function called cuda kernel that run on GPU

(2) copy data from main Ram over GPU memory

(3) CPU tell GPU, that kernel function in parallel to execute

host { CPU → can see all memory

Memory

A 5 8 3
B 7 6 4
C 12 14 7

return back to CPU mem

Transfer array from host to device

because other process also need to use gpu memory, so we put back to cpu memory, free out gpu memory for others

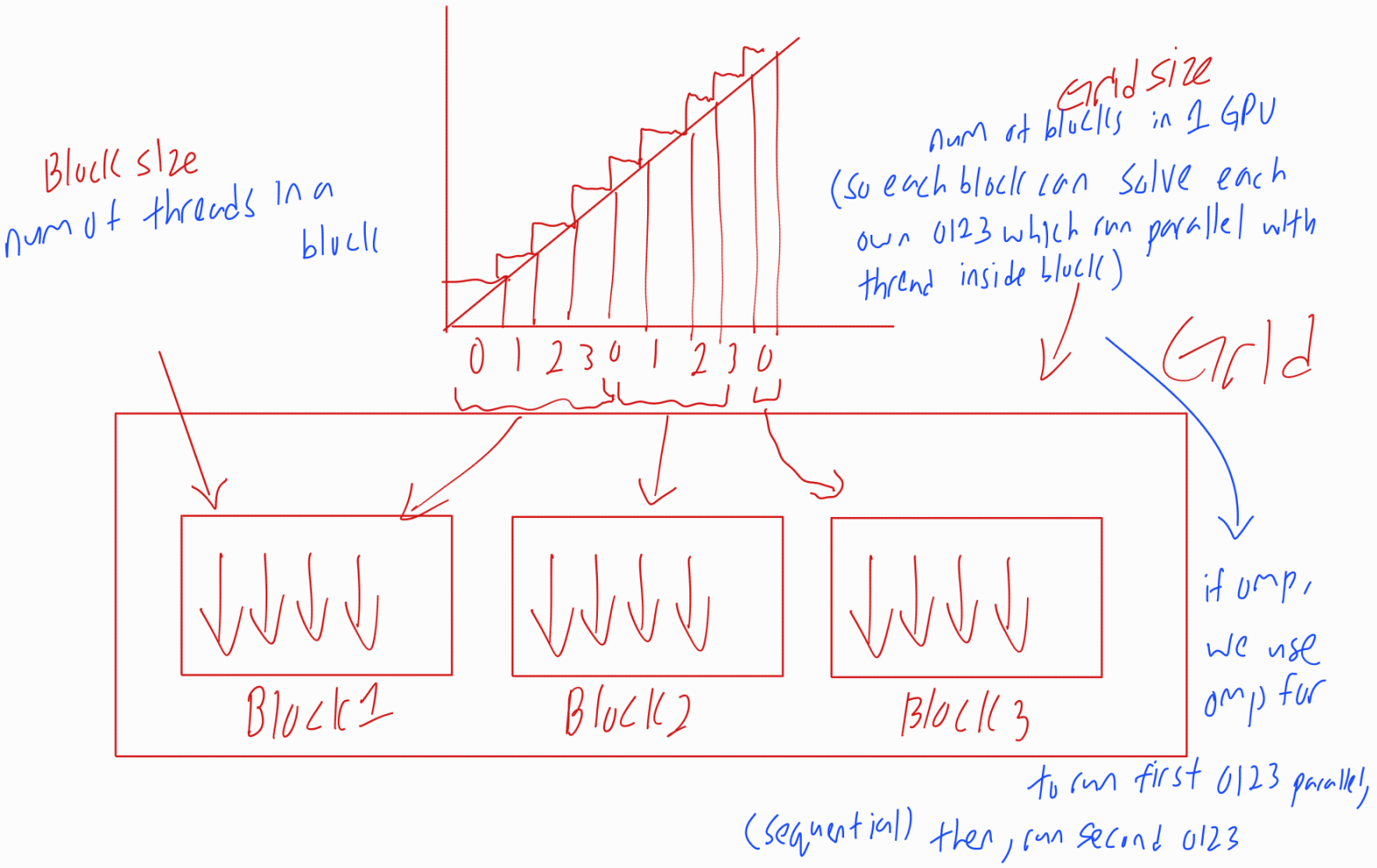
↑ Why need to return back to CPU

device { GPU → can only see their own memory

d_A 5 8 3
d_B 7 6 4
d_C 12 14 7

cudaMalloc if do not specify you want data, OS will reject you to steal memory

sound card → can only see their own memory




### 1. Threads, Blocks, and Grids

- **Thread:** The smallest unit of execution on a GPU. Each thread executes the kernel code.
- **Block:** A group of threads that execute together. Threads in a block can communicate and share memory using **shared memory**.
- **Grid:** A collection of blocks that are launched to execute the kernel.

## Visual Analogy

- Imagine you're assembling a car.
  - Each **thread** is like a worker.
  - A **block** is like a team of workers (limited to a specific number, e.g., 512).
  - The **grid** is the factory floor, containing multiple teams (blocks).
- Each worker (thread) is responsible for assembling one car part (an element in the array). The factory (GPU) runs all teams in parallel to finish the job faster.

## 2. Block Size

The block size determines **how many threads are in each block**. You define it as a `dim3` object, with dimensions in up to three dimensions (x, y, z):

```cpp
dim3 blockSize(512, 1, 1);  // 512 threads in one block
```

- In this example, all 512 threads are aligned along the x-dimension.
- GPU hardware has a **maximum number of threads per block** (often 1024 for many GPUs). You must stay within this limit.

## 3. Grid Size

The grid size determines **how many blocks are launched**. It is also defined as a `dim3` object, with up to three dimensions:

```cpp
dim3 gridSize((arraySize + blockSize.x - 1) / blockSize.x, 1);
```

- The calculation ensures that the grid size is large enough to handle all elements in the array.
- For example, if your array has 1000 elements and the block size is 512 threads:

$$\text{gridSize.x} = \lceil \frac{\text{arraySize}}{\text{blockSize.x}} \rceil = \lceil \frac{1000}{512} \rceil = 2$$

  So, you'll need 2 blocks, each containing up to 512 threads.

## 4. Mapping Work to Threads

Each thread has a unique **thread ID** that is computed based on its position in the grid and block:

```cpp
int threadID = blockIdx.x * blockDim.x + threadIdx.x;
```

- `blockIdx.x`: Index of the block in the grid.
- `blockDim.x`: Number of threads per block.
- `threadIdx.x`: Index of the thread within the block.

This formula ensures each thread gets a unique `threadID` across the entire grid.

## 5. Parallel Execution

- Each thread works on a small part of the problem. For example, in your kernel, thread `i` computes:

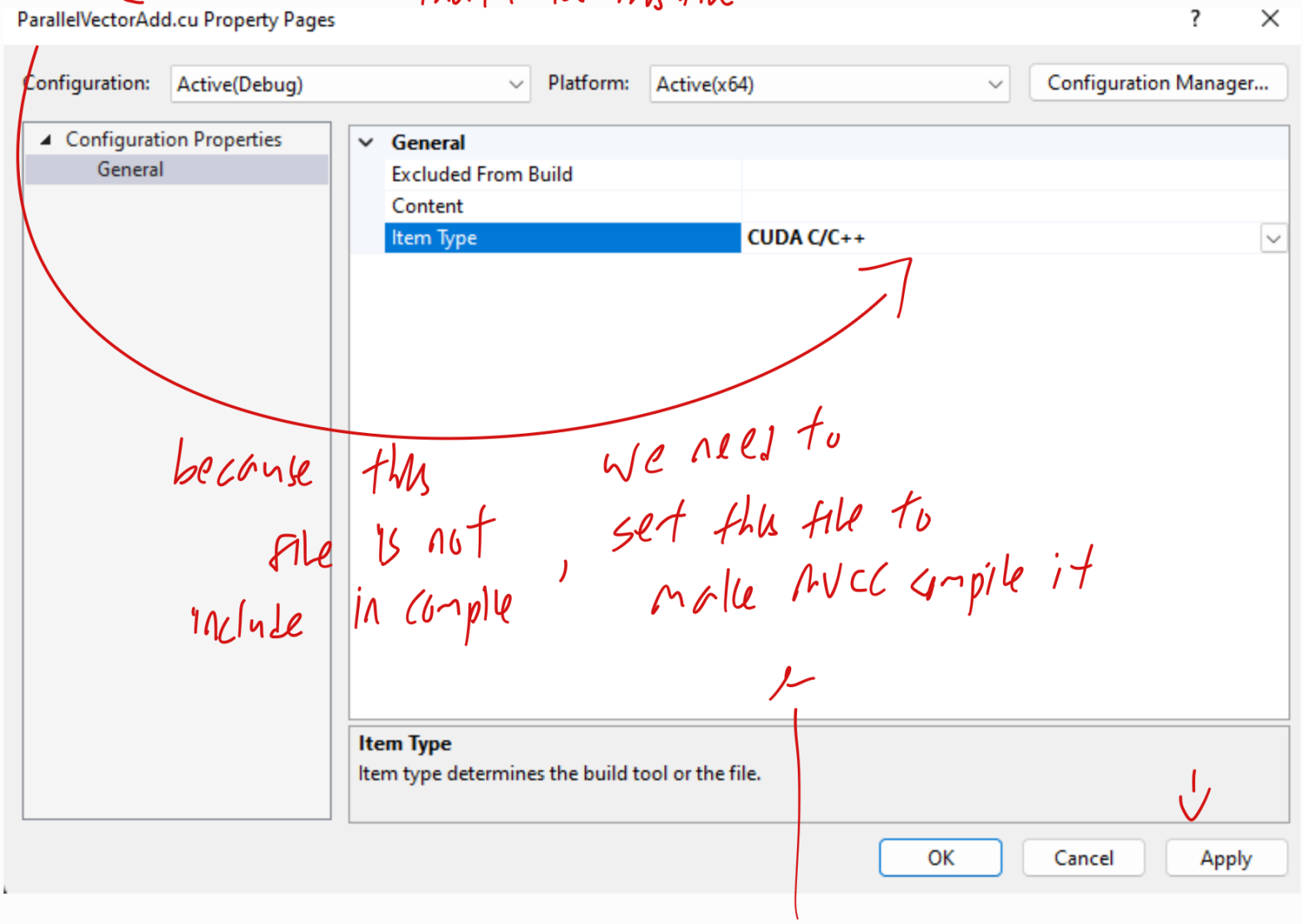
  ```cpp
  C[i] = A[i] + B[i];
  ```

- CUDA handles launching all threads in parallel, dividing the work between them.

If you encounter this error ↓

```
LNK2019 unresolved external symbol "int __cdecl parallel_vec_add(int *,int *,int *,int)" (?parallel_vec_add@@YAHPEAH00H@Z) referenced in function main
LNK1120 1 unresolved externals
```

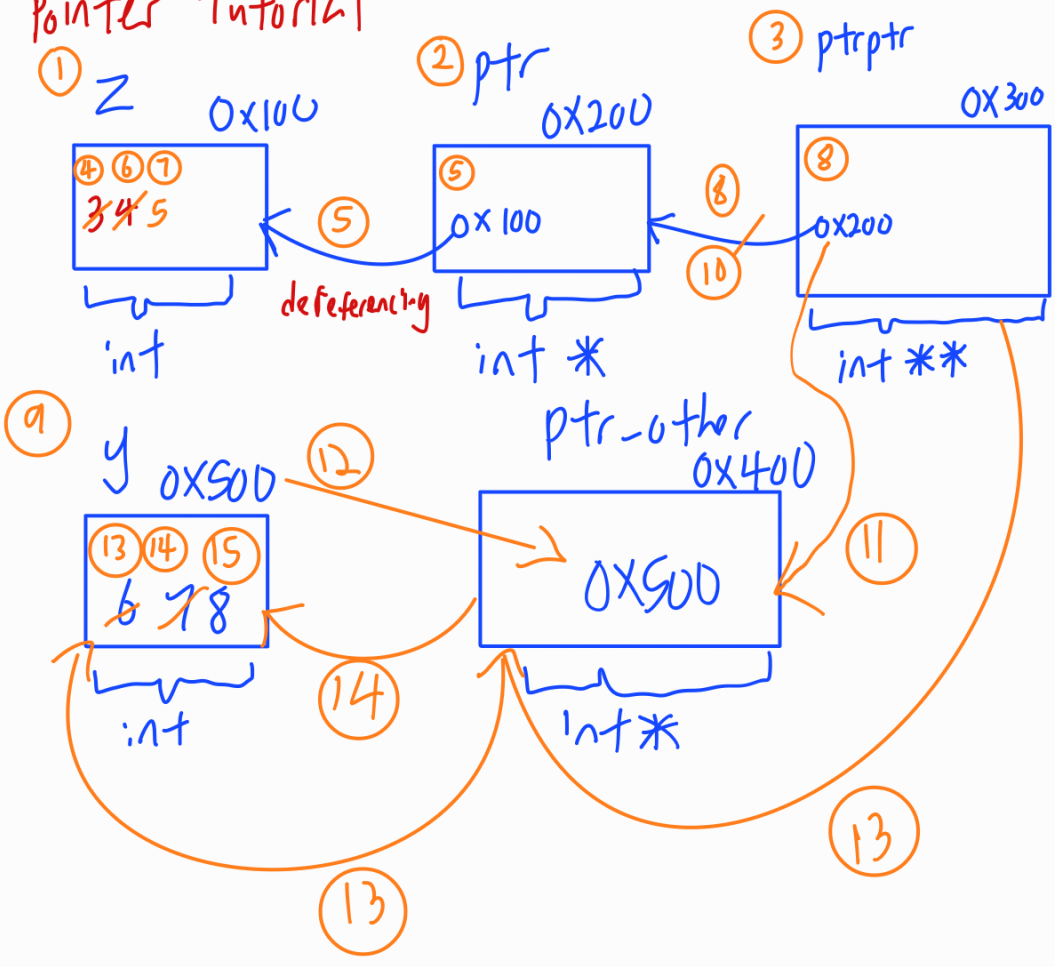
It is because the NVCC compiler does not recognize the function parallel_vec_add that inside this file




because this file is not include in compile, we need to set this file to make NVCC compile it

the reason why this case happen is because in visual studio, we create the cu file first, before we build dependencies > build customization > cuda, which make this file is not included in cuda C/C++ to compile

# Pointer Tutorial




① int z;

② int *ptrj;

Pointer that can hold int type

③ int **ptrptr

Pointer that can hold int pointer type

④ z = 3;

⑤ ptr = &z

addr

⑥ z = 4;

⑦ dereferencing → *ptr = 5;

⑧ ptrptr = &ptrj

⑨ int y;

⑩ ptrptr = &ptrj

move the pointer from point to ptr to ptr_other →

⑪ ptrptr = &ptr_other

⑫ *ptrptr = &y;

⑬ **ptrptr = 6

double dereferencing

⑭ *ptr_other = 7

⑮ y = 8

```c
void test_v1(int* a_ptr){
    // create 4 bytes of SIZE=1024 int block and return the first adddress
    a_ptr = (int*)malloc(SIZE * sizeof(int));
}

void test_v2(int** a_ptr_ptr) {
    *a_ptr_ptr = (int*)malloc(SIZE * sizeof(int));
}

int main() {
    int* a, *b, *c;

    test_v1(a);
    test_v2(&a);
```

Why use double referencing

main

① a

0x100

point to

? = uninitialized value

int *

test_v1

② a_ptr

pass

0x200

? ⑤ 0x300

int *

⑦ after return to main

this value will discard

④

⑤

0x300

SIZE = 1024

test_v2

a_ptr_ptr

① 0x100

int **

④ this var destroy when back to main, but still a is point to the memory block

③ point to

② 0x300

SIZE = 1024

Why use reference pointer ? *&

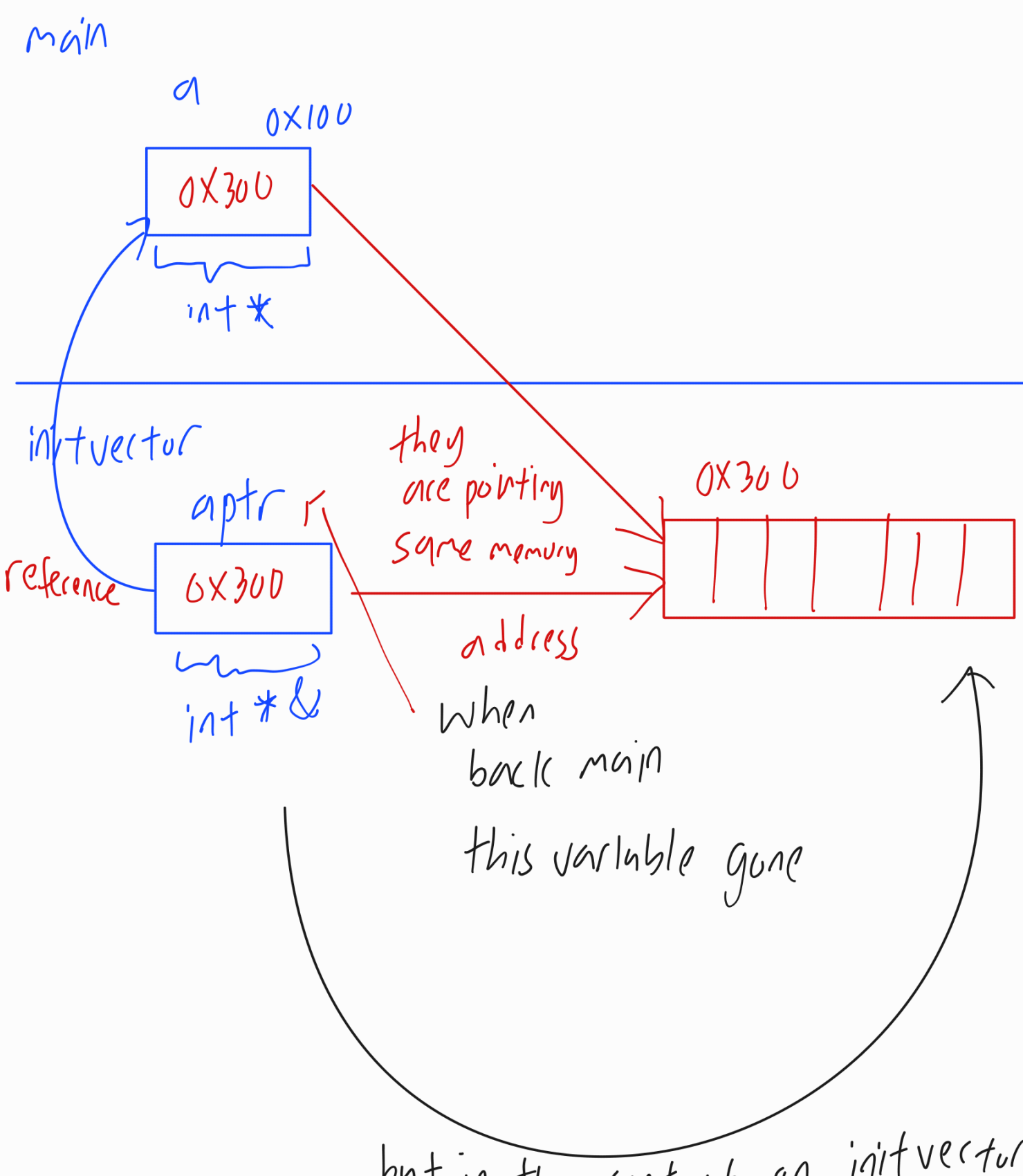
```
void init_vectors(double*& a, double*& b, double*& c) {
    a = (double*)malloc(SIZE * sizeof(double));
    b = (double*)malloc(SIZE * sizeof(double));
    c = (double*)malloc(SIZE * sizeof(double));

    for (int i = 0; i < SIZE; ++i)
    {
        a[i] = (double)i;
        b[i] = (double)i;
        c[i] = (double)0;
    }
}
```

main ↓

```
void serial_testrun() {
    double* a, * b, * c;

    init_vectors(a, b, c);
```

what is reference pointer

- int * means a pointer to an integer.
- The & in int *& means a reference to a pointer. Essentially, int *& is a reference to an int *. Any changes made to the reference will directly modify the pointer it refers to.




```
int n = 1;
int &r = n;
r = 10;
cout << n << endl; // 10
```

NVProf → evaluate performance

```
!./CudaTestrun.out

c[0]=0.000000
c[1]=2.000000
c[2]=4.000000
c[3]=6.000000
c[4]=8.000000
c[5]=10.000000
c[6]=12.000000
c[7]=14.000000
c[8]=16.000000
c[9]=18.000000
c[0]=0.000000
c[1]=2.000000
c[2]=4.000000
c[3]=6.000000
c[4]=8.000000
c[5]=10.000000
c[6]=12.000000
c[7]=14.000000
c[8]=16.000000
c[9]=18.000000

rof - profile performance of GPU

!nvprof ./CudaTestrun.out
```

```
==745== NVPROF is profiling process 745, command: ./CudaTestrun.out
c[0]=0.000000
c[1]=2.000000
c[2]=4.000000
c[3]=6.000000
c[4]=8.000000
c[5]=10.000000
c[6]=12.000000
c[7]=14.000000
c[8]=16.000000
c[9]=18.000000
==745== Profiling application: ./CudaTestrun.out
==745== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   64.98%  3.6284ms         2  1.8142ms  1.8127ms  1.8157ms  [CUDA memcpy HtoD]
                   33.37%  1.8631ms         1  1.8631ms  1.8631ms  1.8631ms  [CUDA memcpy DtoH]
                    1.65%  91.997us         1  91.997us  91.997us  91.997us  vectorAdditionKernel(double*, double*, double*, int)
```

By only copy from host to device, then processs multiple time and get back to host, don't copy multiple time and each time only process once

Make this smaller percentage

Make this bigger percentage

How to define block size and grid size for a problem


```
//configure the blocksize and gridsize
// block size - the number of threads in a block
dim3 blockSize(BLOCKSIZE, 1, 1); // x, y, z total up is 1024 threads which is the max threads we can use
// grid size = the number of blocks in grid - Formula [(MAX_SIZE-1)/blocksize] + 1
// Better Formula [MAX_SIZE + (blocksize -1)] / block_size
dim3 gridSize((MAX_SIZE + BLOCKSIZE) / BLOCKSIZE, 1);
```

problem size

```
const int BLOCKSIZE = 1024;

const int MAX_DIM = 100;
const int MAX_SIZE = MAX_DIM * MAX_DIM;
const int MAX_BYTES = MAX_SIZE * sizeof(float);

__global__ void matrix_mul(float* d_m1, float* d_m2, float* d_m3) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    // guard extra thread that in final block that not inside GPU
    // our MAX_SIZE is 10,000, the problem need to 10,000 runs
    // and our available threads is number of threads(blockSize) * number of blocks(gridSize) is 1024 * 10 = 10240,
    // there is extra 240 threads in the last block that we don't want to include in calculation.
    // How does I count the gridSize = 10
    // The Formula: (MAX_SIZE + BLOCKSIZE) / BLOCKSIZE
    // Assumming MAX_SIZE = 10,000, BLOCKSIZE = 1024, the answer is 10.7656 and discarded to integer become 10
    if (idx >= MAX_SIZE)
        return;
    float tempsum = 0;
    int i = idx / MAX_DIM;
    int j = idx % MAX_DIM;
    for (int k = 0; k < MAX_DIM; k++) {
        tempsum += d_m1[i * MAX_DIM + k] * d_m2[j + k * MAX_DIM];
    }
    d_m3[idx] = tempsum;
}
```

need 10,000 thread

(100 x 100) = 10,000

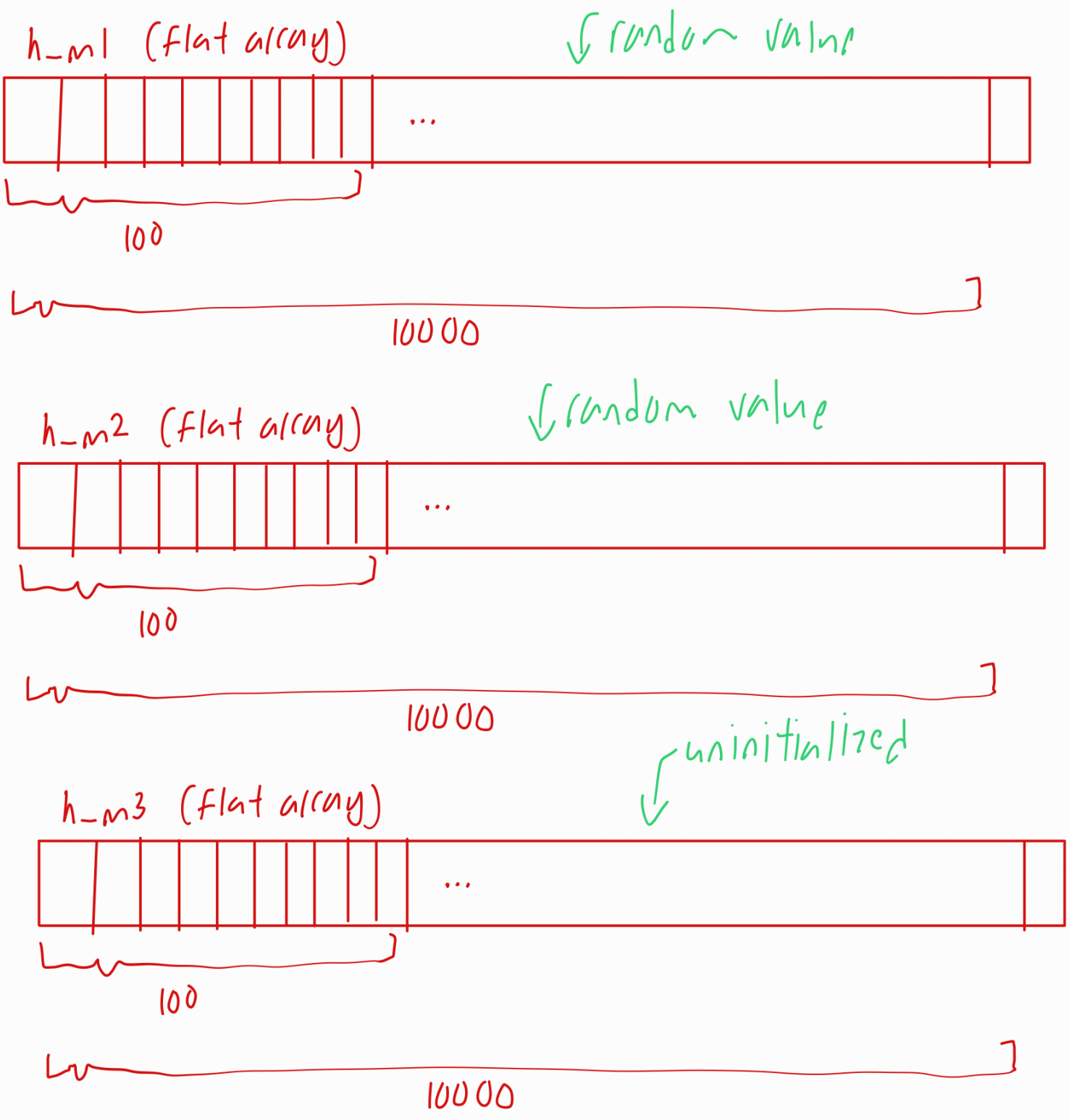
# Explanation on 10-Matrix Multiply.cu

```
const int BLOCKSIZE = 1024;


const int MAX_DIM = 100;
const int MAX_SIZE = MAX_DIM * MAX_DIM;
```

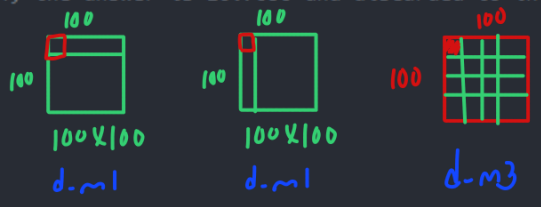
```
// allocating memory for host arrays
float h_m1[MAX_SIZE], h_m2[MAX_SIZE], h_m3[MAX_SIZE];

// generating input arrays
for (int i = 0; i < MAX_SIZE; i++)
    h_m1[i] = (float)(rand() % 100);
for (int i = 0; i < MAX_SIZE; i++)
    h_m2[i] = (float)(rand() % 100);

// declaring device memory pointers
float *d_m1, *d_m2, *d_m3;

// allocating device memory
cudaMalloc((void **)&d_m1, MAX_SIZE * sizeof(float));
cudaMalloc((void **)&d_m2, MAX_SIZE * sizeof(float));
cudaMalloc((void **)&d_m3, MAX_SIZE * sizeof(float));

// copying data from host to device
cudaMemcpy(d_m1, h_m1, MAX_SIZE * sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_m2, h_m2, MAX_SIZE * sizeof(float), cudaMemcpyHostToDevice);

// configure the blocksize and gridsize
//  block size - the number of threads in a block
dim3 blockSize(BLOCKSIZE, 1, 1); // x, y, z total up is 1024 threads which is the max threads we can use
// grid size = the number of blocks in grid - Formula [(MAX_SIZE-1)/blocksize] + 1
// Better Formula [MAX_SIZE + (blocksize -1)] / block_size
dim3 gridSize((MAX_SIZE + BLOCKSIZE - 1) / BLOCKSIZE, 1);

// calling kernel
matrix_mul<<<gridSize, blockSize>>>(d_m1, d_m2, d_m3);

// transferring result from device to host
cudaMemcpy(h_m3, d_m3, MAX_SIZE * sizeof(float), cudaMemcpyDeviceToHost);
```

1024

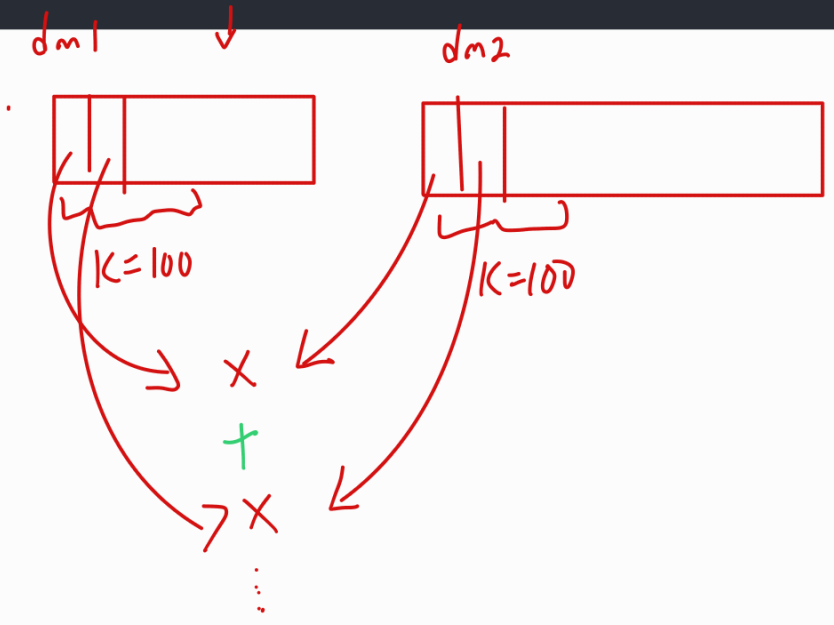
```
// free allocated device memory
cudaFree(d_m1);
cudaFree(d_m2);
cudaFree(d_m3);
```


h_m1 (flat array)
random value
...
100
10000
h_m2 (flat array)
random value
...
100
10000
h_m3 (flat array)
uninitialized
...
100
10000

```cpp
__global__ void matrix_mul(float *d_m1, float *d_m2, float *d_m3)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    // guard extra thread that in final block that not inside GPU
    // our MAX_SIZE is 10,000, the problem need to 10,000 runs
    // and our available threads is number of threads(blockSize) * number of blocks(gridSize) is 1024 * 10 = 10240,
    // there is extra 240 threads in the last block that we don't want to include in calculation.
    // How does I count the gridSize = 10
    // The Formula: (MAX_SIZE + BLOCKSIZE) / BLOCKSIZE
    // Assumming MAX_SIZE = 10,000, BLOCKSIZE = 1024, the answer is 10.7656 and discarded to integer become 10
    if (idx >= MAX_SIZE)
        return;
    float tempsum = 0;
    int i = idx / MAX_DIM;
    int j = idx % MAX_DIM;
    for (int k = 0; k < MAX_DIM; k++)
    {
        tempsum += d_m1[i * MAX_DIM + k] * d_m2[j + k * MAX_DIM];
    }
    d_m3[idx] = tempsum;
}
```


0 / 100 = 0
0 % 100 = 0
100
0 * 100 + 0 = 0
0 + 0 * 100 = 0



Explanation on 11_matrix_multiply2d.cu

```
#define BLOCKSIZE 16
const int MAX_DIM = 100;
```

```
int main()
{
    // Matrix dimensions
    int Width = MAX_DIM;

    // Allocating memory for host arrays
    float h_M[MAX_DIM][MAX_DIM], h_N[MAX_DIM][MAX_DIM], h_P[MAX_DIM][MAX_DIM];

    // Generating input arrays
    for (int i = 0; i < MAX_DIM; i++)
    {
        for (int j = 0; j < MAX_DIM; j++)
        {
            h_M[i][j] = (float)(rand() % 100);
            h_N[i][j] = (float)(rand() % 100);
        }
    }

    // Declaring device memory pointers
    float *d_M, *d_N, *d_P;
      You, 24 minutes ago • Update Matrix Multiply for 2D Version
    // Allocating device memory
    cudaMalloc((void **)&d_M, MAX_DIM * MAX_DIM * sizeof(float));
    cudaMalloc((void **)&d_N, MAX_DIM * MAX_DIM * sizeof(float));
    cudaMalloc((void **)&d_P, MAX_DIM * MAX_DIM * sizeof(float));

    // Copying data from host to device
    cudaMemcpy(d_M, h_M, MAX_DIM * MAX_DIM * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(d_N, h_N, MAX_DIM * MAX_DIM * sizeof(float), cudaMemcpyHostToDevice);

    // Configuring the block size and grid size
    dim3 blockSize(BLOCKSIZE, BLOCKSIZE, 1);
    dim3 gridSize((Width + BLOCKSIZE - 1) / BLOCKSIZE, (Width + BLOCKSIZE - 1) / BLOCKSIZE, 1);

    // Calling the kernel
    MatrixMulKernel<<<gridSize, blockSize>>>(d_M, d_N, d_P, Width);

    // Transferring result from device to host
    cudaMemcpy(h_P, d_P, MAX_DIM * MAX_DIM * sizeof(float), cudaMemcpyDeviceToHost);
```

16 x 16

```
// Free allocated device memory
cudaFree(d_M);
cudaFree(d_N);
cudaFree(d_P);
```

```cpp
__global__ void MatrixMulKernel(float *M, float *N, float *P, int Width)
{
    // Calculate the row index of the P element and M
    int Row = blockIdx.y * blockDim.y + threadIdx.y;
    // Calculate the column index of P and N
    int Col = blockIdx.x * blockDim.x + threadIdx.x;

    if ((Row < Width) && (Col < Width))
    {
        float Pvalue = 0;
        // Each thread computes one element of the block sub-matrix
        for (int k = 0; k < Width; ++k)
        {
            Pvalue += M[Row * Width + k] * N[k * Width + Col];
        }
        P[Row * Width + Col] = Pvalue;
    }
}
```

MATRIX MULTIPLICATION