1. So far, one thread is responsible for one data element, can you change this, say one thread takes care of several data entries ?

test N = 512*10

```
// test  C = A + B
void runTest(int argc, char** argv)
{
    unsigned int N = 512*10 ;
    printf("N = %d\n", N);
```

```
    vecadd_GPU( h_C, h_A, h_B, N) ;
```

```
void vecadd_GPU(float* h_C, const float* h_A, const float* h_B, unsigned int N )
{
    unsigned int num_block = 1 ;
    unsigned int threads = 512 ;
    unsigned int mem_size_A = sizeof(float) * N ;
    unsigned int mem_size_B = sizeof(float) * N ;
```

```
   // execute the kernel
 vecadd<<< num_block, threads >>>(d_C, d_A, d_B, N)
```

```
#include <stdio.h>
#include <assert.h>

__global__  void vecadd( float* C, float* A, float* B, int N)
{
/*
#ifdef __DEVICE_EMULATION__
  int bx = blockIdx.x ;
  assert( 0 == bx) ;
#endif
*/
    int k ;
    int i = threadIdx.x ;

    for( k = 0 ; k < N/512 ; k++ ){
        C[i + 512*k] = A[i + 512*k] + B[i + 512*k] ;
    }

}
```

We only use 512 threads to do 512*10 addition by for loop

2. Maximum number of threads per block is 512, when data set is more than 512, we use multi-thread-block to do parallel computing, however Maximum size of each dimension of a grid of thread blocks is 65535, when data set is more than 131MB, how can we proceed?

We can use more than one-dimension (either block or thread) to do parallel computing or we can let each thread do more than one thing as before .

3. From table 2, data transfer from device to host is about half of CPU computation, it means that if we can accelerate CPU computation, then GPU has no advantage, right?

Not exactly, it depends on the comparison of computation between CPU and GPU. If GPU is much faster than CPU, then maybe the time waste on data transfer can be paid back . Moreover, if you can accelerate CPU computation , then maybe there is a way to accelerate the data transfer ,too !

4. measure your video card and fill-in table 2, also try double-precision if your hardware
supports.

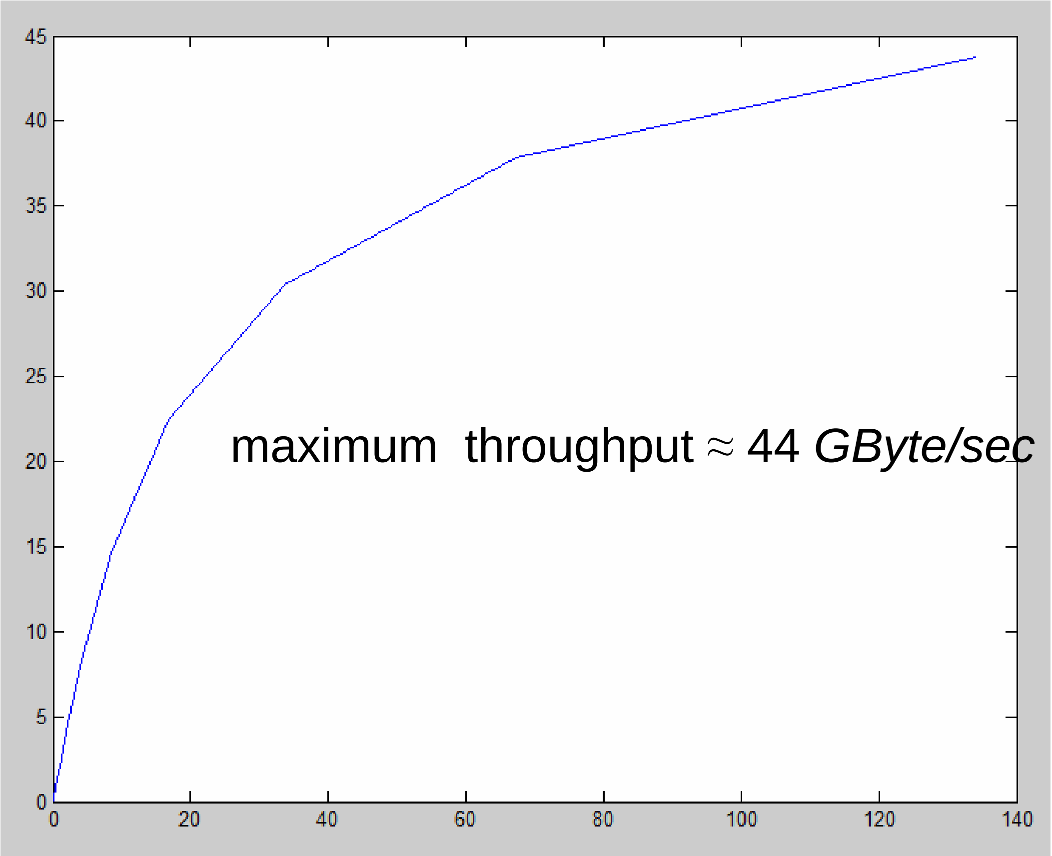
Visual2005 GTX260

Copy C from device to host

$C = A + B$

Table 2

# of block	size	GPU (ms)	Device → Host (ms)	CPU (ms)
16	32 KB	1.156013	0.097848	0
32	64 KB	1.137016	0.103924	0
64	128 KB	1.15099	0.148483	0
128	256 KB	1.154407	0.267771	0
256	512 KB	1.135270	0.486165	0
512	1.024 MB	1.534413	1.498584	2
1024	2.048 MB	1.375454	1.523029	3
2048	4.096 MB	1.513810	2.812648	5
4096	8.192 MB	1.721238	6.170896	11
8192	16.384 MB	2.244629	11.330351	21
16384	32.768 MB	3.312502	24.299248	44
32768	65.536 MB	5.324490	43.954819	89
65535	131 MB	9.193068	96.222427	192



Geforce GTX260

Memory Specs:

Memory Clock (MHz)	999 MHz
Standard Memory Config	896 MB
Memory Interface Width	448-bit
Memory Bandwidth (GB/sec)	111.9

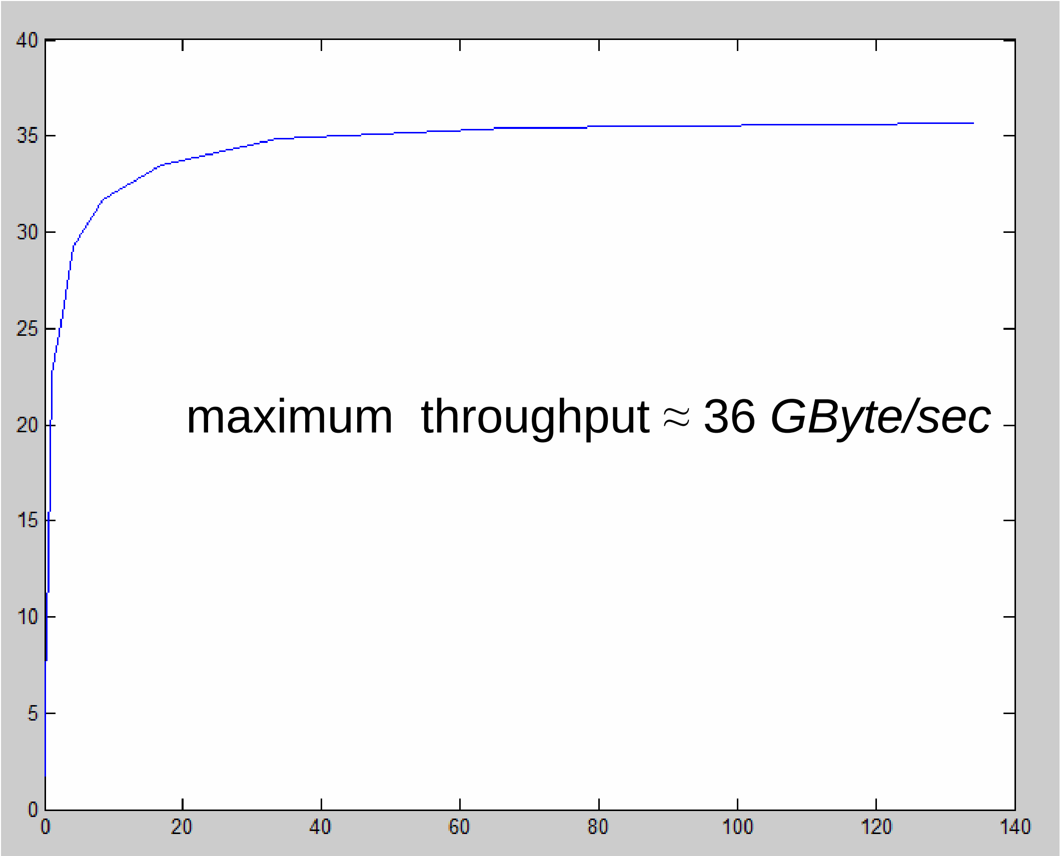
Linux machine GeForce 9600GT

Table 2

$C = A + B$ ← GPU (ms)

Copy C from device to host ← Device → Host (ms)

# of block	size	GPU (ms)	Device → Host (ms)	CPU (ms)
16	32 KB	0.057000	0.053000	0
32	64 KB	0.061	0.092	0
64	128 KB	0.065000	0.186000	0
128	256 KB	0.084000	0.335000	0
256	512 KB	0.120000	0.803000	0
512	1.024 MB	0.169000	1.538000	10
1024	2.048 MB	0.254000	2.358000	0
2048	4.096 MB	0.430000	4.511000	10
4096	8.192 MB	0.794000	10.279000	10
8192	16.384 MB	1.505000	17.690001	20
16384	32.768 MB	2.885000	34.956001	60
32768	65.536 MB	5.689000	69.507004	120
65535	131 MB	11.299000	138.901001	240



Memory Specs: Geforce 9600GT

Memory Clock (MHz)	900 MHz
Standard Memory Config	512 MB
Memory Interface Width	256-bit
Memory Bandwidth (GB/sec)	57.6

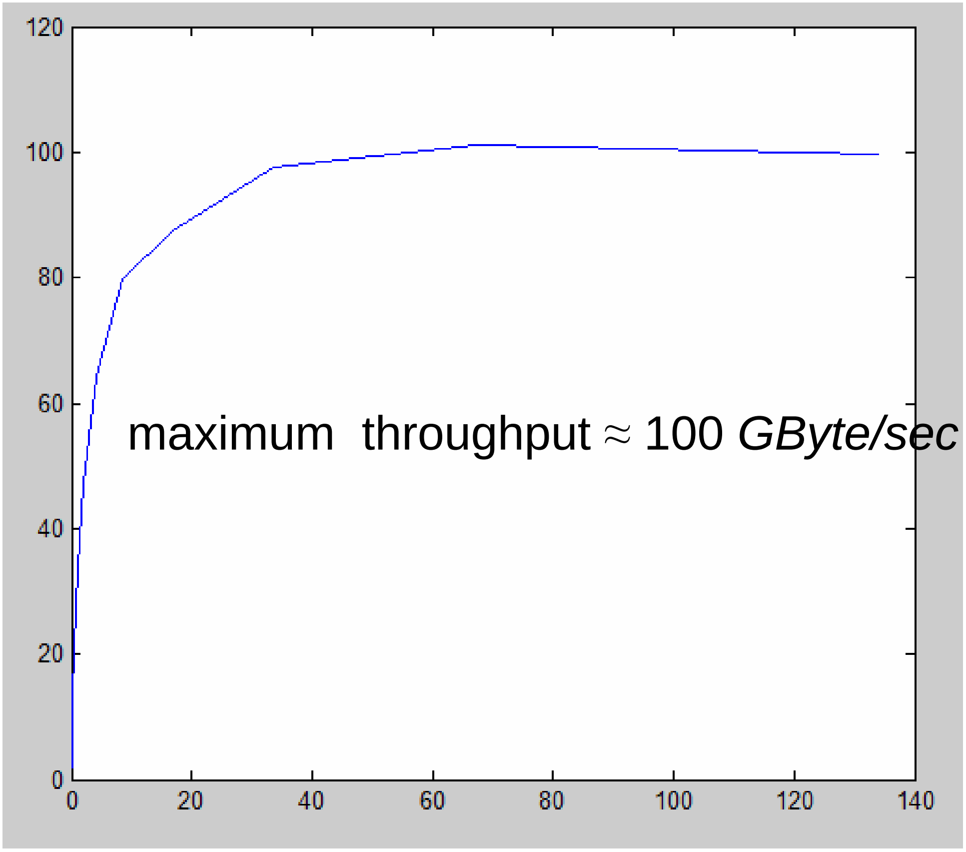
Linux machine GTX260

$C = A + B$

Copy C from device to host

Table 2

# of block	size	GPU (ms)	Device → Host (ms)	CPU (ms)
16	32 KB	0.050000	0.044000	0
32	64 KB	0.057000	0.074000	0
64	128 KB	0.054000	0.138000	0
128	256 KB	0.059000	0.264000	0
256	512 KB	0.076000	0.528000	0
512	1.024 MB	0.093000	1.040000	0
1024	2.048 MB	0.134000	1.926000	0
2048	4.096 MB	0.196000	3.702000	0
4096	8.192 MB	0.315000	7.295000	10
8192	16.384 MB	0.575000	14.424000	30
16384	32.768 MB	1.033000	28.607000	50
32768	65.536 MB	1.993000	57.006001	90
65535	131 MB	4.046000	113.746002	190



Geforce GTX260

Memory Specs:

Memory Clock (MHz)	999 MHz
Standard Memory Config	896 MB
Memory Interface Width	448-bit
Memory Bandwidth (GB/sec)	111.9

5. modify code in matrixMul, measure time for computing golden vector , time for C = A*B under GPU and time for data transfer, compare them.

WA = HA = WB = 200

```
[benzema@matrix matrixmultiply]$ make nvcc_run
nvcc -run -I/usr/local/NVIDIA_CUDA_SDK/common/inc
 matrixMul.cu  matrixMul_gold.cpp
Using device 0: GeForce 9600 GT
host --> device (A): 20.926001 (ms)
host --> device (B): 20.288000 (ms)
in GPU, C = A*B: 1329.659058 (ms)
device --> Host (C): 42.502998 (ms)
compute golden vector needs 450850.0000 (ms)
Test PASSED

Press ENTER to exit...
```

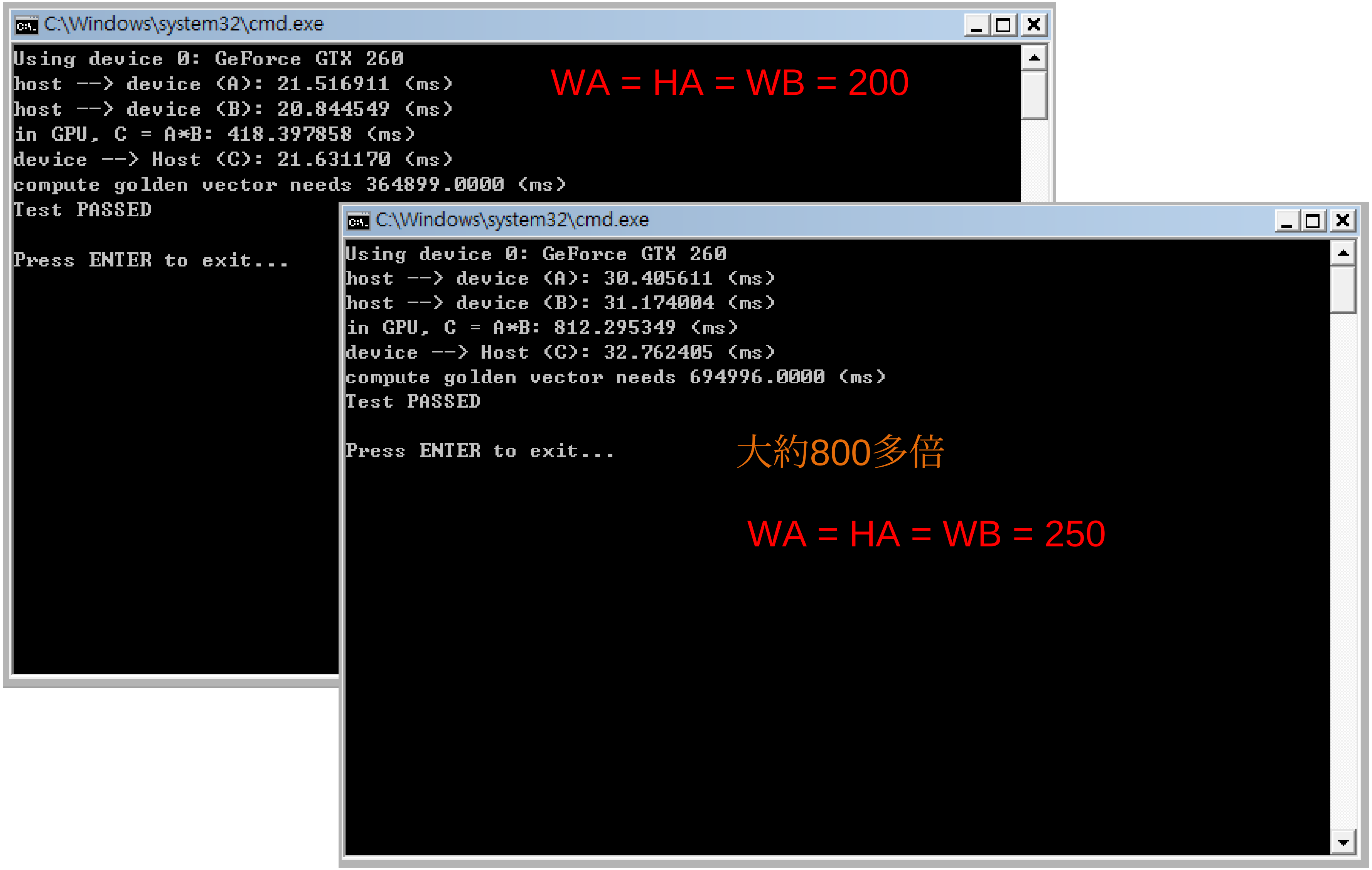
大約 300 倍

WA = HA = WB = 250

```
[benzema@matrix matrixmultiply]$ make nvcc_run
nvcc -run -I/usr/local/NVIDIA_CUDA_SDK/common/inc
 matrixMul.cu  matrixMul_gold.cpp
Using device 0: GeForce 9600 GT
host --> device (A): 32.146000 (ms)
host --> device (B): 31.483000 (ms)
in GPU, C = A*B: 2609.721924 (ms)
device --> Host (C): 66.496002 (ms)
compute golden vector needs 874180.0000 (ms)
Test PASSED

Press ENTER to exit...
```

```
void
computeGold(float* C, const float* A, const float* B, unsigned int hA, unsigned int wA, unsigned int wB)
{
// compute the time for golden vector
    clock_t  start, end ;

    start = clock() ;
    for (unsigned int i = 0; i < hA; ++i)
        for (unsigned int j = 0; j < wB; ++j) {
            double sum = 0;
            for (unsigned int k = 0; k < wA; ++k) {
                double a = A[i * wA + k];
                double b = B[k * wB + j];
                sum += a * b;
            }
            C[i * wB + j] = (float)sum;
        }
    end = clock() ;
    double dt = ((double)(end - start))/((double)CLOCKS_PER_SEC) * 1000.(
    printf("compute golden vector needs %10.4f (ms)\n", dt );
}
```

```
    // create and start timer
    unsigned int timer = 0;
    CUT_SAFE_CALL(cutCreateTimer(&timer));
    CUT_SAFE_CALL(cutStartTimer(timer));
// for A
    // copy host memory to device
    CUDA_SAFE_CALL(cudaMemcpy(d_A, h_A, mem_size_A,
                              cudaMemcpyHostToDevice) );
    // stop and destroy timer
    CUT_SAFE_CALL(cutStopTimer(timer));
    printf("host --> device (A): %f (ms)\n",
        cutGetTimerValue(timer));
    CUT_SAFE_CALL(cutDeleteTimer(timer));
```

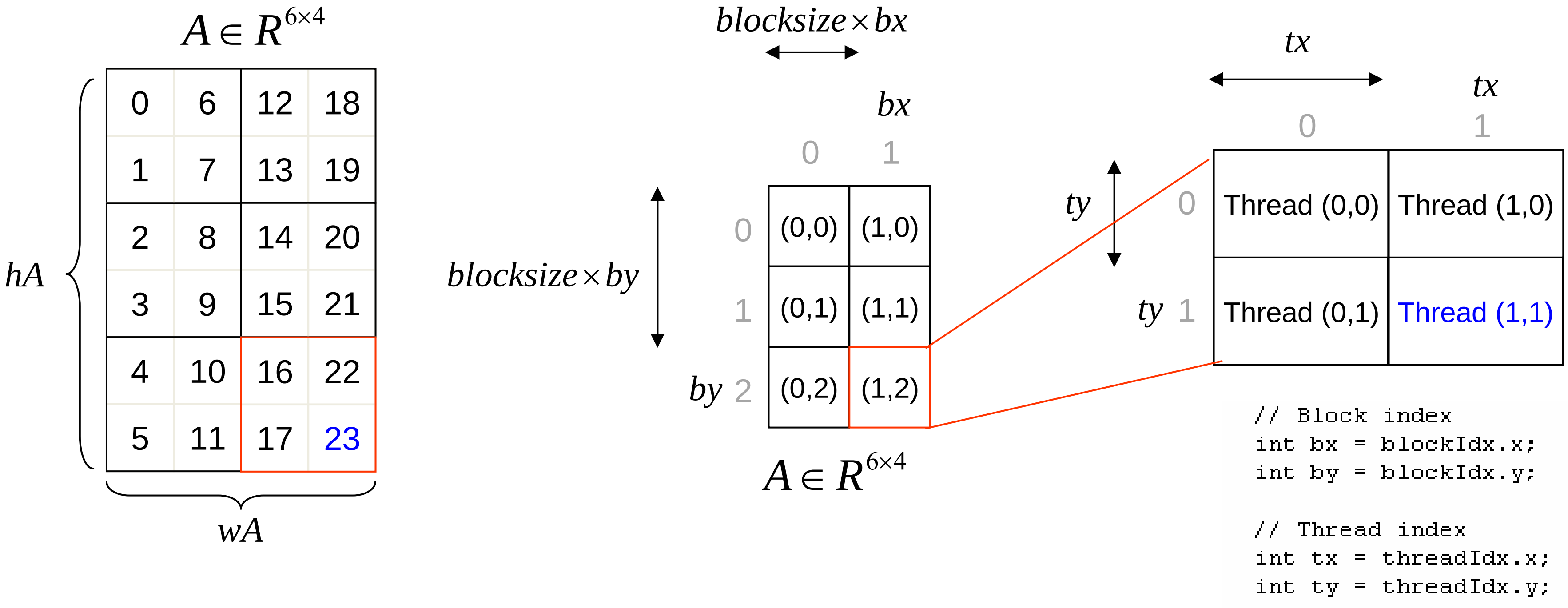


Q1 : why the speed of “host to device “ and “device to host” are different in Linux

Q2 : why can’t we let WA = HA = WB = 300 ? Do we use all storage ?

6. We have shown you vector addition and matrix-matrix product, which one is better in GPU computation, why?(you can compute ratio between floating point operation and
memory fetch operation)

7. modify source code in matrixMul, use column-major index, be careful indexing rule.



The physical index of first entry in block $(bx, by) = (bx \times blocksize) \times hA + blocksize \times by$

e.g. The physical index of first entry in block $(1,2) = (1 \times 2) \times 6 + 2 \times 2 = 12 + 4 = 16$

The physical index of (block index, thread index) is $((bx,by),(tx,ty)) = (bx,by) + (tx \times hA) + ty$

e.g. $((bx,by),(tx,ty)) = ((1,2),(1,1)) = 16 + (1 \times 6) + 1 = 23$

global index

$\big((bx,by),(tx,ty)\big) \longrightarrow (blocksize \times bx + tx,\ blocksize \times by + ty) \longrightarrow$ col-major

Modify code in matrixMul_kernel.cu

```
#define AS(i, j) As[i][j]
#define BS(i, j) Bs[i][j]
```

(I choose to keep this notation and modify the code below)

Part 1: copy $A\begin{pmatrix} (0,1) \end{pmatrix}$ to As and $B\begin{pmatrix} (1,0) \end{pmatrix}$ to Bs

```
    // Index of the first sub-matrix of A processed by the block
    int aBegin = BLOCK_SIZE * by;

    // Index of the last sub-matrix of A processed by the block
// HA = gridDim.y * blockDim.y
// WA = gridDim.x * blockDim.x
// BLOCK_SIZE =  blockDim.x = blockDim.y
//  int aEnd   = aBegin + (gridDim.x-1)*blockDim.x*HA - 1;
    int aEnd   = aBegin + (WA-BLOCK_SIZE)*HA + 1;

    // Step size used to iterate through the sub-matrices of A
    int aStep  = BLOCK_SIZE * HA ;

    // Index of the first sub-matrix of B processed by the block
    int bBegin = BLOCK_SIZE * bx * HA ;

    // Step size used to iterate through the sub-matrices of B
    int bStep  = BLOCK_SIZE ;
```

```
AS(tx, ty) = A[a + HA * tx + ty];
BS(tx, ty) = B[b + HB * tx + ty];
```

Part 2: add first product term to submatrix of C

```
for (int k = 0; k < BLOCK_SIZE; ++k)
    Csub += AS(k, ty) * BS(tx, k);
```

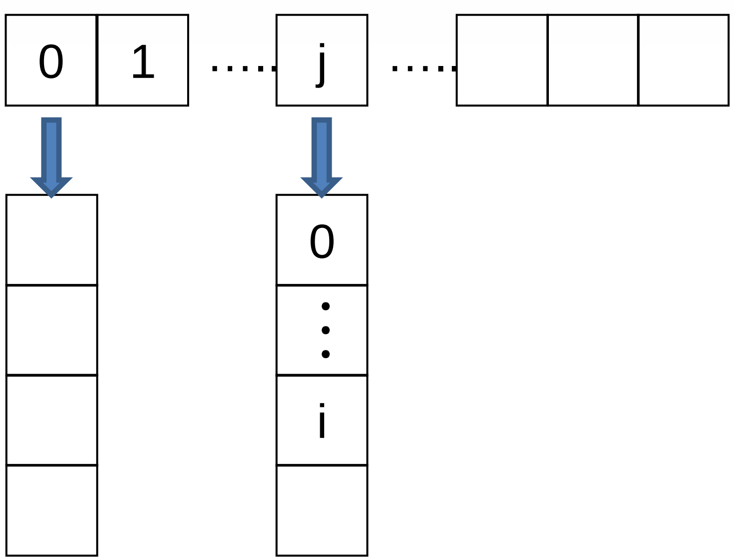
Part 3: rewrite Csub to C

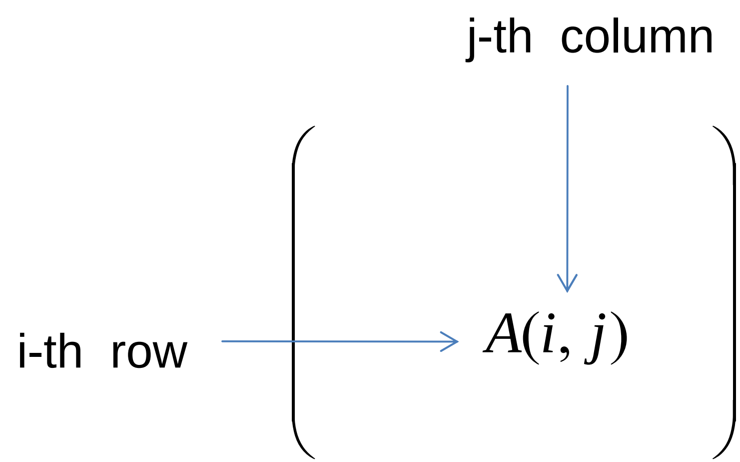
```
int c = HA * BLOCK_SIZE * bx + BLOCK_SIZE * by;
C[c + HA * tx + ty] = Csub;
```

Modify code in matrixMul_gold.cpp

```
void
computeGold(float* C, const float* A, const float* B, unsigned int hA, unsigned int wA, unsigned int wB)
{
// compute the time for golden vector
    clock_t  start, end ;

    start = clock() ;
    for (unsigned int i = 0; i < hA; ++i)
        for (unsigned int j = 0; j < wB; ++j) {
            double sum = 0;
            for (unsigned int k = 0; k < wA; ++k) {
                double a = A[k * hA + i];  // A(i,k)
                double b = B[j * wA + k];  // B(k,j)    hB = wA
                sum += a * b;
            }
            C[j * hA + i] = (float)sum;
        }

    end = clock() ;
    double dt = ((double)(end - start))/((double)CLOCKS_PER_SEC) * 1000.0 ;
    printf("compute golden vector needs %10.4f (ms)\n", dt );

}
```

$$A(i,j) = A[j \times hA + i]$$





8. We have discussed that matrix-vector product has two versions, one is inner-product-based, one is outer-product-based, implement these two methods under GPU