Chapter 20　rounding error

Speaker: Lung-Sheng Chien

OutLine

- Store (a+b) != (a+b)
- Rounding error in GPU
- Qt4 + vc2005
- Performance evaluation: GPU v.s. CPU

Exercise 5: verify subroutine ***matrixMul_block_seq*** with non-block version, you can use high precision package.

Non-block version

```
12  // c = a * B
13  void matrixMul_seq(doublereal* C, const doublereal* A, const doublereal* B,
14    unsigned int hA, unsigned int wA, unsigned int wB )
15  {
16      unsigned int i, j, k ;
17      doublereal sum ;
18      doublereal a, b ;
19
20      for ( i = 0; i < hA; ++i)
21          for (j = 0; j < wB; ++j) {
22              sum = 0.0 ;
23              for (k = 0; k < wA; ++k) {
24                  a = A[i * wA + k];
25                  b = B[k * wB + j];
26                  sum += a * b;
27              }
28              C[i * wB + j] = sum;
29          }
30  }
```

block version

```
60          c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
61            // Multiply the two matrices together
62              for ( ty = 0 ; ty < BLOCK_SIZE ; ty++ ){
63                for ( tx = 0 ; tx < BLOCK_SIZE ; tx++ ){
64                  Csub = 0.0 ;
65                  for (k = 0; k < BLOCK_SIZE; ++k ){
66                      Asub = As[ty][k ] ;
67                      Bsub = Bs[k ][tx] ;
68                      Csub += Asub * Bsub ;
69                  }
70              C[c + wB * ty + tx] += Csub;
71                }// for tx ;
72              }// for ty
```

matrix_Mul_seq is different from *matrixMul_block_seq* since
(1)Associativity does not hold in Floating point operation
(2)Rounding error due to type conversion or computation

Integer multiplication: type float [1]

BLOCK_SIZE = 2

A

6589	7678	8574	6180
23823	24497	26022	25644

$\times$

B

4252	18356
4065	18154
11087	30995
6303	29283

$=$

C

1,9323,9976	7,0705,4166
6,4871,9667	24,2958,6428

$C(non-block)$

1.93239968E8	7.07054208E8
6.48719680E8	2.429586432E9

$\neq$

$C(block)$

1.93239968E8	7.07054144E8
6.48719616E8	2.429586432E9

$\neq C$

$$\Delta C \equiv \left| C(non-block) - C(block) \right| = \begin{bmatrix} 0 & 64 \\ 64 & 0 \end{bmatrix}$$

$$\left| C(non-block) - C \right| = \begin{bmatrix} 8 & 42 \\ 13 & 4 \end{bmatrix}$$

Exercise 1: If we choose type “double”, then $C(non-block) = C(block) = C$

Integer multiplication: type float [2]

Non-block: $7.07054208E8 = \left(\left(\left(A_{11}B_{12} + A_{12}B_{22}\right) + A_{13}B_{32}\right) + A_{14}B_{42}\right)$

A

6589	7678	8574	6180
23823	24497	26022	25644

$\times$

B

4252	18356
4065	18154
11087	30995
6303	29283

$=$

$C\left(non-block\right)$

1.93239968E8	7.07054208E8
6.48719680E8	2.429586432E9

```
for ( i = 0; i < hA; ++i)
    for (j = 0; j < wB; ++j) {
        sum = 0.0 ;
        for (k = 0; k < wA; ++k) {
            a = A[i * wA + k];
            b = B[k * wB + j];
            sum += a * b;
        }
        C[i * wB + j] = sum;
    }
```

$sum := 0 \longrightarrow sum += A_{11}B_{12} \longrightarrow sum += A_{12}B_{22} \longrightarrow sum += A_{13}B_{32} \longrightarrow sum += A_{14}B_{42}$

Integer multiplication: type float [3]

block: $7.07054144E8 = (A_{11}B_{12} + A_{12}B_{22}) + (A_{13}B_{32} + A_{14}B_{42})$ *BLOCK_SIZE = 2*

A

6589	7678	8574	6180
23823	24497	26022	25644

$\times$

B

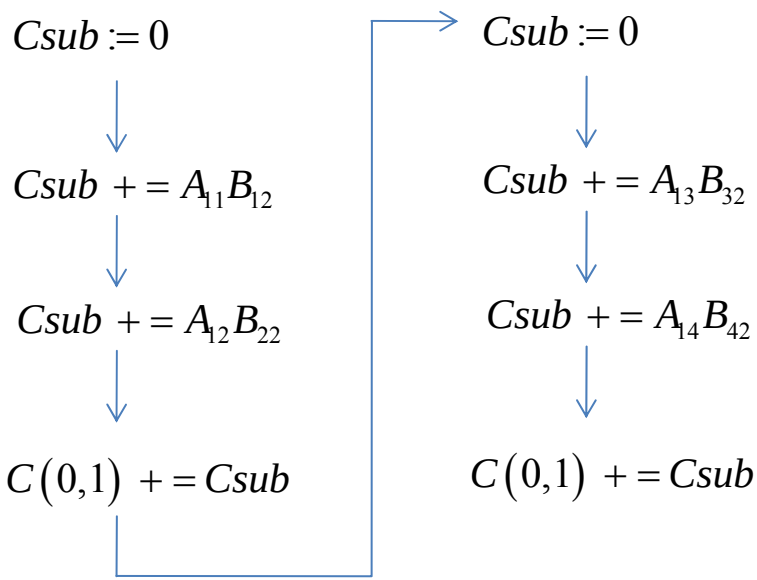
4252	18356
4065	18154
11087	30995
6303	29283

$=$

$C(block)$

1.93239968E8	7.07054144E8
6.48719616E8	2.429586432E9

```
c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
// Multiply the two matrices together
  for ( ty = 0 ; ty < BLOCK_SIZE ; ty++ ){
    for ( tx = 0 ; tx < BLOCK_SIZE ; tx++ ){
      Csub = 0.0 ;
      for (k = 0; k < BLOCK_SIZE; ++k ){
          Asub = As[ty][k ] ;
          Bsub = Bs[k ][tx] ;
          Csub += Asub * Bsub ;
      }
      C[c + wB * ty + tx] += Csub;
    }// for tx ;
  }// for ty
```



Integer multiplication: type float [4]

A

6589	7678	8574	6180
23823	24497	26022	25644

B

4252	18356
4065	18154
11087	30995
6303	29283

$=$

$C(block)$

1.93239968E8	7.07054144E8
6.48719616E8	2.429586432E9

8574	6180	$\times$	30995	$=$	446720070	$\xrightarrow{type\ \ float}$	4.46720064E8
			29283				

Question 1: what happens when do type conversion? Why cannot we keep accuracy?

$$446720070 = 1\times16^7 + 10\times16^6 + 10\times16^5 + 0\times16^4 + 6\times16^3 + 8\times16^2 + 4\times16^1 + 6\times16^0$$

$$= 0x1AA06846$$

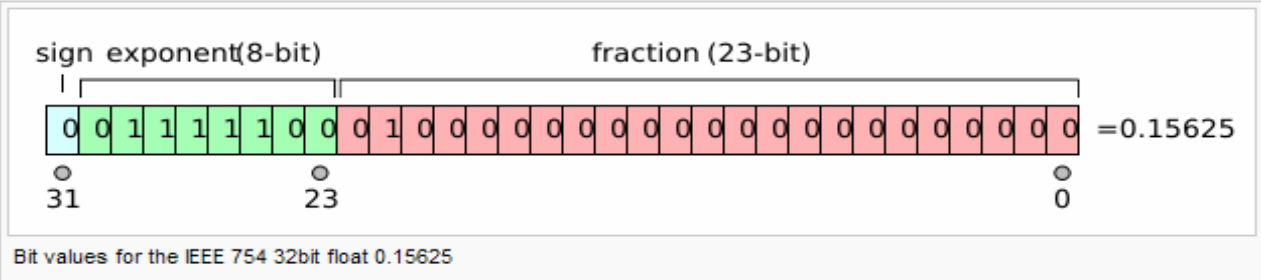
$$= 1,1010,1010,0000,0110,1000,0100,0110$$

$$= 1.1010\ 1010\ 0000\ 0110\ 1000\ 0100\ 0110\times2^{28}$$

$$= 1 + \frac{10}{16} + \frac{10}{16^2} + \frac{0}{16^3} + \frac{6}{16^4} + \frac{8}{16^5} + \frac{4}{16^6} + \frac{6}{16^7}$$

Integer multiplication: type float [5]

IEEE 754: single precision (32-bit)



Bit values for the IEEE 754 32bit float 0.15625

8 bit excess-127

Binary value	Excess-127 interpretation	Unsigned interpretation
00000000	-127	0
00000001	-126	1
...	...	...
01111111	0	127
10000000	+1	128
...	...	...
11111111	+128	255

$$s = 0 : v > 0$$

$$\exp = 01111100 = 0x7C = 7 \times 16 + 12 = 124 \qquad N = \exp - 127 = -3$$

$$m = 1.01 = 1 + \frac{1}{4} = 1.25$$

Normalized value: $v = s \times 2^{E} \times m = +1.25 \times 2^{-3} = +0.15625$

$$446720070 = 1.1010\ 1010\ 0000\ 0110\ 1000\ 0100\ 0110 \times 2^{28}$$

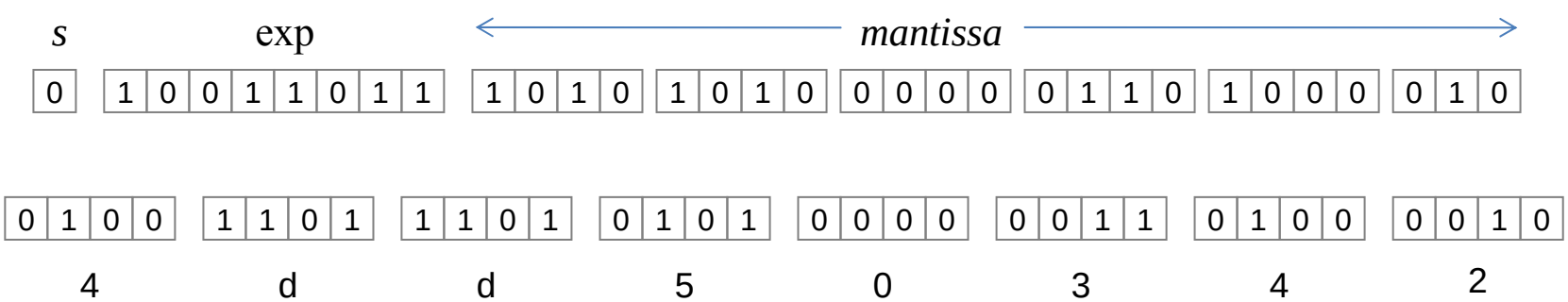
1. $N = 28 \longrightarrow \exp = 127 + 28 = 155 = 9 \times 16 + 11 = 1001{,}1011$
2. Mantissa (fraction) has 23-bits, we need to do rounding (cause of rounding -error)

$$1010\ 1010\ 0000\ 0110\ 1000\ 0100\ 0110 \longrightarrow 1010\ 1010\ 0000\ 0110\ 1000\ 010$$

3. sign bit = 0 (positive)

Integer multiplication: type float [6]

$s = 0 \qquad \exp = 1001,1011 \qquad mantissa = 1010\ 1010\ 0000\ 0110\ 1000\ 010$



Exercise 2: write a program to show $446720070 = 0x4dd50342$

$C(non-block)$

1.93239968E8	7.07054208E8
6.48719680E8	2.429586432E9

$C(block)$

1.93239968E8	7.07054144E8
6.48719616E8	2.429586432E9

ΔC

0	64
64	0

$\left|\frac{\Delta C}{C(non-block)}\right| \sim O(10^{-7})$ reaches machine zero of single precision

Integer multiplication: type float [7]

Exercise 3: modify subroutine *matrixMul_block_parallel* as following such that it is the same as subroutine *matrixMul_seq.*
How about performance? Compare your experiment with original code.

```
for (a = aBegin, b = bBegin;
     a <= aEnd;
     a += aStep, b += bStep) {
// Load the matrices from main memory
  for ( ty = 0 ; ty < BLOCK_SIZE ; ty++ ){
    for ( tx = 0 ; tx < BLOCK_SIZE ; tx++ ){
      As[ty][tx] = A[a + wA * ty + tx];
      Bs[ty][tx] = B[b + wB * ty + tx];
    }// for tx ;
  }// for ty

  c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
// Multiply the two matrices together
  for ( ty = 0 ; ty < BLOCK_SIZE ; ty++ ){
    for ( tx = 0 ; tx < BLOCK_SIZE ; tx++ ){
      for (k = 0; k < BLOCK_SIZE; ++k ){
         Asub = As[ty][k ] ;
         Bsub = Bs[k ][tx] ;

         C[c + wB * ty + tx] += Asub * Bsub ;

      }// for k
    }// for tx ;
  }// for ty
}// for each submatrix A and B
```

```
// C = A * B
void matrixMul_seq(doublereal* C, const doublereal* A,
  unsigned int hA, unsigned int wA, unsigned int wB )
{
    unsigned int i, j, k ;
    doublereal sum ;
    doublereal a, b ;

    for ( i = 0; i < hA; ++i)
        for (j = 0; j < wB; ++j) {
            sum = 0.0 ;
            for (k = 0; k < wA; ++k) {
                a = A[i * wA + k];
                b = B[k * wB + j];
                sum += a * b;
            }
            C[i * wB + j] = sum;
        }
}
```

OutLine

- Store (a+b) != (a+b)
- Rounding error in GPU
  - FMAD operation in GPU
  - rounding mode
  - mathematical formulation of rounding error
- Qt4 + vc2005
- Performance evaluation: GPU v.s. CPU

Rounding error in GPU computation [1]

Question 2: in matrix multiplication, when matrix size > 250x16, then error occurs in matrix product, why?

matrixMul.cu

```
129     // setup execution parameters
130     dim3 threads(BLOCK_SIZE, BLOCK_SIZE);
131     dim3 grid(WC / threads.x, HC / threads.y);
132
133     // execute the kernel
134     matrixMul<<< grid, threads >>>(d_C, d_A, d_B, WA, WB);
135
136     // check if kernel execution generated and error
137     CUT_CHECK_ERROR("Kernel execution failed");
138
139     // copy result from device to host
140     CUDA_SAFE_CALL(cudaMemcpy(h_C, d_C, mem_size_C,
141                               cudaMemcpyDeviceToHost) );
142
143     // stop and destroy timer
144     CUT_SAFE_CALL(cutStopTimer(timer));
145     printf("Processing time: %f (ms) \n", cutGetTimerValue(timer));
146     CUT_SAFE_CALL(cutDeleteTimer(timer));
147
148     // compute reference solution
149     float* reference = (float*) malloc(mem_size_C);
150     computeGold(reference, h_A, h_B, HA, WA, WB);
151
152     // check result
153     CUTBoolean res = cutCompareL2fe(reference, h_C, size_C, 1e-6f);
154     printf("Test %s \n", (1 == res) ? "PASSED" : "FAILED");
155     if (res!=1) printDiff(reference, h_C, WC, HC);
156
```

What is functionality of cutCompareL2fe?

verify $\| reference - h\_C \|$

Print all data

Rounding error in GPU computation [2]

C:\Program Files (x86)\NVIDIA Corporation\NVIDIA CUDA SDK\common\src\cutil.cpp

```
1343  CUTBoolean CUTIL_API
1344  cutCompareL2fe( const float* reference, const float* data,
1345                  const unsigned int len, const float epsilon )
1346  {
1347      CUT_CONDITION( epsilon >= 0);
1348
1349      float error = 0;
1350      float ref = 0;
1351
1352      for( unsigned int i = 0; i < len; ++i) {
1353
1354          float diff = reference[i] - data[i];
1355          error += diff * diff;
1356          ref += reference[i] * reference[i];
1357      }
1358
1359      float normRef = sqrtf(ref);
1360      if (fabs(ref) < 1e-7) {
1361  #ifdef _DEBUG
1362          std::cerr << "ERROR, reference l2-norm is 0\n";
1363  #endif
1364          return CUTFalse;
1365      }
1366      float normError = sqrtf(error);
1367      error = normError / normRef;
1368      bool result = error < epsilon;
1369  #ifdef _DEBUG
1370      if( ! result)
1371      {
1372          std::cerr << "ERROR, l2-norm error "
1373              << error << " is greater than epsilon " << epsilon << "\n";
1374      }
1375  #endif
1376
1377      return result ? CUTTrue : CUTFalse;
1378  }
```

$$normRef = \| reference \|$$

$$normError = \| reference - data \|$$

$$error = \frac{\| reference - data \|}{\| reference \|}$$ is relative error

if $error > epsilon$, then report error message

Rounding error in GPU computation [3]

We copy the content of “cutCompareL2fe” to another function “compare_L2err” but report more information than “cutCompareL2fe”

matrixMul.cu

```
void compare_L2err( const float* reference, const float* data,
                const unsigned int len, const float epsilon )
{
    assert( epsilon >= 0);
    float error = 0;
    float ref = 0;

    for( unsigned int i = 0; i < len; ++i) {
        float diff = reference[i] - data[i];
        error += diff * diff;
        ref += reference[i] * reference[i];
    }

    float normRef = sqrtf(ref);
    if (fabs(ref) < 1e-7) {
        std::cerr << "ERROR, reference l2-norm is 0 (machine zero)\n";
    }
    float normError = sqrtf(error);
    error = normError / normRef;
    bool result = error < epsilon;
        std::cout  << "L2(referece) = " << normRef << endl ;
        std::cout  << "L2(reference - data) = " << normError << endl ;
    if( ! result) {
        std::cerr << "ERROR, relative l2-norm error "
            << error << " is greater than epsilon " << epsilon << "\n";
    }else{
        std::cout << "relative L2-norm error " << error << " is O.K." << endl ;
    }


}
```

matrixMul.cu

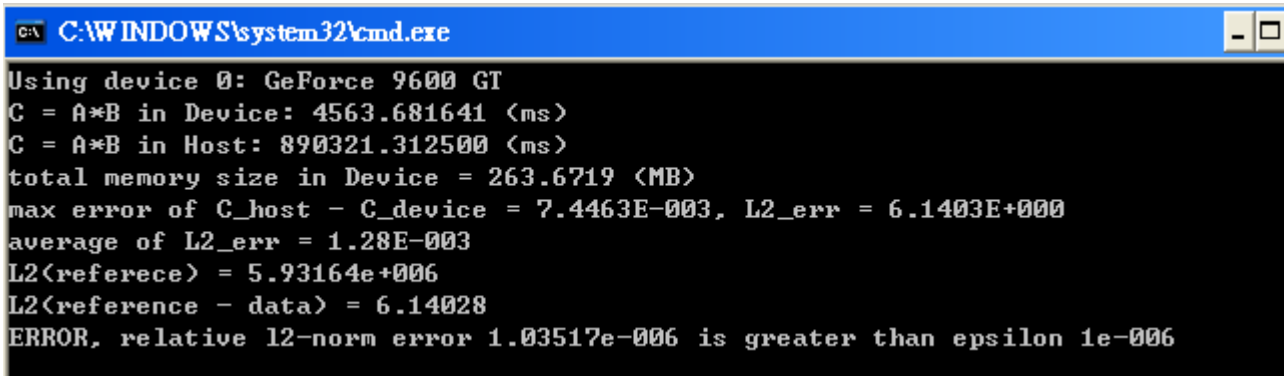
```
void compare_supnorm( float* A, float *B, unsigned int size,
      float* max_err, float* L2_err )
{
    unsigned int i ;
    float err ;

    *max_err = 0.0 ;
    *L2_err = 0.0 ;
    for( i = 0 ; i < size ; i++){
       err = fabs( A[i] - B[i] ) ;
       if ( err > *max_err ){
          *max_err = err ;
       }
       *L2_err += err*err ;
    }
    *L2_err = sqrt( *L2_err ) ;
}
```

find $\| A-B \|_\infty$ and $\| A-B \|_2$

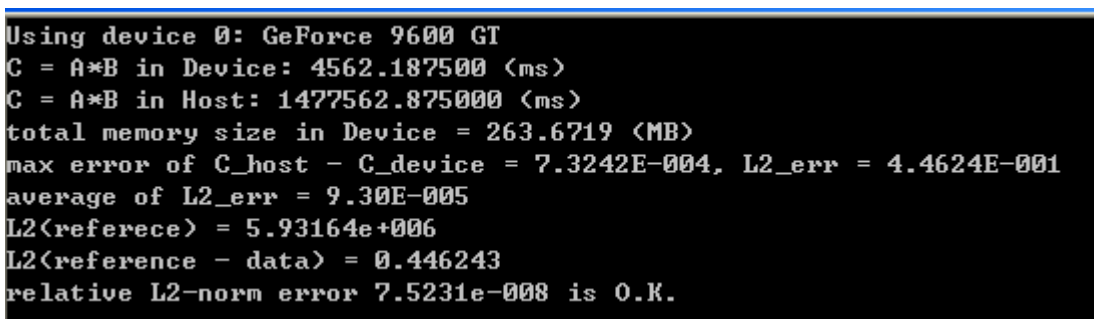
Rounding error in GPU computation [4]

$size(A) = size(B) = size(C) = (300 \times 16)^2$

Compare *matrixMul_block_seq* with *matrixMul* in GPU



Compare *matrixMul_seq* (non-block version) with *matrixMul* in GPU



To sum up, one can measure relative error but disable function printDiff

Rounding error in GPU computation [5]

matrixMul.cu

```
void computeGold(float* C, const float* A, const float* B,
            unsigned int hA, unsigned int wA, unsigned int wB)
{
    float sum, a, b ;

    for (unsigned int i = 0; i < hA; ++i)
        for (unsigned int j = 0; j < wB; ++j) {
            sum = 0.0 ;
            for (unsigned int k = 0; k < wA; ++k) {
                a = A[i * wA + k];
                b = B[k * wB + j];
                sum += a * b;
            }
            C[i * wB + j] = sum;
        }
}
```

matrixMul_kernel.cu

```
// Loop over all the sub-matrices of A and B
// required to compute the block sub-matrix
for (int a = aBegin, b = bBegin;
         a <= aEnd;
         a += aStep, b += bStep) {

    // Declaration of the shared memory array As used to
    // store the sub-matrix of A
    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];

    // Declaration of the shared memory array Bs used to
    // store the sub-matrix of B
    __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

    // Load the matrices from device memory
    // to shared memory; each thread loads
    // one element of each matrix
    AS(ty, tx) = A[a + wA * ty + tx];
    BS(ty, tx) = B[b + wB * ty + tx];

    // Synchronize to make sure the matrices are loaded
    __syncthreads();

    // Multiply the two matrices together;
    // each thread computes one element
    // of the block sub-matrix
    for (int k = 0; k < BLOCK_SIZE; ++k){
        Csub += AS(ty, k) * BS(k, tx);
    }

    // Synchronize to make sure that the preceding
    // computation is done before loading two new
    // sub-matrices of A and B in the next iteration
    __syncthreads();
}
```

Question 3: floating point operation order of *matrixMul_seq* (CPU non-block version) is the same as *matrixMul* in GPU, Can we modify the program such that they execute the same result ?

Rounding error in GPU computation [6]

In page 81, NVIDIA_CUDE_Programming_Guide_2.0.pdf

Addition and multiplication are IEEE-compliant, so have a maximum error of 0.5 ulp. However, on the device, the compiler often combines them into a single multiply-add instruction (FMAD), which truncates the intermediate result of the multiplication. This combination can be avoided by using the `__fadd_rn()` and `__fmul_rn()` intrinsic functions (see Section B.2).

Table B-1. Mathematical Standard Library Functions with Maximum ULP Error

The maximum error is stated as the absolute value of the difference in ulps between a correctly rounded single-precision result and the result returned by the CUDA library function.

Function	Maximum ulp error
`x+y`	0 (IEEE-754 round-to-nearest-even) (except when merged into an FMAD)
`x*y`	0 (IEEE-754 round-to-nearest-even) (except when merged into an FMAD)

In page 86, NVIDIA_CUDE_Programming_Guide_2.0.pdf

`__fadd_rn()` and `__fmul_rn()` map to addition and multiplication operations that the compiler never merges into FMADs. By contrast, additions and multiplications generated from the '*' and '+' operators will frequently be combined into FMADs.

matrixMul_kernel.cu

```
// Multiply the two matrices together;
// each thread computes one element
// of the block sub-matrix
for (int k = 0; k < BLOCK_SIZE; ++k){
    Csub += AS(ty, k) * BS(k, tx);
}
```

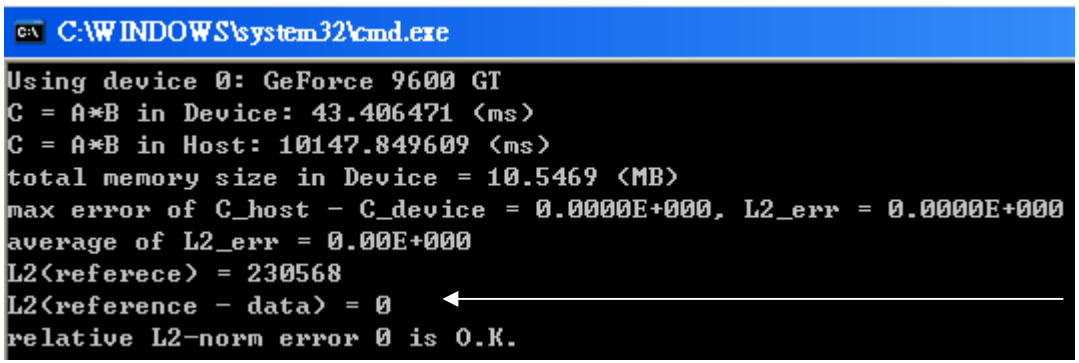
Inner-product based,
FMAD is done
automatically by compiler

Rounding error in GPU computation [7]

1 use __fmul_rn to avoid FMAD operation

```
        // Multiply the two matrices together;
        // each thread computes one element
        // of the block sub-matrix
        for (int k = 0; k < BLOCK_SIZE; ++k){
//          Csub += AS(ty, k) * BS(k, tx);
            Csub = Csub + __fmul_rn( AS(ty, k), BS(k, tx)) ;
        }
```

Functions suffixed with **_rn** operate using the round-to-nearest-even rounding mode.



No error !!!

2 use __fmul_rz to avoid FMAD operation

```
        // Multiply the two matrices together;
        // each thread computes one element
        // of the block sub-matrix
        for (int k = 0; k < BLOCK_SIZE; ++k){
//          Csub += AS(ty, k) * BS(k, tx);
            Csub = Csub + __fmul_rz( AS(ty, k), BS(k, tx)) ;
        }
```

Functions suffixed with **_rz** operate using the round-towards-zero rounding mode

Exercise 4: what is the result of above program?

Rounding error in GPU computation [8]

Exercise 5: in vc2005, we show that under *__fmul_rn*, CPU (non-block version) and GPU (block version) have the same result, check this in Linux machine.

Exercise 6: compare performance between FMAD and non-FMAD

FMAD

```
// Multiply the two matrices together;
// each thread computes one element
// of the block sub-matrix
for (int k = 0; k < BLOCK_SIZE; ++k){
    Csub += AS(ty, k) * BS(k, tx);
}
```

Non-FMAD

```
        // Multiply the two matrices together;
        // each thread computes one element
        // of the block sub-matrix
        for (int k = 0; k < BLOCK_SIZE; ++k){
//          Csub += AS(ty, k) * BS(k, tx);
            Csub = Csub + __fmul_rn( AS(ty, k), BS(k, tx)) ;
        }
```

Rounding mode

Reference: http://en.wikipedia.org/wiki/Floating_point

- Rounding modes are used when the exact result of a floating-point operation (or a conversion to floating-point format) would need more significant digits than there are digits in the significand. There are several different rounding schemes (or *rounding modes*)
  - *round to nearest* (the default): results are rounded to the nearest representable value. If the result is midway between two representable values, the even representable is chosen. *Even* here means the lowest-order bit is zero. This rounding mode prevents statistical bias and guarantees numeric stability: round-off errors in a lengthy calculation will remain smaller than half of FLT_EPSILON

  - *round up* (toward +∞; negative results round toward zero)
  - *round down* (toward −∞; negative results round away from zero)
  - *round toward zero* (sometimes called "chop" mode; it is similar to the common behavior of float-to-integer conversions, which convert −3.9 to −3)

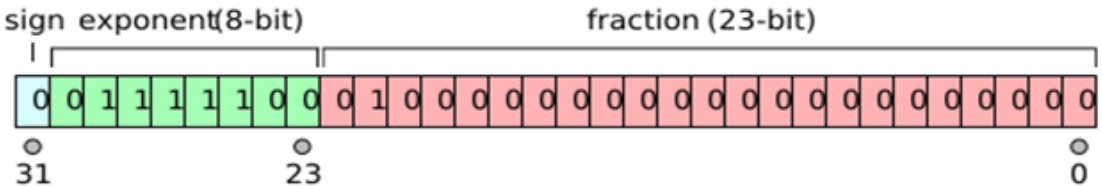
- the default method (*round to nearest, ties to even*, sometimes called Banker's Rounding) is more commonly used since the introduction of IEEE 754

See http://en.wikipedia.org/wiki/Rounding for detailed description

IEEE 754: standard for binary floating-point arithmetic

- Single-precision (32-bit)
- Double-precision (64-bit)
- Double-extended precision (80-bit), used in Intel CPU

$$v = s \times 2^E \times m$$



1. $s = \pm : sign\ bit$
2. $E = \exp - N$ (Excess-N biased)
3. $m = 1.b_1 b_2 \cdots$ normalized and $m = 0.b_1 b_2 \cdots$ denormalized

single precision

8 bit excess-127

Binary value	Excess-127 interpretation	Unsigned interpretation
00000000	-127	0
00000001	-126	1
...	...	...
01111111	0	127
10000000	+1	128
...	...	...
11111111	+128	255

$$F = \left\{ \pm 1.b_1 b_2 \cdots b_{23} \times 2^E \mid b_k \in \{0,1\},\ -127 \le E \le 128 \right\} \cup \{0\}$$

is a collection of all rational number (floating point) represented in computer

$$m \equiv \min |F| = 1.000\cdots 0 \times 2^{-127} \qquad M \equiv \max |F| = 1.111\cdots 1 \times 2^{128}$$

$$G = \left\{ x \in R \mid m \le |x| \le M \right\} \cup \{0\}$$

Define floating point operation $fl : G \to F$ by

$fl(x) =$ the nearest $c \in F$ to x by rounding arithmetic

Accumulation of rounding error [1]

Example: $x = 1.b_1 b_2 \cdots b_{22} 0 b_{24} b_{25} \cdots \times 2^E$ then $x \in G - F$

Suppose floating operation arithmetic is $fl(x) = 1.b_1 b_2 \cdots b_{22} \begin{cases} 1 & \text{if } b_{24} = 1 \\ 0 & \text{if } b_{24} = 0 \end{cases} \times 2^E$

then $\left| fl(x) - x \right| \le 2^{-24} \approx 5.96 \times 10^{-8} \equiv eps$ where $eps = machine\ zero$

Property: if $a, b, a \circ b \in G$, then $fl(a \circ b) = (a \circ b)(1 + \varepsilon)$

where $|\varepsilon| \le eps$ and $\circ = +, -, \times, /$

Theorem: consider algorithm for computing inner-product $x^T y$

$s := 0$

for $k = 1:1:n$

$s = s + x_k y_k$

endfor

then $fl(x^T y) = \sum_{k=1}^{n} x_k y_k (1 + \gamma_k)$ where $1 + \gamma_k = (1 + \delta_k) \prod_{j=k}^{n} (1 + \varepsilon_j)$ and $\varepsilon_1 = 0$

$|\delta_k| \le eps, \quad |\varepsilon_j| \le eps$

Accumulation of rounding error [2]

< Proof of Theorem > define partial sum $s_k = fl\left(\sum_{j=1}^{k} x_j y_j\right)$

$s_1 = fl(x_1 y_1) = x_1 y_1 (1+\delta_1)$ where $|\delta_1| \le eps$

$$s_2 = fl\left(s_1 + fl(x_2 y_2)\right) = fl\left(s_1 + x_2 y_2 (1+\delta_2)\right) = \left(s_1 + x_2 y_2 (1+\delta_2)\right)(1+\varepsilon_2)$$
$$= x_1 y_1 (1+\delta_1)(1+\varepsilon_2) + x_2 y_2 (1+\delta_2)(1+\varepsilon_2)$$
$$= x_1 y_1 (1+\delta_1)(1+\varepsilon_1)(1+\varepsilon_2) + x_2 y_2 (1+\delta_2)(1+\varepsilon_2) \quad \text{if} \quad \varepsilon_1 = 0$$

$$s_3 = fl\left(s_2 + fl(x_3 y_3)\right) = fl\left(s_2 + x_3 y_3 (1+\delta_3)\right) = \left(s_2 + x_3 y_3 (1+\delta_3)\right)(1+\varepsilon_3)$$
$$= x_1 y_1 (1+\delta_1)\underbrace{(1+\varepsilon_1)(1+\varepsilon_2)(1+\varepsilon_3)} + x_2 y_2 (1+\delta_2)\underbrace{(1+\varepsilon_2)(1+\varepsilon_3)} + x_3 y_3 (1+\delta_3)(1+\varepsilon_3)$$

propagation of rounding error

Inductively

$$s_3 = x_1 y_1 (1+\delta_1)(1+\varepsilon_1)(1+\varepsilon_2)\cdots(1+\varepsilon_k) + x_2 y_2 (1+\delta_2)(1+\varepsilon_2)\cdots(1+\varepsilon_k) + \cdots + x_k y_k (1+\delta_k)(1+\varepsilon_k)$$

Question 4: what is relative rounding error, say $1 - \dfrac{fl\left(x^T y\right)}{x^T y}$

Could you interpret result of previous example of rounding error?

OutLine

- Store (a+b) != (a+b)
- Rounding error in GPU
- Qt4 + vc2005
- Performance evaluation: GPU v.s. CPU

OpenMP + QT4 + vc2005: method 1 [1]

Step 1: open command prompt: issue command “qmake -v” to check version of qmake

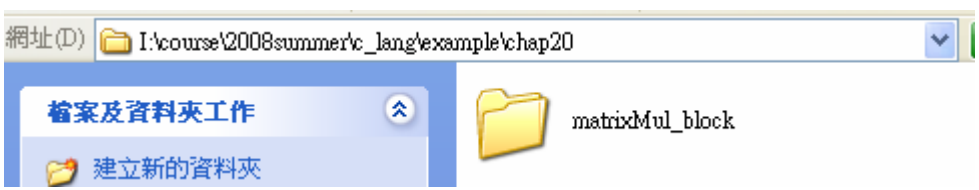
```
C:\Documents and Settings\lschien>qmake -v
QMake version 2.01a
Using Qt version 4.4.3 in C:\Qt\4.4.3\lib
```

Step 2: type “set” to show environment variables, check if
QTDIR=C:\Qt\4.4.3 (or QTDIR=C:\Qt\4.4.0) and
QMAKESPEC=win32-msvc2005

```
C:\Documents and Settings\lschien>set
ALLUSERSPROFILE=C:\Documents and Settings\All Users
APPDATA=C:\Documents and Settings\lschien\Application Data
CC=cl
CNL_COMPILER_VERSION=Microsoft (R) C/C++ Optimizing Compiler Version 14.00.40310
.41 for AMD64
CNL_DIR=C:\Program Files (x86)\VNI\imsl\cnl600
CNL_EXAMPLES=C:\Program Files (x86)\VNI\imsl\cnl600\ms64pc\examples
CNL_VERSION=6.0.0
```

```
QMAKESPEC=win32-msvc2005
QTDIR=C:\Qt\4.4.3
```

Step 3: prepare source code



OpenMP + QT4 + vc2005: method 1 [2]

```
I:\course\2008summer\c_lang\example\chap20\matrixMul_block>
I:\course\2008summer\c_lang\example\chap20\matrixMul_block>dir
 磁碟區 I 中的磁碟是 code
 磁碟區序號:  484D-55C8

 I:\course\2008summer\c_lang\example\chap20\matrixMul_block 的目錄

2008/12/25  下午 09:59    <DIR>          .
2008/12/25  下午 09:59    <DIR>          ..
2008/12/24  上午 09:13            10,977 global.h
2008/12/25  下午 08:17             3,103 main.cpp
2008/12/24  上午 12:54             1,801 matrixMul.cpp
2008/12/25  下午 08:15             2,355 matrixMul.h
2008/12/24  上午 09:12             5,415 matrixMul_block.cpp
               5 個檔案          23,651 位元組
               2 個目錄  36,218,388,480 位元組可用
```

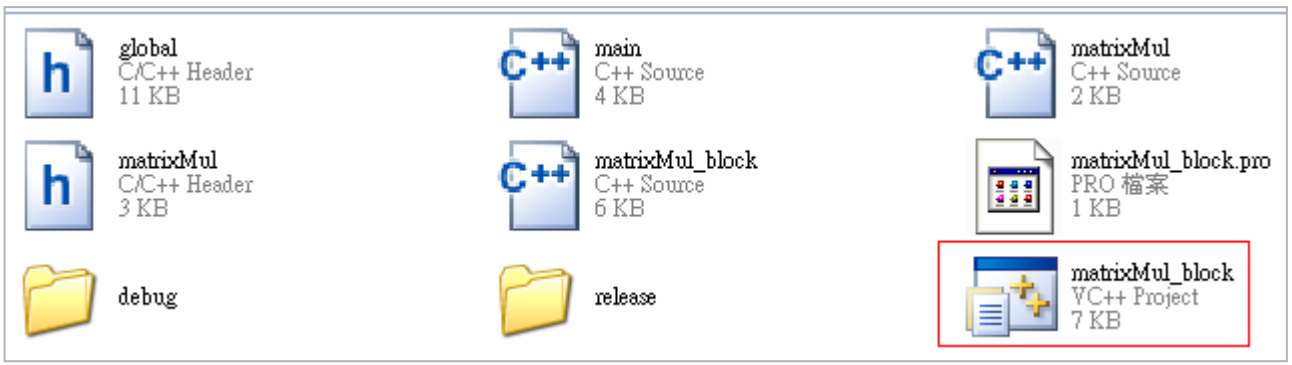
Step 4: type “qmake -project” to generate project file matrixMul_block.pro

```
I:\course\2008summer\c_lang\example\chap20\matrixMul_block>qmake -project

I:\course\2008summer\c_lang\example\chap20\matrixMul_block>dir
 磁碟區 I 中的磁碟是 code
 磁碟區序號:  484D-55C8

 I:\course\2008summer\c_lang\example\chap20\matrixMul_block 的目錄

2008/12/25  下午 10:02    <DIR>          .
2008/12/25  下午 10:02    <DIR>          ..
2008/12/24  上午 09:13            10,977 global.h
2008/12/25  下午 08:17             3,103 main.cpp
2008/12/24  上午 12:54             1,801 matrixMul.cpp
2008/12/25  下午 08:15             2,355 matrixMul.h
2008/12/24  上午 09:12             5,415 matrixMul_block.cpp
2008/12/25  下午 10:03               376 matrixMul_block.pro
               6 個檔案          24,027 位元組
               2 個目錄  36,218,327,040 位元組可用
```

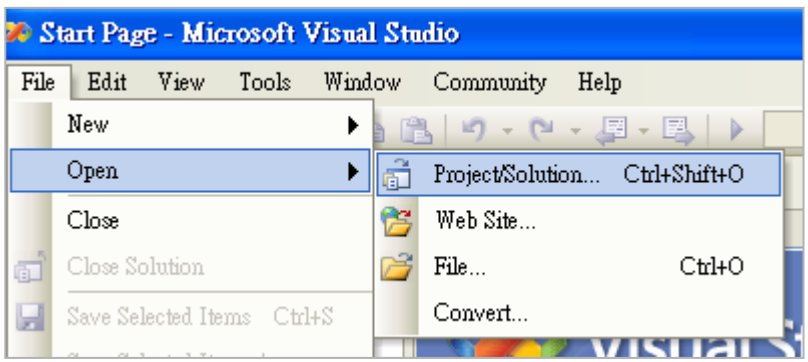
OpenMP + QT4 + vc2005: method 1 [3]

Step 5: type “qmake –tp vc matrixMul_block.pro” to generate VC project file

```
I:\course\2008summer\c_lang\example\chap20\matrixMul_block>qmake -tp vc matrixMu
l_block.pro
```

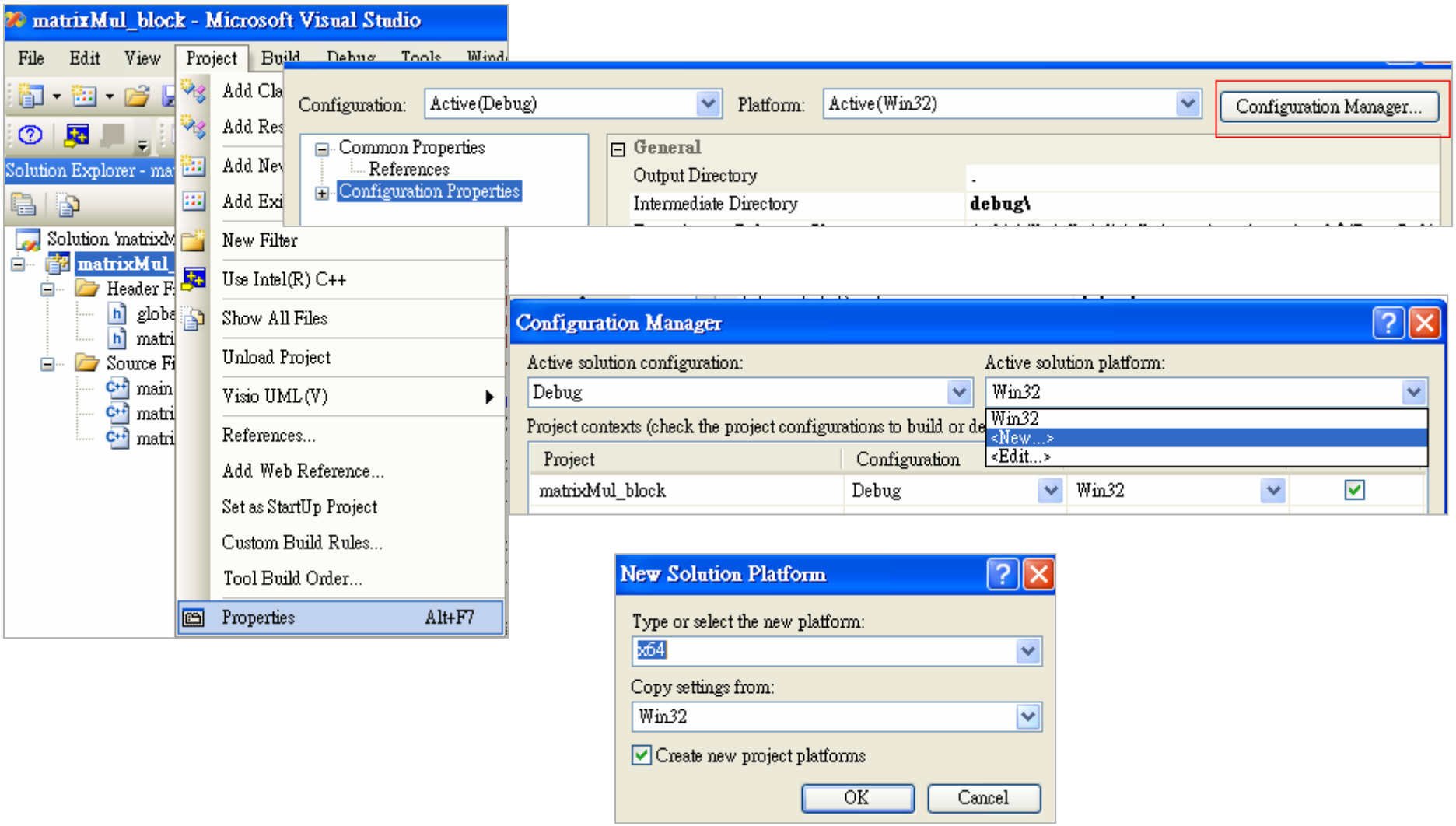


Step 6: activate vc2005 and use File→ Open → Project/Solution to open above VC project



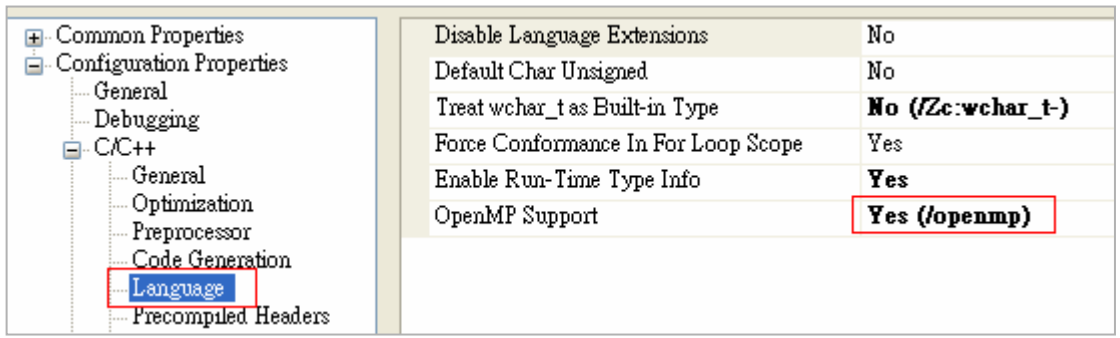
OpenMP + QT4 + vc2005: method 1 [4]

Step 7: Project → Properties → configuration Manager: change platform to x64 (64-bit application)

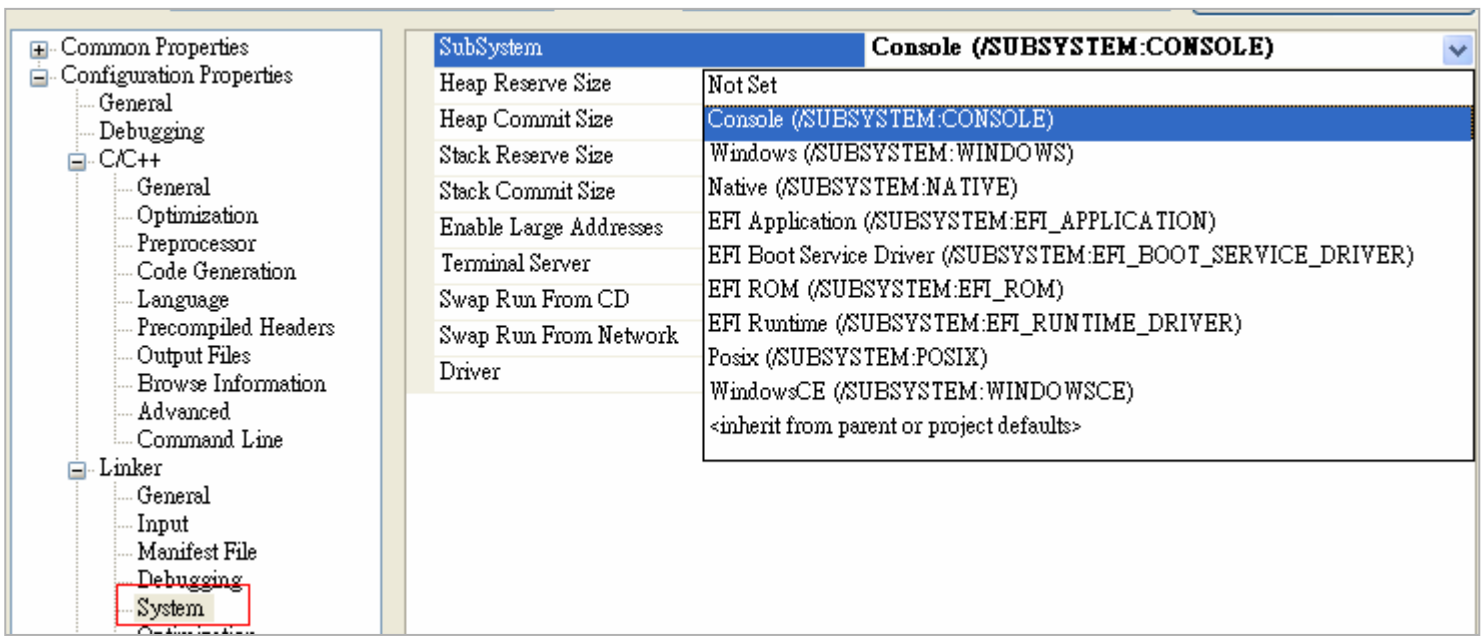


OpenMP + QT4 + vc2005: method 1 [5]

Step 8: Language → OpenMP Support : turn on

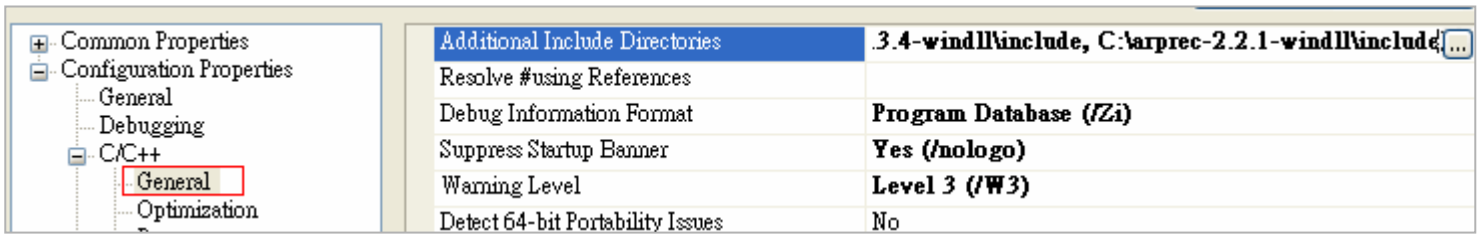


Step 9: Linker → System → SubSystem → change to “console” (this will disable GUI window since we don’t use GUI module)



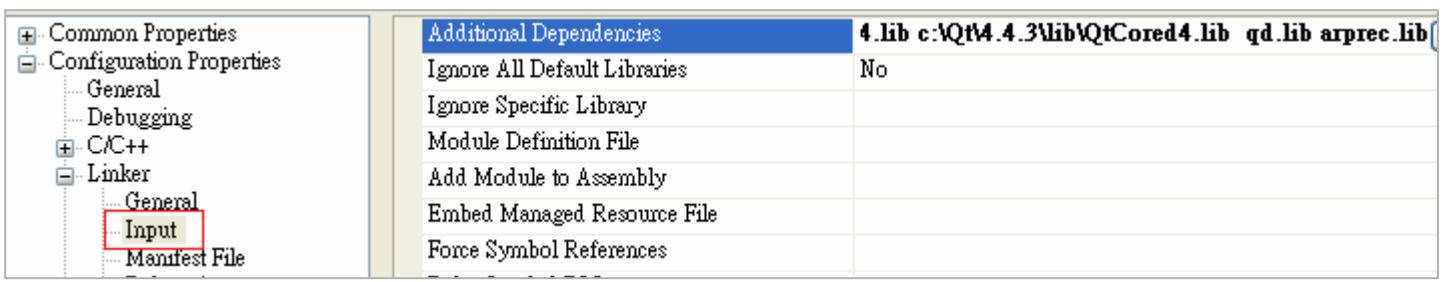
OpenMP + QT4 + vc2005: method 1 [6]

Step 10: C/C++ → General → Additional Include Directories: add include files for high precision package, “C:\qd-2.3.4-windll\include, C:\arprec-2.2.1-windll\include”



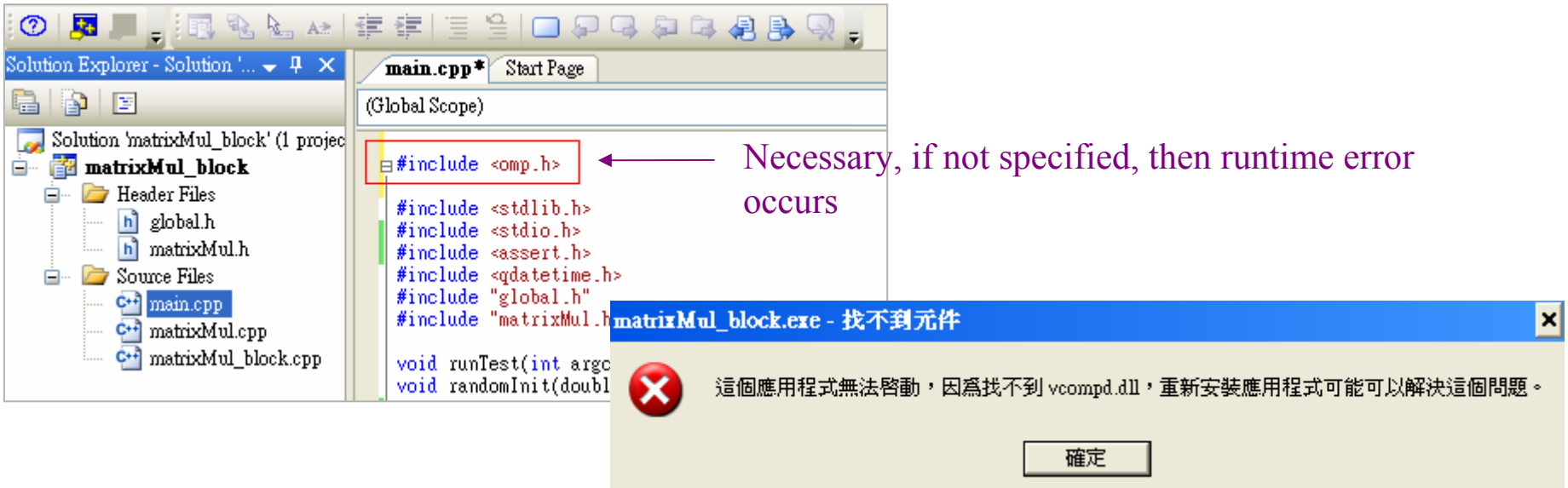
Step 11: Linker → General → Additional Library Directories: add library path for high precision package, “C:\qd-2.3.4-windll\x64\Debug, C:\arprec-2.2.1-windll\x64\Debug”

Step 12: Linker → Input → Additional Dependence: add library filesfor high precision package, “qd.lib arprec.lib”

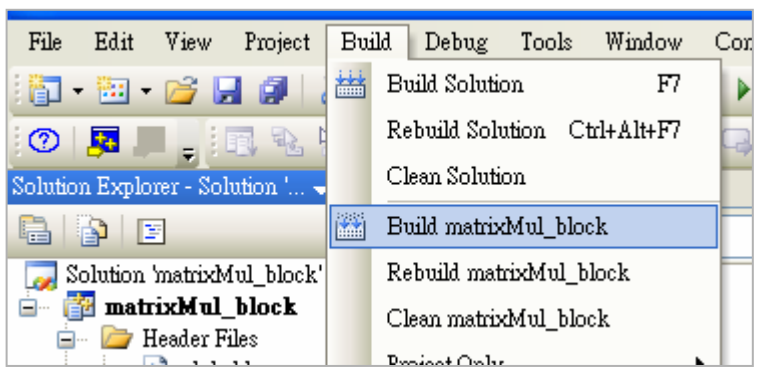


OpenMP + QT4 + vc2005: method 1 [7]

Step 13: check if header file “omp.h” is included in main.cpp (where function ***main*** locates)

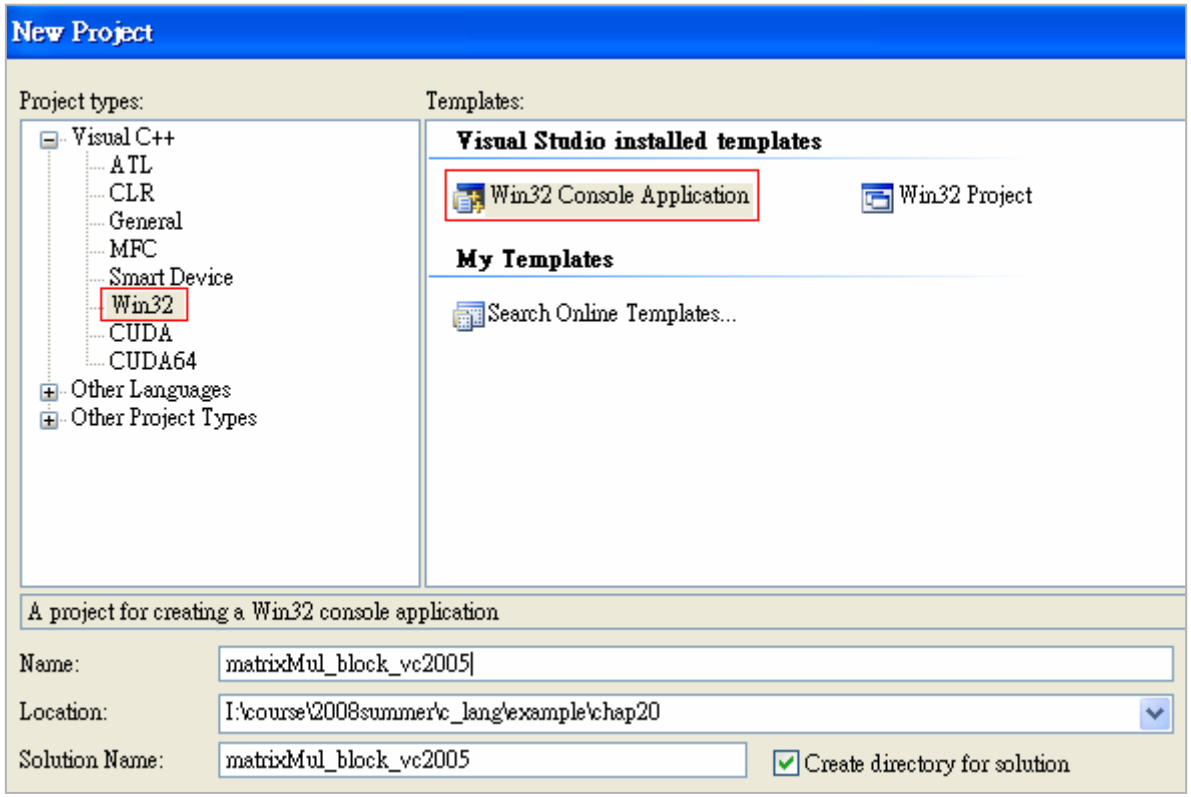


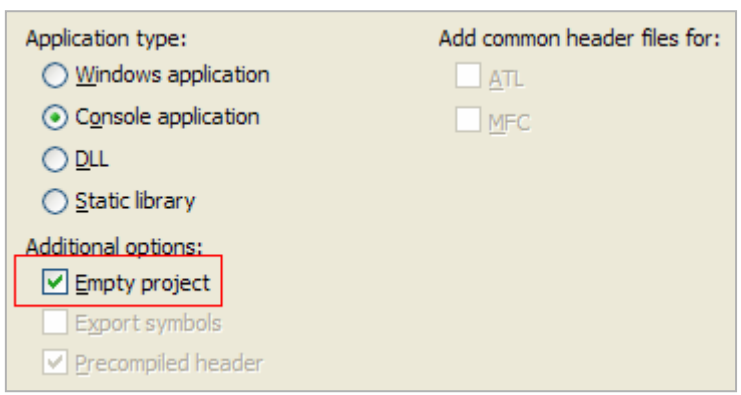
Step 14: build matrixMul_block and then execute

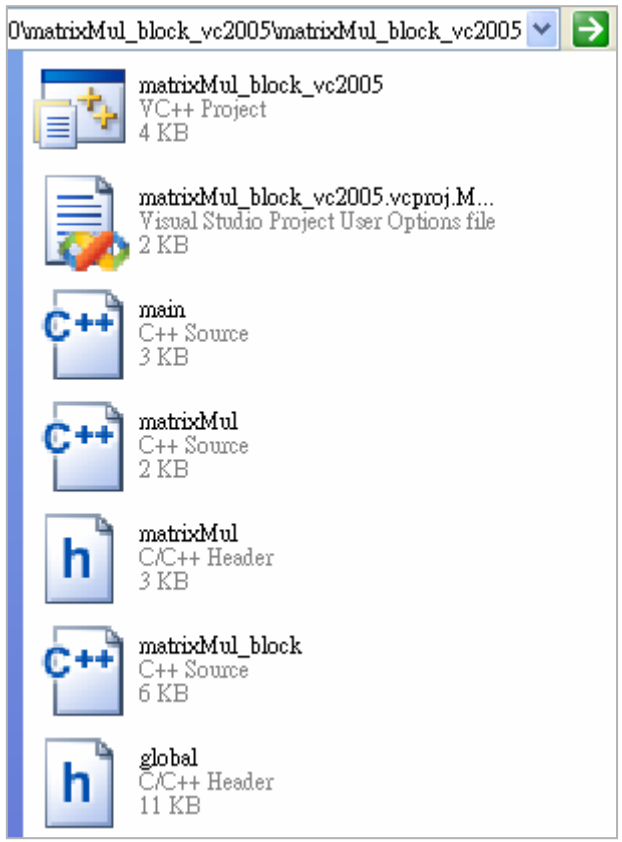


OpenMP + QT4 + vc2005: method 2 [1]

Step 1: create an empty project and copy source files to this project

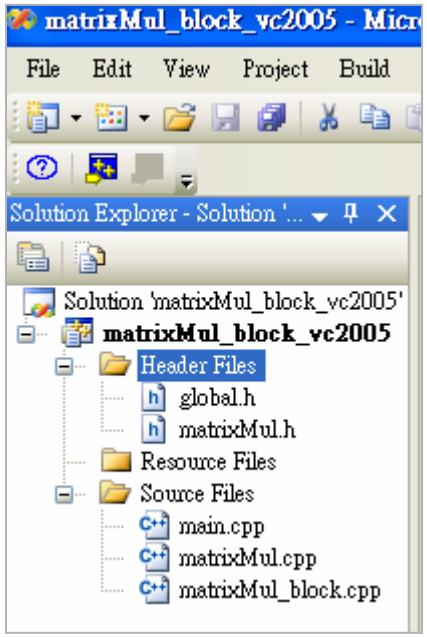


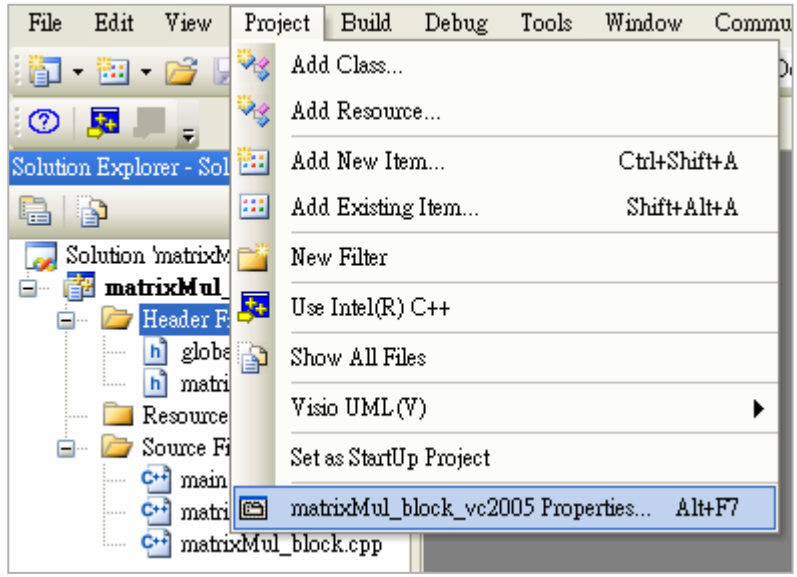




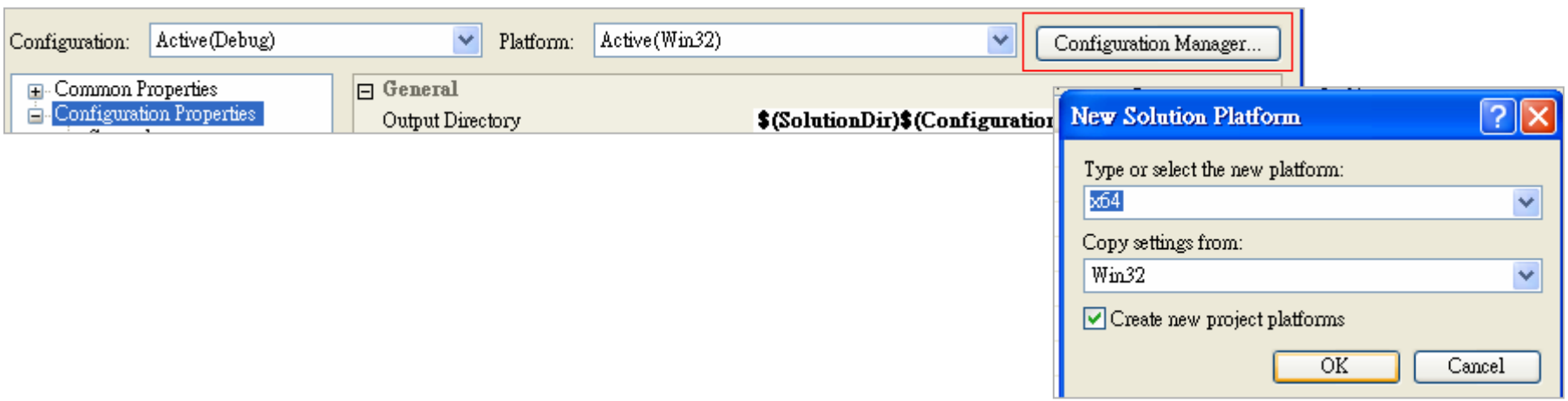
OpenMP + QT4 + vc2005: method 2 [2]

Step 2: add source files in project manager



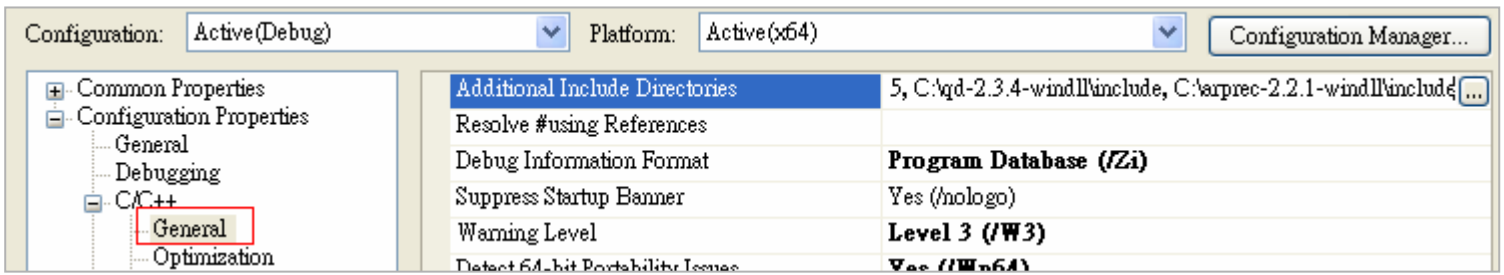


Step 3: Project → Properties → Monfiguration Manager: change platform to x64

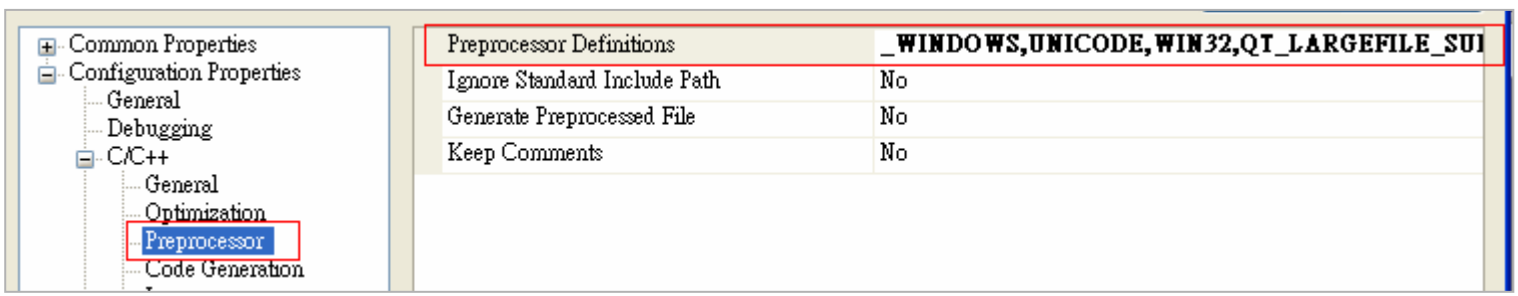


OpenMP + QT4 + vc2005: method 2 [3]

Step 4: C/C++ → General --> Additional Include Directories
"c:\Qt\4.4.3\include\QtCore","c:\Qt\4.4.3\include\QtCore","c:\Qt\4.4.3\include\QtGui","c:\Qt\4.4.3\include\QtGui","c:\Qt\4.4.3\include",".","c:\Qt\4.4.3\include\ActiveQt","debug",".",c:\Qt\4.4.3\mkspecs\win32-msvc2005, C:\qd-2.3.4-windll\include, C:\arprec-2.2.1-windll\include

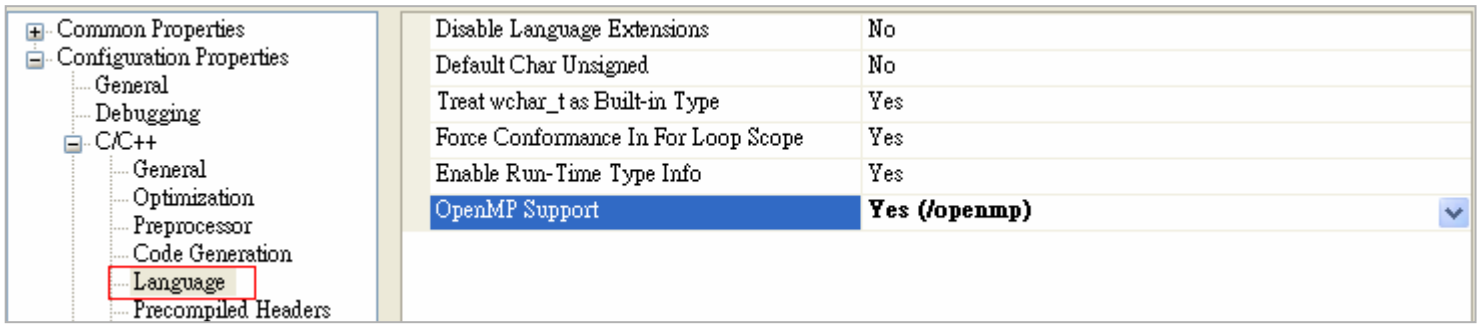


Step 5: C/C++ → Preprocessor --> Preprocessor Definitions
_WINDOWS,UNICODE,WIN32,QT_LARGEFILE_SUPPORT,QT_DLL,QT_GUI_LIB,QT_CORE_LIB,QT_THREAD_SUPPORT

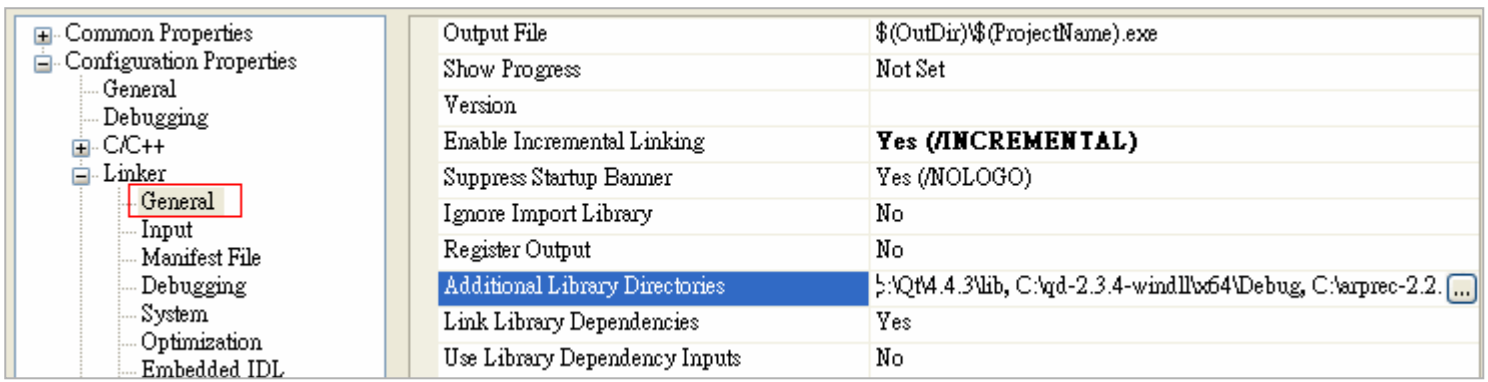


OpenMP + QT4 + vc2005: method 2 [4]

Step 6: Language → OpenMP Support : turn on

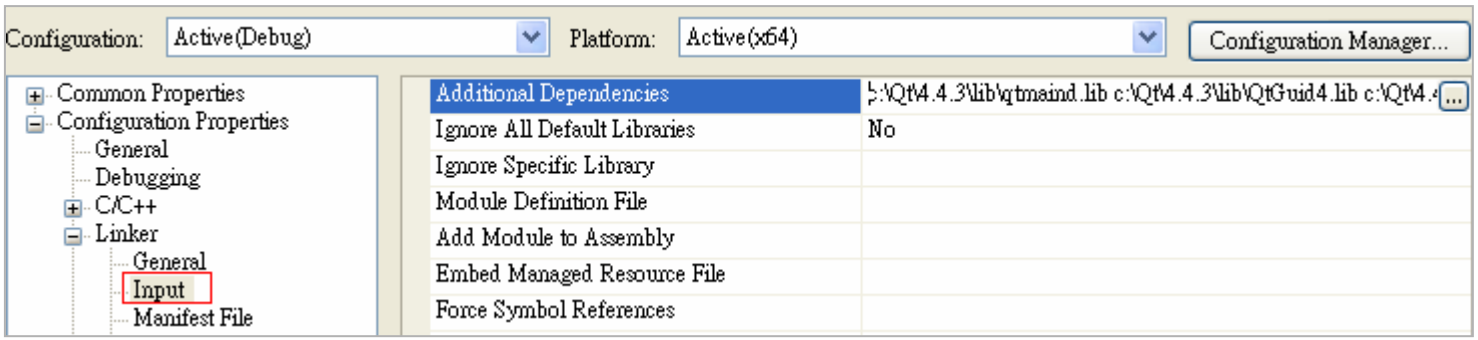


Step 7: Linker → General → Additional Library Directories
c:\Qt\4.4.3\lib, C:\qd-2.3.4-windll\x64\Debug, C:\arprec-2.2.1-windll\x64\Debug

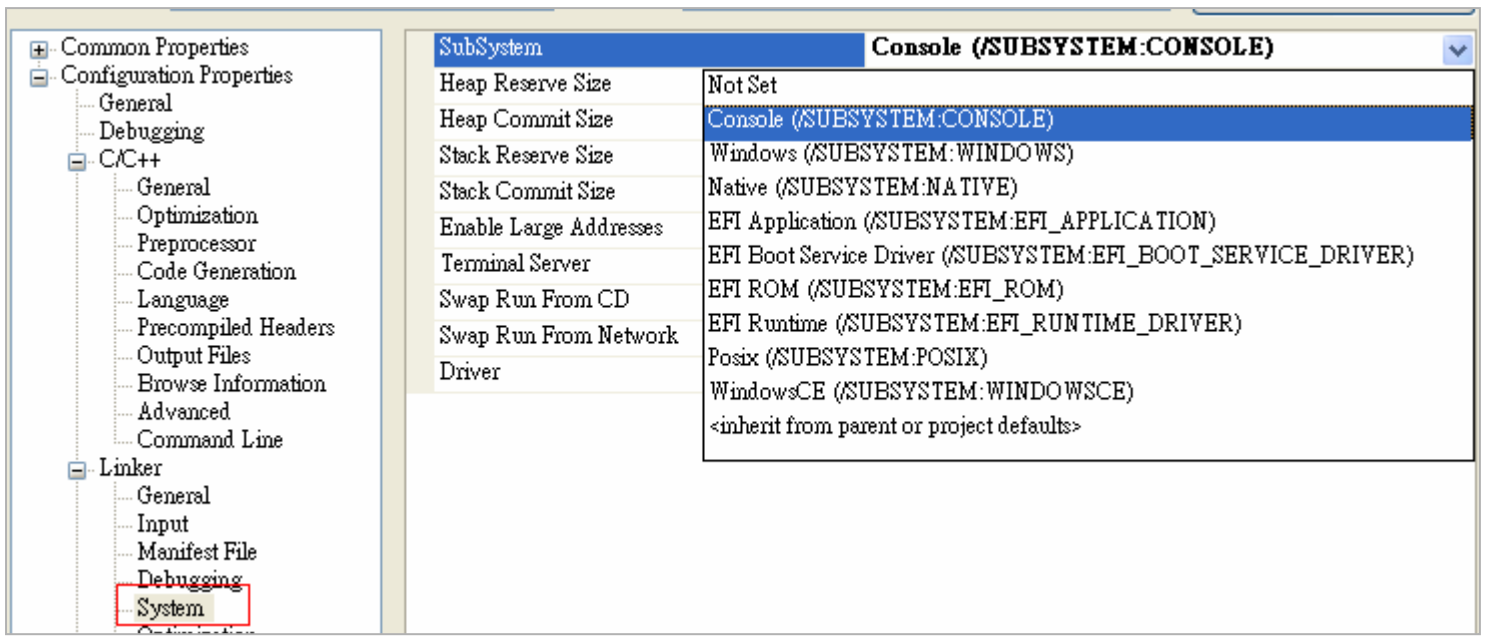


OpenMP + QT4 + vc2005: method 2 [5]

Step 8: Linker → Input → Additional Dependences
c:\Qt\4.4.3\lib\qtmaind.lib c:\Qt\4.4.3\lib\QtGuid4.lib c:\Qt\4.4.3\lib\QtCored4.lib qd.lib arprec.lib

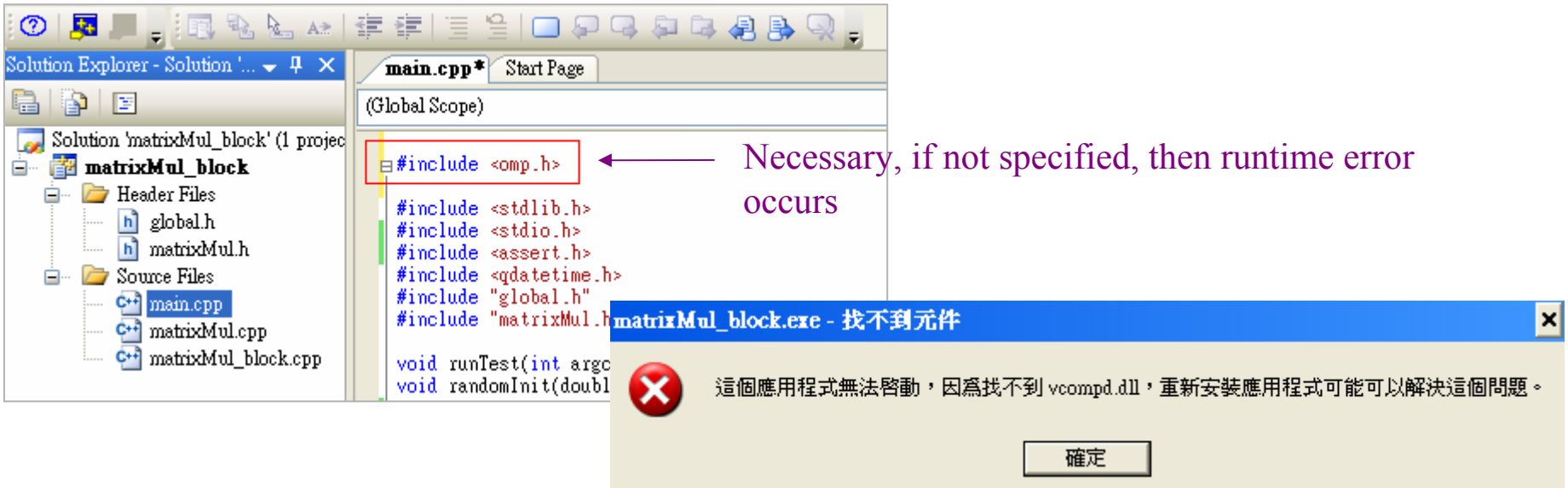


Step 9: Linker → System → SubSystem → change to “console”

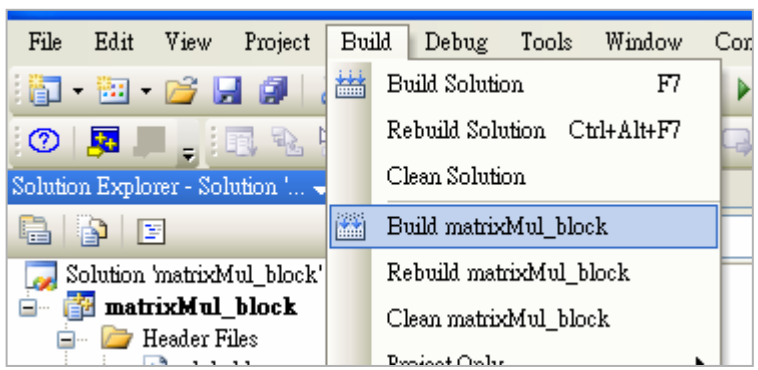


OpenMP + QT4 + vc2005: method 2 [6]

Step 10: check if header file “omp.h” is included in main.cpp (where function ***main*** locates)



Step 11: build matrixMul_block and then execute



OutLine

- Store (a+b) != (a+b)
- Rounding error in GPU
- Qt4 + vc2005
- Performance evaluation: GPU v.s. CPU

GPU versus CPU: matrix multiplication [1]

GPU: FMAD

```
// Multiply the two matrices together;
// each thread computes one element
// of the block sub-matrix
for (int k = 0; k < BLOCK_SIZE; ++k){
    Csub += AS(ty, k) * BS(k, tx);
}
```

CPU: block version

```
60          c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
61            // Multiply the two matrices together
62              for ( ty = 0 ; ty < BLOCK_SIZE ; ty++ ){
63                for ( tx = 0 ; tx < BLOCK_SIZE ; tx++ ){
64                  Csub = 0.0 ;
65                  for (k = 0; k < BLOCK_SIZE; ++k ){
66                      Asub = As[ty][k ] ;
67                      Bsub = Bs[k ][tx] ;
68                      Csub += Asub * Bsub ;
69                  }
70                C[c + wB * ty + tx] += Csub;
71                  }// for tx ;
72                }// for ty
```

Makefile

```
nvcc_run:
        nvcc -run -02 $(INLCUDE) $(LIBS) $(SRC_CU)   $(SRC_CXX)
```

nvcc uses g++ as default compiler

```
[macrold@matrix matrixMul_cuda]$ gcc -v
Using built-in specs.
Target: x86_64-redhat-linux
Configured with: ../configure --prefix=/usr --mandir=/usr/share/man --infodir=/usr/sh
are/info --enable-shared --enable-threads=posix --enable-checking=release --with-syst
em-zlib --enable-__cxa_atexit --disable-libunwind-exceptions --enable-languages=c,c++
,objc,obj-c++,java,fortran,ada --enable-java-awt=gtk --disable-dssi --enable-plugin -
-with-java-home=/usr/lib/jvm/java-1.5.0-gcj-1.5.0.0/jre --enable-libgcj-multifile --e
nable-java-maintainer-mode --with-ecj-jar=/usr/share/java/eclipse-ecj.jar --with-cpu=
generic --host=x86_64-redhat-linux
Thread model: posix
gcc version 4.1.2 20070925 (Red Hat 4.1.2-33)
[macrold@matrix matrixMul_cuda]$
```

GPU versus CPU: matrix multiplication [2]

$BLOCK\_SIZE = 16$ $\quad size(A) = size(B) = size(C) = (N \times 16)^2$

Experimental platform: matrix (quad-core, Q6600), Geforce 9600 GT

$total\ size = size(A) + size(B) + size(C)$

h2d: time for host to device (copy h_A → d_A and h_B → d_B)

GPU: time for C = A* B in GPU

d2h: time for device to host (copy d_C → h_C)

CPU: time for C = A* B in CPU, sequential block version

N	Total size MB	(1) h2d ms	(2) GPU ms	(3) d2h ms	(4) = (1)+(2)+(3)	(5) CPU ms	(5)/(4)
16	0.75	0.39	0.75	0.4	1.54	31.04	20.2
32	3	1.37	5.49	1.47	8.34	224.64	26.9
64	12	4.81	42.87	4.54	52.21	1967.2	37.6
128	48	16.65	345.24	17.5	379.39	17772.87	46.8
256	192	66.13	2908.76	69.64	3044.53	146281	48
350	358.9	123.83	7174.93	130.1	7428.88	314599.69	42.3
397	461.7	158.29	10492.08	166.73	10817.1	468978.09	43.4

GPU versus CPU: matrix multiplication [3]

Let BLOCK_SIZE = 16 and $size(A) = size(B) = size(C) = (N \cdot BLOCK\_SIZE)^2$

total memory usage $= size(A) + size(B) + size(C)$ float

Platform: matrix, with compiler icpc 10.0, -O2

N	Total size	Thread 1	Thread 2	Thread 4	Thread 8
16	0.75 MB	13 ms	10 ms	10 ms	139 ms
32	3 MB	371 ms	271 ms	185 ms	616 ms
64	12 MB	3,172 ms	2,091 ms	1,435 ms	3,854 ms
128	48 MB	27,041 ms	17,373 ms	13,357 ms	29,663 ms
256	192 MB	220,119 ms	143,366 ms	101,031 ms	235,353 ms

N	Total size (MB)	(1) GPU (transfer + computation)	(2) CPU (block version), 4 threads	(2)/(1)
16	0.75	1.54	10	6.5
32	3	8.34	185	22.2
64	12	52.21	1435	27.5
128	48	379.39	13357	35.2
256	192	3044.53	101031	33.2

GPU versus CPU: matrix multiplication [4]

Platform: matrix, with compiler icpc 10.0, -O2

Block version, BLOCK_SIZE = 512

N	Total size	Thread 1	Thread 2	Thread 4	Thread 8
2	12 MB	4,226 ms	2,144 ms	1,074 ms	1,456 ms
4	48 MB	33,736 ms	17,095 ms	8,512 ms	9,052 ms
8	192 MB	269,376 ms	136,664 ms	68,026 ms	70,823 ms

N	Total size (MB)	(1) GPU (transfer + computation)	N	(2) CPU (block version), 4 threads	(2)/(1)
64	12	52.21	2	1074	20.6
128	48	379.39	4	8512	22.4
256	192	3044.53	8	68026	22.3

Geforce 9600GT

GPU Engine Specs:

Processor Cores	64
Graphics Clock (MHz)	650 MHz
Processor Clock (MHz)	1625 MHz
Texture Fill Rate (billion/sec)	20.8

Question 4: GPU is 20 times faster then CPU, is this number reasonable ?