M.S. RAMAIAH INSTITUTE OF TECHNOLOGY, BANGALORE – 560 054.

HPC LAB Manual

VII SEM

CSE

Prepared by,

Mallegowda M

1. Hello World 1

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

int main()
{
#pragma omp parallel
printf("Hello, world! This is thread %d of %d\n", omp_get_thread_num(), omp_get_num_threads());
}
```

Output

```
Hello, world! This is thread 2 of 4
```

Hello, world! This is thread 0 of 4

Hello, world! This is thread 3 of 4

Hello, world! This is thread 1 of 4

****

2. Hello World 2

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

int main(int argc, char *argv[])
{
int iam = 0, np = 1;

#pragma omp parallel default(shared) private(iam, np)
{
#if defined (_OPENMP)
np = omp_get_num_threads();
iam = omp_get_thread_num();
#endif
printf("Hello from thread %d out of %d\n", iam, np);
}

}
```

Output

Hello from thread 0 out of 4

Hello from thread 3 out of 4

Hello from thread 1 out of 4

Hello from thread 2 out of 4

****

3. Addition of two array A & B to get array C using scheduling concept

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define CHUNKSIZE 10
#define N 100
int main (int argc, char *argv[])
{
int nthreads, tid, i, chunk;
float a[N], b[N], c[N];

/* Some initializations */
for (i=0; i < N; i++)
a[i] = b[i] = i * 1.0;
```

```
chunk = CHUNKSIZE;

#pragma omp parallel shared(a,b,c,nthreads,chunk) private(i,tid)
{
tid = omp_get_thread_num();
if (tid == 0)
{
nthreads = omp_get_num_threads();
printf("Number of threads = %d\n", nthreads);
}
printf("Thread %d starting...\n",tid);

#pragma omp for schedule (dynamic, chunk)
for (i=0; i<N; i++)
{
c[i] = a[i] + b[i];
printf("Thread %d: c[%d]= %f\n",tid,i,c[i]);
}

} /* end of parallel section */

}
```

Output

Number of threads = 4

Thread 0 starting...

Thread 0: c[0]= 0.000000

Thread 0: c[1]= 2.000000

Thread 0: c[2]= 4.000000

Thread 0: c[3]= 6.000000

Thread 0: c[4]= 8.000000

Thread 0: c[5]= 10.000000

Thread 0: c[6]= 12.000000

Thread 0: c[7]= 14.000000

Thread 0: c[8]= 16.000000

Thread 0: c[9]= 18.000000

Thread 0: c[10]= 20.000000

Thread 2 starting...

Thread 2: c[20]= 40.000000

Thread 2: c[21]= 42.000000

Thread 2: c[22]= 44.000000

Thread 2: c[23]= 46.000000

Thread 2: c[24]= 48.000000

Thread 2: c[25]= 50.000000

Thread 2: c[26]= 52.000000

Thread 2: c[27]= 54.000000

Thread 0: c[11]= 22.000000

Thread 0: c[12]= 24.000000

Thread 0: c[13]= 26.000000

Thread 0: c[14]= 28.000000

Thread 0: c[15]= 30.000000

Thread 0: c[16]= 32.000000

Thread 0: c[17]= 34.000000

Thread 3 starting...

Thread 2: c[28]= 56.000000

Thread 2: c[29]= 58.000000

Thread 2: c[40]= 80.000000

Thread 2: c[41]= 82.000000

Thread 2: c[42]= 84.000000

Thread 2: c[43]= 86.000000

Thread 2: c[44]= 88.000000

Thread 2: c[45]= 90.000000

Thread 2: c[46]= 92.000000

Thread 2: c[47]= 94.000000

Thread 2: c[48]= 96.000000

Thread 2: c[49]= 98.000000

Thread 2: c[50]= 100.000000

Thread 2: c[51]= 102.000000

Thread 2: c[52]= 104.000000

Thread 2: c[53]= 106.000000

Thread 0: c[18]= 36.000000

Thread 1 starting...

Thread 1: c[60]= 120.000000

Thread 1: c[61]= 122.000000

Thread 1: c[62]= 124.000000

Thread 1: c[63]= 126.000000

Thread 1: c[64]= 128.000000

Thread 1: c[65]= 130.000000

Thread 1: c[66]= 132.000000

Thread 1: c[67]= 134.000000

Thread 1: c[68]= 136.000000

Thread 1: c[69]= 138.000000

Thread 1: c[70]= 140.000000

Thread 3: c[30]= 60.000000

Thread 3: c[31]= 62.000000

Thread 3: c[32]= 64.000000

Thread 2: c[54]= 108.000000

Thread 2: c[55]= 110.000000

Thread 2: c[56]= 112.000000

Thread 2: c[57]= 114.000000

Thread 2: c[58]= 116.000000

Thread 2: c[59]= 118.000000

Thread 2: c[80]= 160.000000

Thread 2: c[81]= 162.000000

Thread 2: c[82]= 164.000000

Thread 2: c[83]= 166.000000

Thread 2: c[84]= 168.000000

Thread 2: c[85]= 170.000000

Thread 2: c[86]= 172.000000

Thread 2: c[87]= 174.000000

Thread 2: c[88]= 176.000000

Thread 2: c[89]= 178.000000

Thread 2: c[90]= 180.000000

Thread 2: c[91]= 182.000000

Thread 2: c[92]= 184.000000

Thread 2: c[93]= 186.000000

Thread 2: c[94]= 188.000000

Thread 2: c[95]= 190.000000

Thread 2: c[96]= 192.000000

Thread 2: c[97]= 194.000000

Thread 2: c[98]= 196.000000

Thread 2: c[99]= 198.000000

Thread 3: c[33]= 66.000000

Thread 3: c[34]= 68.000000

Thread 3: c[35]= 70.000000

Thread 3: c[36]= 72.000000

Thread 3: c[37]= 74.000000

Thread 3: c[38]= 76.000000

Thread 3: c[39]= 78.000000

Thread 0: c[19]= 38.000000

Thread 1: c[71]= 142.000000

Thread 1: c[72]= 144.000000

Thread 1: c[73]= 146.000000

Thread 1: c[74]= 148.000000

Thread 1: c[75]= 150.000000

Thread 1: c[76]= 152.000000

Thread 1: c[77]= 154.000000

Thread 1: c[78]= 156.000000

Thread 1: c[79]= 158.000000

4. There are two arrays A and B write a program that has two blocks: one for generating array C = A+B and another array D = A+B, such that work in blocks will be done by different threads.

```
#include <omp.h>

#include <stdio.h>
```

```
#include <stdlib.h>

#define N 50

int main (int argc, char *argv[])

{

int i, nthreads, tid;

float a[N], b[N], c[N], d[N];



/* Some initializations */

for (i=0; i<N; i++) {

a[i] = i * 1.5;

b[i] = i + 22.35;

c[i] = d[i] = 0.0;

}



#pragma omp parallel shared (a,b,c,d,nthreads) private(i,tid)

{
```

```
tid = omp_get_thread_num();

if (tid == 0)

{

nthreads = omp_get_num_threads();

printf("Number of threads = %d\n", nthreads);

}

printf("Thread %d starting...\n",tid);



#pragma omp sections nowait

{

#pragma omp section

{

printf("Thread %d doing section 1\n",tid);

for (i=0; i<N; i++)

{

c[i] = a[i] + b[i];
```

```
printf("Thread %d: c[%d]= %f\n",tid,i,c[i]);

}

}



#pragma omp section

{

printf("Thread %d doing section 2\n",tid);

for (i=0; i<N; i++)

{

d[i] = a[i] * b[i];

printf("Thread %d: d[%d]= %f\n",tid,i,d[i]);

}



}



} /* end of sections */
```

```
printf("Thread %d done.\n",tid);



} /* end of parallel section */



}
```

Output

```
Number of threads = 4

Thread 0 starting...

Thread 0 doing section 1

Thread 0: c[0]= 22.350000

Thread 0: c[1]= 24.850000

Thread 0: c[2]= 27.350000

Thread 0: c[3]= 29.850000

Thread 0: c[4]= 32.349998

Thread 0: c[5]= 34.849998
```

Thread 0: c[6]= 37.349998

Thread 0: c[7]= 39.849998

Thread 0: c[8]= 42.349998

Thread 0: c[9]= 44.849998

Thread 3 starting...

Thread 3 doing section 2

Thread 3: d[0]= 0.000000

Thread 3: d[1]= 35.025002

Thread 3: d[2]= 73.050003

Thread 3: d[3]= 114.075005

Thread 3: d[4]= 158.100006

Thread 3: d[5]= 205.125000

Thread 3: d[6]= 255.150009

Thread 3: d[7]= 308.175018

Thread 3: d[8]= 364.200012

Thread 3: d[9]= 423.225006

Thread 3: d[10]= 485.249969

Thread 2 starting...

Thread 0: c[10]= 47.349998

Thread 1 starting...

Thread 1 done.

Thread 2 done.

Thread 0: c[11]= 49.849998

Thread 0: c[12]= 52.349998

Thread 0: c[13]= 54.849998

Thread 0: c[14]= 57.349998

Thread 0: c[15]= 59.849998

Thread 0: c[16]= 62.349998

Thread 0: c[17]= 64.849998

Thread 0: c[18]= 67.349998

Thread 0: c[19]= 69.849998

Thread 0: c[20]= 72.349998

Thread 0: c[21]= 74.849998

Thread 0: c[22]= 77.349998

Thread 0: c[23]= 79.849998

Thread 0: c[24]= 82.349998

Thread 3: d[11]= 550.274963

Thread 3: d[12]= 618.299988

Thread 3: d[13]= 689.324951

Thread 3: d[14]= 763.349976

Thread 0: c[25]= 84.849998

Thread 0: c[26]= 87.349998

Thread 0: c[27]= 89.849998

Thread 0: c[28]= 92.349998

Thread 0: c[29]= 94.849998

Thread 0: c[30]= 97.349998

Thread 0: c[31]= 99.849998

Thread 0: c[32]= 102.349998

Thread 0: c[33]= 104.849998

Thread 0: c[34]= 107.349998

Thread 0: c[35]= 109.849998

Thread 0: c[36]= 112.349998

Thread 0: c[37]= 114.849998

Thread 0: c[38]= 117.349998

Thread 0: c[39]= 119.849998

Thread 0: c[40]= 122.349998

Thread 0: c[41]= 124.849998

Thread 0: c[42]= 127.349998

Thread 0: c[43]= 129.850006

Thread 0: c[44]= 132.350006

Thread 0: c[45]= 134.850006

Thread 0: c[46]= 137.350006

Thread 0: c[47]= 139.850006

Thread 0: c[48]= 142.350006

Thread 0: c[49]= 144.850006

Thread 3: d[15]= 840.374939

Thread 3: d[16]= 920.399963

Thread 3: d[17]= 1003.424988

Thread 3: d[18]= 1089.449951

Thread 3: d[19]= 1178.474976

Thread 3: d[20]= 1270.500000

Thread 3: d[21]= 1365.524902

Thread 3: d[22]= 1463.549927

Thread 3: d[23]= 1564.574951

Thread 3: d[24]= 1668.599976

Thread 3: d[25]= 1775.625000

Thread 0 done.

Thread 3: d[26]= 1885.649902

Thread 3: d[27]= 1998.674927

Thread 3: d[28]= 2114.699951

Thread 3: d[29]= 2233.724854

Thread 3: d[30]= 2355.750000

Thread 3: d[31]= 2480.774902

Thread 3: d[32]= 2608.799805

Thread 3: d[33]= 2739.824951

Thread 3: d[34]= 2873.849854

Thread 3: d[35]= 3010.875000

Thread 3: d[36]= 3150.899902

Thread 3: d[37]= 3293.924805

Thread 3: d[38]= 3439.949951

Thread 3: d[39]= 3588.974854

Thread 3: d[40]= 3741.000000

Thread 3: d[41]= 3896.024902

Thread 3: d[42]= 4054.049805

Thread 3: d[43]= 4215.074707

Thread 3: d[44]= 4379.100098

Thread 3: d[45]= 4546.125000

Thread 3: d[46]= 4716.149902

Thread 3: d[47]= 4889.174805

Thread 3: d[48]= 5065.199707

Thread 3: d[49]= 5244.225098

Thread 3 done.

****

5. Example on using critical Directive

```
#include <omp.h>

main()
{
int x;
x = 0;
#pragma omp parallel shared(x)
{
#pragma omp critical
 x = x + 1;
     Printf("%d",x);
} /* end of parallel section */
```

```
}
```

****

6. Add two arrays A & B each of 1000 to generate an array C using reduction clause

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[])
{
int i, n;
float a[1000], b[1000], sum;

/* Some initializations */
n = 1000;
for (i=0; i < n; i++)
a[i] = b[i] = i * 1.0;
sum = 0.0;

#pragma omp parallel for reduction(+:sum)
for (i=0; i < n; i++)
sum = sum + (a[i] * b[i]);
```

```
printf(" Sum = %f\n",sum);


}
```

Output

```
 Sum = 332833152.000000
```

****

7. Multiply two matrices A & B and find the resultant matrix C

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>

#define NRA 62
#define NCA 15
#define NCB 7

int main (int argc, char *argv[])
{
int
tid, nthreads, i, j, k, chunk;
double a[NRA][NCA],
/* matrix A to be multiplied */
b[NCA][NCB],
```

```
/* matrix B to be multiplied */
c[NRA][NCB];
/* result matrix C */

/* number of rows in matrix A */
/* number of columns in matrix A */
/* number of columns in matrix B */
chunk = 10;

/*** Spawn a parallel region explicitly scoping all variables ***/
#pragma omp parallel shared(a,b,c,nthreads,chunk) private(tid,i,j,k)
{
tid = omp_get_thread_num();
if (tid == 0)
{
nthreads = omp_get_num_threads();
printf("Starting matrix multiple example with %d threads\n",nthreads);
printf("Initializing matrices...\n");
}
/*** Initialize matrices ***/
#pragma omp for schedule (static, chunk)
for (i=0; i<NRA; i++)
for (j=0; j<NCA; j++)
a[i][j]= i+j;
#pragma omp for schedule (static, chunk)
```

```
for (i=0; i<NCA; i++)
for (j=0; j<NCB; j++)
b[i][j]= i*j;
#pragma omp for schedule (static, chunk)
for (i=0; i<NRA; i++)
for (j=0; j<NCB; j++)
c[i][j]= 0;

/*** Do matrix multiply sharing iterations on outer loop ***/
/*** Display who does which iterations for demonstration purposes ***/
printf("Thread %d starting matrix multiply...\n",tid);
#pragma omp for schedule (static, chunk)
for (i=0; i<NRA; i++)
{
printf("Thread=%d did row=%d\n",tid,i);
for(j=0; j<NCB; j++)
for (k=0; k<NCA; k++)
c[i][j] += a[i][k] * b[k][j];
}
} /*** End of parallel region ***/

/*** Print results ***/

/* set loop iteration chunk size */
printf("******************************************************\n");
```

```
printf("Result Matrix:\n");
for (i=0; i<NRA; i++)
{
for (j=0; j<NCB; j++)
printf("%6.2f ", c[i][j]);
printf("\n");
}
printf("******************************************************\n");
printf ("Done.\n");

}
```

Output

```
Starting matrix multiple example with 4 threads

Initializing matrices...

Thread 0 starting matrix multiply...

Thread 3 starting matrix multiply...

Thread=3 did row=30

Thread=3 did row=31

Thread=3 did row=32

Thread=3 did row=33
```

Thread=3 did row=34

Thread=3 did row=35

Thread=0 did row=0

Thread 2 starting matrix multiply...

Thread 1 starting matrix multiply...

Thread=3 did row=36

Thread=3 did row=37

Thread=3 did row=38

Thread=2 did row=20

Thread=1 did row=10

Thread=0 did row=1

Thread=0 did row=2

Thread=0 did row=3

Thread=2 did row=21

Thread=0 did row=4

Thread=2 did row=22

Thread=3 did row=39

Thread=1 did row=11

Thread=0 did row=5

Thread=1 did row=12

Thread=1 did row=13

Thread=1 did row=14

Thread=1 did row=15

Thread=1 did row=16

Thread=1 did row=17

Thread=1 did row=18

Thread=1 did row=19

Thread=1 did row=50

Thread=1 did row=51

Thread=1 did row=52

Thread=1 did row=53

Thread=1 did row=54

Thread=1 did row=55

Thread=1 did row=56

Thread=1 did row=57

Thread=1 did row=58

Thread=1 did row=59

Thread=2 did row=23

Thread=2 did row=24

Thread=2 did row=25

Thread=0 did row=6

Thread=0 did row=7

Thread=0 did row=8

Thread=0 did row=9

Thread=2 did row=26

Thread=2 did row=27

Thread=2 did row=28

Thread=2 did row=29

Thread=2 did row=60

Thread=2 did row=61

Thread=0 did row=40

Thread=0 did row=41

Thread=0 did row=42

Thread=0 did row=43

Thread=0 did row=44

Thread=0 did row=45

Thread=0 did row=46

Thread=0 did row=47

Thread=0 did row=48

Thread=0 did row=49

****

Result Matrix:

0.00 1015.00 2030.00 3045.00 4060.00 5075.00 6090.00

0.00 1120.00 2240.00 3360.00 4480.00 5600.00 6720.00

0.00 1225.00 2450.00 3675.00 4900.00 6125.00 7350.00

0.00 1330.00 2660.00 3990.00 5320.00 6650.00 7980.00

0.00 1435.00 2870.00 4305.00 5740.00 7175.00 8610.00

0.00 1540.00 3080.00 4620.00 6160.00 7700.00 9240.00

0.00 1645.00 3290.00 4935.00 6580.00 8225.00 9870.00

0.00 1750.00 3500.00 5250.00 7000.00 8750.00 10500.00

0.00 1855.00 3710.00 5565.00 7420.00 9275.00 11130.00

0.00 1960.00 3920.00 5880.00 7840.00 9800.00 11760.00

0.00 2065.00 4130.00 6195.00 8260.00 10325.00 12390.00

0.00 2170.00 4340.00 6510.00 8680.00 10850.00 13020.00

0.00 2275.00 4550.00 6825.00 9100.00 11375.00 13650.00

0.00 2380.00 4760.00 7140.00 9520.00 11900.00 14280.00

0.00 2485.00 4970.00 7455.00 9940.00 12425.00 14910.00

0.00 2590.00 5180.00 7770.00 10360.00 12950.00 15540.00

0.00 2695.00 5390.00 8085.00 10780.00 13475.00 16170.00

0.00 2800.00 5600.00 8400.00 11200.00 14000.00 16800.00

0.00 2905.00 5810.00 8715.00 11620.00 14525.00 17430.00

0.00 3010.00 6020.00 9030.00 12040.00 15050.00 18060.00

0.00 3115.00 6230.00 9345.00 12460.00 15575.00 18690.00

0.00 3220.00 6440.00 9660.00 12880.00 16100.00 19320.00

0.00 3325.00 6650.00 9975.00 13300.00 16625.00 19950.00

0.00 3430.00 6860.00 10290.00 13720.00 17150.00 20580.00

0.00 3535.00 7070.00 10605.00 14140.00 17675.00 21210.00

0.00 3640.00 7280.00 10920.00 14560.00 18200.00 21840.00

0.00 3745.00 7490.00 11235.00 14980.00 18725.00 22470.00

0.00 3850.00 7700.00 11550.00 15400.00 19250.00 23100.00

0.00 3955.00 7910.00 11865.00 15820.00 19775.00 23730.00

0.00 4060.00 8120.00 12180.00 16240.00 20300.00 24360.00

0.00 4165.00 8330.00 12495.00 16660.00 20825.00 24990.00

0.00 4270.00 8540.00 12810.00 17080.00 21350.00 25620.00

0.00 4375.00 8750.00 13125.00 17500.00 21875.00 26250.00

0.00 4480.00 8960.00 13440.00 17920.00 22400.00 26880.00

0.00 4585.00 9170.00 13755.00 18340.00 22925.00 27510.00

0.00 4690.00 9380.00 14070.00 18760.00 23450.00 28140.00

0.00 4795.00 9590.00 14385.00 19180.00 23975.00 28770.00

0.00 4900.00 9800.00 14700.00 19600.00 24500.00 29400.00

0.00 5005.00 10010.00 15015.00 20020.00 25025.00 30030.00

0.00 5110.00 10220.00 15330.00 20440.00 25550.00 30660.00

0.00 5215.00 10430.00 15645.00 20860.00 26075.00 31290.00

0.00 5320.00 10640.00 15960.00 21280.00 26600.00 31920.00

0.00 5425.00 10850.00 16275.00 21700.00 27125.00 32550.00

0.00 5530.00 11060.00 16590.00 22120.00 27650.00 33180.00

0.00 5635.00 11270.00 16905.00 22540.00 28175.00 33810.00

0.00 5740.00 11480.00 17220.00 22960.00 28700.00 34440.00

0.00 5845.00 11690.00 17535.00 23380.00 29225.00 35070.00

0.00 5950.00 11900.00 17850.00 23800.00 29750.00 35700.00

0.00 6055.00 12110.00 18165.00 24220.00 30275.00 36330.00

0.00 6160.00 12320.00 18480.00 24640.00 30800.00 36960.00

0.00 6265.00 12530.00 18795.00 25060.00 31325.00 37590.00

0.00 6370.00 12740.00 19110.00 25480.00 31850.00 38220.00

0.00 6475.00 12950.00 19425.00 25900.00 32375.00 38850.00

0.00 6580.00 13160.00 19740.00 26320.00 32900.00 39480.00

0.00 6685.00 13370.00 20055.00 26740.00 33425.00 40110.00

0.00 6790.00 13580.00 20370.00 27160.00 33950.00 40740.00

0.00 6895.00 13790.00 20685.00 27580.00 34475.00 41370.00

0.00 7000.00 14000.00 21000.00 28000.00 35000.00 42000.00

0.00 7105.00 14210.00 21315.00 28420.00 35525.00 42630.00

0.00 7210.00 14420.00 21630.00 28840.00 36050.00 43260.00

0.00 7315.00 14630.00 21945.00 29260.00 36575.00 43890.00

0.00 7420.00 14840.00 22260.00 29680.00 37100.00 44520.00

****

Done.

8. Write a program to find the number of processes, number of threads etc

(environment information)

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[])
{
int nthreads, tid, procs, maxt, inpar, dynamic, nested;

/* Start parallel region */
#pragma omp parallel private(nthreads, tid)
{

/* Obtain thread number */
tid = omp_get_thread_num();

/* Only master thread does this */
if (tid == 0)
{
printf("Thread %d getting environment info...\n", tid);

/* Get environment information */
procs = omp_get_num_procs();
nthreads = omp_get_num_threads();
```

```
maxt = omp_get_max_threads();

inpar = omp_in_parallel();

dynamic = omp_get_dynamic();

nested = omp_get_nested();


/* Print environment information */

printf("Number of processors = %d\n", procs);

printf("Number of threads = %d\n", nthreads);

printf("Max threads = %d\n", maxt);

printf("In parallel? = %d\n", inpar);

printf("Dynamic threads enabled? = %d\n", dynamic);

printf("Nested parallelism supported? = %d\n", nested);


}


} /* Done */


}
```

Output

```
Thread 0 getting environment info...

Number of processors = 4

Number of threads = 4
```

Max threads = 4

In parallel? = 1

Dynamic threads enabled? = 0

Nested parallelism supported? = 0

****

9. Write a program to find the largest element in an array

```
#include <stdio.h>
#include <omp.h>
#include<stdlib.h>
#define MAXIMUM 65536

/* Main Program */

main()
{
int *array, i, Noofelements, cur_max, current_value;

printf("Enter the number of elements\n");
scanf("%d", &Noofelements);

if (Noofelements <= 0) {
```

```
printf("The array elements cannot be stored\n");
exit(1);
}
/* Dynamic Memory Allocation */

array = (int *) malloc(sizeof(int) * Noofelements);

*array, i, Noofelements, cur_max, current_value;
/* Allocating Random Number Values To The Elements Of An Array */

srand(MAXIMUM);
for (i = 0; i < Noofelements; i++)
array[i] = rand();

if (Noofelements == 1) {
printf("The Largest Number In The Array is %d", array[0]);
exit(1);
}
/* OpenMP Parallel For Directive And Critical Section */

cur_max = 0;
omp_set_num_threads(8);
#pragma omp parallel for
for (i = 0; i < Noofelements; i = i + 1) {
if (array[i] > cur_max)
```

```
#pragma omp critical
if (array[i] > cur_max)
cur_max = array[i];
}

/* Serial Calculation */

current_value = array[0];
for (i = 1; i < Noofelements; i++)
if (array[i] > current_value)
current_value = array[i];

printf("The Input Array Elements Are \n");

for (i = 0; i < Noofelements; i++)
printf("\t%d", array[i]);

printf("\n");

/* Checking For Output Validity */

if (current_value == cur_max)
printf("\nThe Max Value Is Same From Serial And Parallel OpenMP Directive\n");
else {
printf("\nThe Max Value Is Not Same In Serial And Parallel OpenMP Directive\n");
```

```
exit(1);
}

/* Freeing Allocated Memory */

printf("\n");
free(array);
printf("\nThe Largest Number In The Given Array Is %d\n", cur_max);

}
```

Output

```
Enter the number of elements
4
The Input Array Elements Are
553316596       1748907888      680492731       191440832

The Max Value Is Same From Serial And Parallel OpenMP Directive

The Largest Number In The Given Array Is 1748907888
```

10. Write a program to find the largest element in an array (usage of locks)

```
#include <stdio.h>
#include <omp.h>
#include<stdlib.h>

#define MINUS_INFINITY -9999
#define MAXIMUM_VALUE 65535

/* Main Program */

main()
{
int *array, i, Noofelements, cur_max, current_value;
omp_lock_t MAXLOCK;

printf("Enter the number of elements\n");
scanf("%d", &Noofelements);

if (Noofelements <= 0) {
printf("The array elements cannot be stored\n");
exit(1);
}
```

```
/* Dynamic Memory Allocation */
array = (int *) malloc(sizeof(int) * Noofelements);

/* Allocating Random Number To Array Elements */

srand(MAXIMUM_VALUE);
for (i = 0; i < Noofelements; i++)
array[i] = rand();

if (Noofelements == 1) {
printf("The Largest Element In The Array Is %d", array[0]);
exit(1);
}
/* Initializing The Lock */

printf("The locking is going to start\n");

omp_set_num_threads(8);
omp_init_lock(&MAXLOCK);
cur_max = MINUS_INFINITY;
printf("the lock s initialized\n");
/* OpenMP Parallel For Directive And Lock Functions */

#pragma omp parallel for
for (i = 0; i < Noofelements; i = i + 1) {
```

```
if (array[i] > cur_max) {
omp_set_lock(&MAXLOCK);
if (array[i] > cur_max)
cur_max = array[i];
omp_unset_lock(&MAXLOCK);
}
}

/* Destroying The Lock */

omp_destroy_lock(&MAXLOCK);

/* Serial Calculation */
current_value = array[0];
for (i = 1; i < Noofelements; i++)
if (array[i] > current_value)
current_value = array[i];

printf("The Array Elements Are \n");

for (i = 0; i < Noofelements; i++)
printf("\t%d", array[i]);

/* Checking For Output Validity */
```

```
if (current_value == cur_max)
printf("\nThe Max Value Is Same For Serial And Using Parallel OpenMP Directive\n");
else {
printf("\nThe Max Value Is Not Same In Serial And Using Parallel OpenMP Directive\n");
exit(1);
}

/* Freeing Allocated Memory */

free(array);

printf("\nThe Largest Number Of The Array Is %d\n", cur_max);
}
```

Output

```
Enter the number of elements
4
The locking is going to start
the lock s initialized
The Array Elements Are
        842357681       845752218       1085970682      559636718
```

The Max Value Is Same For Serial And Using Parallel OpenMP Directive

The Largest Number Of The Array Is 1085970682

11. Write a program to find the sum of an array A

```
#include<stdio.h>
#include<omp.h>
#include<stdlib.h>
#include<malloc.h>

/* Main Program */

double partialsum;
/* #pragma omp threadprivate (partialsum) */
main()
{
double *Array, *Array1, *Check, serial_sum, sum;
int array_size, i,threadid,tval;

printf("Enter the size of the array\n");
scanf("%d", &array_size);

if (array_size <= 0) {
```

```
printf("Array Size Should Be Of Positive Value ");
exit(1);
}
/* Dynamic Memory Allocation */

Array = (double *) malloc(sizeof(double) * array_size);
Check = (double *) malloc(sizeof(double) * array_size);

/* Array Elements Initialization */

for (i = 0; i < array_size; i++) {
Array[i] = i * 5;
Check[i] = Array[i];
}

printf("The Array Elements Are \n");

for (i = 0; i < array_size; i++)
printf("Array[%d]=%lf\n", i, Array[i]);

/* OpenMP Parallel For Directive And Critical Section */

sum=0.0;

omp_set_num_threads(4);
```

```
#pragma omp parallel for
for (i = 0; i < array_size; i++)
{
/* printf("the thread num and its iteration is %d %d
\n",omp_get_thread_num(),i); */
#pragma omp critical
sum = sum + Array[i];
}

serial_sum = 0.0;

/* Serial Calculation */
for (i = 0; i < array_size; i++)
serial_sum = serial_sum + Check[i];
printf("the sums are %lf and %lf",sum,serial_sum);
if (serial_sum == sum)
printf("\nThe Serial And Parallel Sums Are Equal\n");
else {
printf("\nThe Serial And Parallel Sums Are UnEqual\n");
exit(1);
}
/* Freeing Memory */
free(Check);
free(Array);
printf("\nThe SumOfElements Of The Array Using OpenMP Directives Is %lf\n", sum);
```

```
printf("\nThe SumOfElements Of The Array By Serial Calculation Is %lf\n", serial_sum);

}
```

Output

```
Enter the size of the array

5

The Array Elements Are

Array[0]=0.000000

Array[1]=5.000000

Array[2]=10.000000

Array[3]=15.000000
Array[4]=20.000000
the sums are 50.000000 and 50.000000
The Serial And Parallel Sums Are Equal

The SumOfElements Of The Array Using OpenMP Directives Is 50.000000

The SumOfElements Of The Array By Serial Calculation Is 50.000000
```

12. Write a program to Multiply a matrix by a vector and get the result of the operation.

```
#include <stdio.h>
#include <omp.h>
#include<stdlib.h>

/* Main Program */

main()
{

int
NoofRows, NoofCols, Vectorsize, i, j;
/*float
**Matrix, *Vector, *Result, *Checkoutput;*/
double **Matrix, *Vector, *Result, *Checkoutput;

printf("Read the matrix size noofrows and columns and vectorsize\n");
scanf("%d%d%d", &NoofRows, &NoofCols, &Vectorsize);

if (NoofRows <= 0 || NoofCols <= 0 || Vectorsize <= 0) {
printf("The Matrix and Vectorsize should be of positive sign\n");
```

```
exit(1);
}
/* Checking For Matrix Vector Computation Necessary Condition */

if (NoofCols != Vectorsize) {
printf("Matrix Vector computation cannot be possible \n");
exit(1);
}
/* Dynamic Memory Allocation And Initialization Of Matrix Elements */

/*

Matrix = (float **) malloc(sizeof(float) * NoofRows); */
Matrix = (double **) malloc(sizeof(double) * NoofRows);

for (i = 0; i < NoofRows; i++) {
/*
Matrix[i] = (float *) malloc(sizeof(float) * NoofCols); */
Matrix[i] = (double *) malloc(sizeof(double) * NoofCols);
for (j = 0; j < NoofCols; j++)
Matrix[i][j] = i + j;
}

/* Printing The Matrix */
```

```
printf("The Matrix is \n");
for (i = 0; i < NoofRows; i++) {
for (j = 0; j < NoofCols; j++)
printf("%lf \t", Matrix[i][j]);
printf("\n");
}

printf("\n");
/* Dynamic Memory Allocation */

/*Vector = (float *) malloc(sizeof(float) * Vectorsize);*/
Vector = (double *) malloc(sizeof(double) * Vectorsize);

/* vector Initialization */

for (i = 0; i < Vectorsize; i++)
Vector[i] = i;

printf("\n");

/* Printing The Vector Elements */

printf("The Vector is \n");
for (i = 0; i < Vectorsize; i++)
printf("%lf \t", Vector[i]);
```

```
/* Dynamic Memory Allocation */

/* Result = (float *) malloc(sizeof(float) * NoofRows);

Checkoutput = (float *) malloc(sizeof(float) * NoofRows); */

Result = (double *) malloc(sizeof(double) * NoofRows);

Checkoutput = (double *) malloc(sizeof(double) * NoofRows);

for (i = 0; i < NoofRows; i = i + 1)
{
Result[i]=0;
Checkoutput[i]=0;
}

/* OpenMP Parallel Directive */

omp_set_num_threads(32);
#pragma omp parallel for private(j)
for (i = 0; i < NoofRows; i = i + 1)
for (j = 0; j < NoofCols; j = j + 1)
Result[i] = Result[i] + Matrix[i][j] * Vector[j];
/* Serial Computation */
```

```
for (i = 0; i < NoofRows; i = i + 1)

for (j = 0; j < NoofCols; j = j + 1)

Checkoutput[i] = Checkoutput[i] + Matrix[i][j] * Vector[j];


/* Checking with the serial calculation */


for (i = 0; i < NoofRows; i = i + 1)

if (Checkoutput[i] == Result[i])

continue;

else {

printf("There is a difference from Serial and Parallel Computation \n");

exit(1);

}


printf("\nThe Matrix Computation result is \n");

for (i = 0; i < NoofRows; i++)

printf("%lf \n", Result[i]);


/* Freeing The Memory Allocations */


free(Vector);

free(Result);

free(Matrix);

free(Checkoutput);
```

}

Output

Read the matrix size noofrows and columns and vectorsize

2 2 2

The Matrix is

0.000000 1.000000
1.000000 2.000000
The Vector is
0.000000 1.000000
The Matrix Computation result is
1.000000
2.000000
****

13. Write a program to print all the letters of the alphabet

threads.

```
#include <stdio.h>
#include <omp.h>
int main(void)
{
int i;
omp_set_num_threads(4);
#pragma omp parallel private(i)
```

```
{ // OMP_NUM_THREADS is not a multiple of 26,
// which can be considered a bug in this code.
int LettersPerThread = 26 / omp_get_num_threads();
int ThisThreadNum = omp_get_thread_num();
int StartLetter = 'a'+ThisThreadNum*LettersPerThread;
int EndLetter = 'a'+ThisThreadNum*LettersPerThread+LettersPerThread;
for (i=StartLetter; i<EndLetter; i++)
printf("%c", i);
}
printf("\n");
return 0;
}
```

Output

abcdefmnopqrghijklstuvwx

****

14. Write a program to show how thread private clause works.

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

int a, b, i, tid;
float x;
```

```
#pragma omp threadprivate(a, x)

main () {

/* Explicitly turn off dynamic threads */
omp_set_dynamic(0);
printf("1st Parallel Region:\n");
#pragma omp parallel private(b,tid)
{
tid = omp_get_thread_num();
a = tid;
b = tid;
x = 1.1 * tid +1.0;
printf("Thread %d: a,b,x= %d %d %f\n",tid,a,b,x);
} /* end of parallel section */

printf("************************************\n");
printf("Master thread doing serial work here\n");
printf("************************************\n");

printf("2nd Parallel Region:\n");
#pragma omp parallel private(tid)
{
tid = omp_get_thread_num();
```

```
printf("Thread %d: a,b,x= %d %d %f\n",tid,a,b,x);

} /* end of parallel section */


}
```

Output

```
1st Parallel Region:

Thread 0: a,b,x= 0 0 1.000000

Thread 1: a,b,x= 1 1 2.100000

Thread 2: a,b,x= 2 2 3.200000

Thread 3: a,b,x= 3 3 4.300000

**********************************

Master thread doing serial work here

**********************************

2nd Parallel Region:

Thread 3: a,b,x= 3 0 4.300000

Thread 2: a,b,x= 2 0 3.200000

Thread 1: a,b,x= 1 0 2.100000
```

Thread 0: a,b,x= 0 0 1.000000

****

15. Write a program to show how first private clause works.(Factorial program)

```
#include <stdio.h>
#include <malloc.h>
#include <omp.h>

long long factorial(long n)
{
long long i,out;
out = 1;
for (i=1; i<n+1; i++) out *= i;
return(out);
}

int main(int argc, char **argv)
{
int i,j,threads;
long long *x;
long long n=12;

/* Set number of threads equal to argv[1] if present */
if (argc > 1)
```

```
{
threads = atoi(argv[1]);
if (omp_get_dynamic())
{
omp_set_dynamic(0);
printf("called omp_set_dynamic(0)\n");
}
omp_set_num_threads(threads);
}
printf("%d threads\n",omp_get_max_threads());

x = (long long *) malloc(n * sizeof(long));
for (i=0;i<n;i++) x[i]=factorial(i);
j=0;
/* Is the output the same if the following line is commented out? */
#pragma omp parallel for firstprivate(x,j)
for (i=1; i<n; i++)
{
j += i;
x[i] = j*x[i-1];
}
for (i=0; i<n; i++)
printf("factorial(%2d)=%14lld x[%2d]=%14lld\n",i,factorial(i),i,x[i]);
return 0;
```

```
}
```

Output

4 threads

```
factorial( 0)=           1 x[ 0]=           1

factorial( 1)=           1 x[ 1]=           1

factorial( 2)=           2 x[ 2]=           3

factorial( 3)=           6 x[ 3]=          18

factorial( 4)=          24 x[ 4]=          72

factorial( 5)=         120 x[ 5]=         648

factorial( 6)=         720 x[ 6]=        9720

factorial( 7)=        5040 x[ 7]=        5040

factorial( 8)=       40320 x[ 8]=       75600

factorial( 9)=      362880 x[ 9]=     1814400

factorial(10)=     3628800 x[10]=     3628800

factorial(11)=    39916800 x[11]=    76204800
```

****

16. Write a program to show how last private clause works. (Sum of powers)

```
#include <stdio.h>
#include <malloc.h>
#include <omp.h>

int main(int argc, char **argv)
{
int i,j,threads;
int x[10];
int *sum_of_powers;
int n=10;

/* Set number of threads equal to argv[1] if present */
if (argc > 1)
{
threads = atoi(argv[1]);
if (omp_get_dynamic())
{
omp_set_dynamic(0);
printf("called omp_set_dynamic(0)\n");
}
omp_set_num_threads(threads);
}
printf("%d threads max\n",omp_get_max_threads());
```

```
sum_of_powers = (int *) malloc(n * sizeof(int));
/* Is the output the same if the lastprivate clause is commented out? */
#pragma omp parallel for private(j) lastprivate(x)
for (i=0; i<n; i++)
{
printf("%d threads currently executing\n",omp_get_num_threads());
x[0] = 1;
for (j=1; j<4; j++) x[j] = x[j-1] * (i+1);
sum_of_powers[i] = x[0] + x[1] + x[2] + x[3];
}
for (i=0; i<n; i++)
{
printf("Sum of powers 0-3 of %d = %d\n",i+1,sum_of_powers[i]);
}
printf("n cubed = %d\n",x[3]);
return 0;

}
```

Output

4 threads max

4 threads currently executing

4 threads currently executing

4 threads currently executing

4 threads currently executing

4 threads currently executing

4 threads currently executing

4 threads currently executing

4 threads currently executing

4 threads currently executing

4 threads currently executing

Sum of powers 0-3 of 1 = 4

Sum of powers 0-3 of 2 = 15

Sum of powers 0-3 of 3 = 40

Sum of powers 0-3 of 4 = 85

Sum of powers 0-3 of 5 = 156

Sum of powers 0-3 of 6 = 259

Sum of powers 0-3 of 7 = 400

Sum of powers 0-3 of 8 = 585

Sum of powers 0-3 of 9 = 820

Sum of powers 0-3 of 10 = 1111

n cubed = 1000

****

17.Write a program to find prime numbers (split)

```
#include <stdio.h>
#include <omp.h>
#define N 100000000
#define TRUE 1
#define FALSE 0

int main(int argc, char **argv )
{
char host[80];
int *a;
int i, k, threads, pcount;
double t1, t2;
int found;

/* Set number of threads equal to argv[1] if present */
if (argc > 1)
{
```

```
threads = atoi(argv[1]);
if (omp_get_dynamic())
{
omp_set_dynamic(0);
printf("called omp_set_dynamic(0)\n");
}
omp_set_num_threads(threads);
}
printf("%d threads max\n",omp_get_max_threads());

a = (int *) malloc((N+1) * sizeof(int));
// 1. create a list of natural numbers 2, 3, 4, ... none of which is marked.
for (i=2;i<=N;i++) a[i] = 1;
// 2. Set k = 2, the first unmarked number on the list.
k = 2;

t1 = omp_get_wtime();
// 3. Repeat
#pragma omp parallel firstprivate(k) private(i,found)
while (k*k <= N)
{
// a. Mark all multiples of k between k^2 and N
#pragma omp for

for (i=k*k; i<=N; i+=k) a[i] = 0;
```

```
// b. Find the smallest number greater than k that is unmarked
// and set k to this new value until k^2 > N
found = FALSE;
for (i=k+1;!found;i++)
{
if (a[i]){ k = i; found = TRUE; }
}

}
t2 = omp_get_wtime();
printf("%.2f seconds\n",t2-t1);

// 4. The unmarked numbers are primes
pcount = 0;
for (i=2;i<=N;i++)
{
if( a[i] )
{
pcount++;
//printf("%d\n",i);
}
}
printf("%d primes between 0 and %d\n",pcount,N);

}
```

Output

4 threads max

5.11 seconds

5761455 primes between 0 and 100000000

****

MPI programs

P2p.c

```
#include <stdio.h>
#include <mpi.h>
#include <string.h>
#define BUFFER_SIZE 32

int main(int argc,char *argv[])
{
	int  MyRank,Numprocs, Destination, iproc;
	int  tag = 0;
	int  Root = 0, temp = 1;
	char Message[BUFFER_SIZE];
	MPI_Init(&argc,&argv);
	MPI_Status status;

	MPI_Comm_rank(MPI_COMM_WORLD,&MyRank);
	MPI_Comm_size(MPI_COMM_WORLD,&Numprocs);

	/* print host name, and send message from process with rank 0 to all other
processes */
	if(MyRank == 0)		{
		system("hostname");
		strcpy(Message, "Hello India");
		for (temp=1; temp<Numprocs;temp++)
		{
			MPI_Send(Message, BUFFER_SIZE, MPI_CHAR, temp, tag,MPI_COMM_WORLD);
		}
	}
	else {
```

```
            system("hostname");
            MPI_Recv(Message, BUFFER_SIZE, MPI_CHAR, Root, tag,MPI_COMM_WORLD,
&status);
            printf("\n%s in process with rank %d from Process with rank %d\n",
Message,MyRank,Root);
       }

       MPI_Finalize();
}
```

P2p sum.c

```
#include <stdio.h>
#include "mpi.h"

int main(int argc,char *argv[])
{
    int iproc;
    int MyRank, Numprocs, Root = 0;
    int value, sum = 0;
    int       Source, Source_tag;
    int Destination, Destination_tag;
    MPI_Status status;


    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&Numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&MyRank);

    if(MyRank == Root){

       for(iproc = 1 ; iproc < Numprocs ; iproc++){
           Source     = iproc;
          Source_tag = 0;

          MPI_Recv(&value, 1, MPI_INT, Source, Source_tag,
                   MPI_COMM_WORLD, &status);
          sum = sum + value;
       }
       printf("MyRank = %d, SUM = %d\n", MyRank, sum);
    }
    else{
       Destination     = 0;
       Destination_tag = 0;

       MPI_Send(&MyRank, 1, MPI_INT, Destination, Destination_tag,
                MPI_COMM_WORLD);
    }

    MPI_Finalize();

}
```

Broadcast

```
#include <stdio.h>
#include "mpi.h"

int main (int argc, char *argv[])
{
        int rank, i;

         MPI_Init (&argc, &argv);

   MPI_Comm_rank (MPI_COMM_WORLD, &rank);


   if (rank == 0) i = 27;

   MPI_Bcast ((void *)&i, 1, MPI_INT, 0, MPI_COMM_WORLD);

   printf ("[%d] i = %d\n", rank, i);

   // Wait for every process to reach this code


   MPI_Barrier (MPI_COMM_WORLD);

   MPI_Finalize();


   return 0;


}
```

Gather.c

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

  void main(int argc, char *argv[])
  {
       int rank,size;
       double param[6],mine;
       int sndcnt,rcvcnt;
       int i;

       MPI_Init(&argc, &argv);
       MPI_Comm_rank(MPI_COMM_WORLD,&rank);
       MPI_Comm_size(MPI_COMM_WORLD,&size);

       sndcnt=1;
       mine=23.0+rank;
       if(rank==3) rcvcnt=1;

       MPI_Gather(&mine,sndcnt,MPI_DOUBLE,param,rcvcnt,MPI_DOUBLE,3,MPI_COMM_WORLD);
```

```
        if(rank==3)
                for(i=0;i<size;++i)
                  //printf("PE:%d param[%d] is %f \n",rank,i,param[i]]);
                  printf(" %d %d  \n",rank,i);

        MPI_Finalize();
}
```

Pie collective.c

```
#include <stdio.h>
#include <math.h>
#include "mpi.h"

double func(double x)
{
    return (4.0 / (1.0 + x*x));
}

int main(int argc,char *argv[])
{
    int    NoInterval, interval;
    int    MyRank, Numprocs, Root = 0;
    double mypi, pi, h, sum, x;
    double PI25DT = 3.14159265358979323846264338327950288419716939937510;

    /*....MPI initialisation....*/
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&Numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&MyRank);

    if(MyRank == Root){
       printf("\nEnter the number of intervals : ");
       scanf("%d",&NoInterval);
    }

   /*....Broadcast the number of subintervals to each processor....*/
   MPI_Bcast(&NoInterval, 1, MPI_INT, 0, MPI_COMM_WORLD);

   if(NoInterval <= 0){
      if(MyRank == Root)
        printf("Invalid Value for Number of Intervals .....\n");
      MPI_Finalize();
      exit(-1);
   }

   h   = 1.0 / (double)NoInterval;
   sum = 0.0;
   for(interval = MyRank + 1; interval <= NoInterval; interval += Numprocs){
       x = h * ((double)interval - 0.5);
       sum += func(x);
   }
   mypi = h * sum;

   /*....Collect the areas calculated in P0....*/
```

```
   MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, Root, MPI_COMM_WORLD);

   if(MyRank == Root){
      printf("pi is approximately %.16f, Error is %.16f\n",
                                              pi, fabs(pi - PI25DT));
   }

   MPI_Finalize();

}
```

// CUDA programs

```
// include files
//

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
```

```
#include <math.h>


//
// kernel routine
//

__global__ void my_first_kernel(float *x)
{
  int tid = threadIdx.x + blockDim.x*blockIdx.x;


  x[tid] = (float) threadIdx.x;
}


//
// main code
//

int main(int argc, char **argv)
{
  float *h_x, *d_x;
  int   nblocks, nthreads, nsize, n;

// set number of blocks, and threads per block

  nblocks  = 2;
  nthreads = 1024;
  nsize    = nblocks*nthreads ;

  // allocate memory for array

  h_x = (float *)malloc(nsize*sizeof(float));

   //Allocate device memory
  cudaMalloc((void **)&d_x, nsize*sizeof(float));



// copy from host to device
cudaMemcpy(d_x,h_x,nsize*sizeof(float),cudaMemcpyHostToDevice);

  // execute kernel

  my_first_kernel<<<nblocks, nthreads>>>(d_x);

  // copy back results and print them out

  cudaMemcpy(h_x,d_x,nsize*sizeof(float),cudaMemcpyDeviceToHost);

  for (n=0; n<nsize; n++)
      printf(" n,  x  =  %d  %f \n",n,h_x[n]);
```

```
  // free memory

  cudaFree(d_x);
  free(h_x);

  // CUDA exit -- needed to flush printf write buffer

  cudaDeviceReset();

  return 0;
}
```

Program2

```
#include "stdio.h"
__global__ void my_kernel()
{
}
int main()
{
my_kernel<<<1,1>>>();
printf("Hello world\n");
return 0;
}
```

Program3

```
#include <stdio.h>

#define NUM_BLOCKS 32
#define BLOCK_WIDTH 1

__global__ void hello()
{
    printf("Hello world! I'm a thread in block %d\n", blockIdx.x);
}


int main(int argc,char **argv)
{
    // launch the kernel
    hello<<<NUM_BLOCKS, BLOCK_WIDTH>>>();

    // force the printf()s to flush
    cudaDeviceSynchronize();
```

```
    printf("That's all!\n");

    return 0;
}


#include <stdio.h>

#define NUM_BLOCKS 1
#define BLOCK_WIDTH 512

__global__ void hello()
{
    printf("Hello world! I'm thread %d\n", threadIdx.x);
}


int main(int argc,char **argv)
{
    // launch the kernel
    hello<<<NUM_BLOCKS, BLOCK_WIDTH>>>();

    // force the printf()s to flush
    cudaDeviceSynchronize();

    printf("That's all!\n");

    return 0;
}



// Using different memory spaces in CUDA
#include <stdio.h>

/**********************
 * using local memory *
 **********************/

// a __device__ or __global__ function runs on the GPU
__global__ void use_local_memory_GPU(float in)
{
    float f;    // variable "f" is in local memory and private to each
thread
    f = in;     // parameter "in" is in local memory and private to each
thread
    // ... real code would presumably do other stuff here ...
}

/**********************
 * using global memory *
 **********************/
```

```
// a __global__ function runs on the GPU & can be called from host
__global__ void use_global_memory_GPU(float *array)
{
    // "array" is a pointer into global memory on the device
    array[threadIdx.x] = 2.0f * (float) threadIdx.x;
}

/**********************
 * using shared memory *
 **********************/

// (for clarity, hardcoding 128 threads/elements and omitting out-of-
bounds checks)
__global__ void use_shared_memory_GPU(float *array)
{
    // local variables, private to each thread
    int i, index = threadIdx.x;
    float average, sum = 0.0f;

    // __shared__ variables are visible to all threads in the thread block
    // and have the same lifetime as the thread block
    __shared__ float sh_arr[128];

    // copy data from "array" in global memory to sh_arr in shared memory.
    // here, each thread is responsible for copying a single element.
    sh_arr[index] = array[index];

    __syncthreads();    // ensure all the writes to shared memory have
completed

    // now, sh_arr is fully populated. Let's find the average of all
previous elements
    for (i=0; i<index; i++) { sum += sh_arr[i]; }
    average = sum / (index + 1.0f);

      printf("Thread id = %d\t Average = %f\n",index,average);
    // if array[index] is greater than the average of array[0..index-1],
replace with average.
    // since array[] is in global memory, this change will be seen by the
host (and potentially
    // other thread blocks, if any)
    if (array[index] > average) { array[index] = average; }

    // the following code has NO EFFECT: it modifies shared memory, but
    // the resulting modified data is never copied back to global memory
    // and vanishes when the thread block completes
    sh_arr[index] = 3.14;
}

int main(int argc, char **argv)
{
    /*
     * First, call a kernel that shows using local memory
     */
```

```
    use_local_memory_GPU<<<1, 128>>>(2.0f);

    /*
     * Next, call a kernel that shows using global memory
     */
    float h_arr[128];   // convention: h_ variables live on host
    float *d_arr;       // convention: d_ variables live on device (GPU
global mem)

    // allocate global memory on the device, place result in "d_arr"
    cudaMalloc((void **) &d_arr, sizeof(float) * 128);
    // now copy data from host memory "h_arr" to device memory "d_arr"
    cudaMemcpy((void *)d_arr, (void *)h_arr, sizeof(float) * 128,
cudaMemcpyHostToDevice);
    // launch the kernel (1 block of 128 threads)
    use_global_memory_GPU<<<1, 128>>>(d_arr);  // modifies the contents of
array at d_arr
    // copy the modified array back to the host, overwriting contents of
h_arr
    cudaMemcpy((void *)h_arr, (void *)d_arr, sizeof(float) * 128,
cudaMemcpyDeviceToHost);
    // ... do other stuff ...

    /*
     * Next, call a kernel that shows using shared memory
     */

    // as before, pass in a pointer to data in global memory
    use_shared_memory_GPU<<<1, 128>>>(d_arr);
    // copy the modified array back to the host
    cudaMemcpy((void *)h_arr, (void *)d_arr, sizeof(float) * 128,
cudaMemcpyHostToDevice);
    // ... do other stuff ...

// force the printf()s to flush
    cudaDeviceSynchronize();
    return 0;
}
```

Pgogram5

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

// CUDA kernel. Each thread takes care of one element of c
__global__ void vecAdd(double *a, double *b, double *c, int n)
{
    // Get our global thread ID
    int id = blockIdx.x*blockDim.x+threadIdx.x;
```

```
    // Make sure we do not go out of bounds
    if (id < n)
        c[id] = a[id] + b[id];
}

int main( int argc, char* argv[] )
{
    // Size of vectors
    int n = 100;

    // Host input vectors
    double *h_a;
    double *h_b;
    //Host output vector
    double *h_c;

    // Device input vectors
    double *d_a;
    double *d_b;
    //Device output vector
    double *d_c;

    // Size, in bytes, of each vector
    size_t bytes = n*sizeof(double);

    // Allocate memory for each vector on host
    h_a = (double*)malloc(bytes);
    h_b = (double*)malloc(bytes);
    h_c = (double*)malloc(bytes);

    // Allocate memory for each vector on GPU
    cudaMalloc(&d_a, bytes);
    cudaMalloc(&d_b, bytes);
    cudaMalloc(&d_c, bytes);

    int i;
    // Initialize vectors on host
    for( i = 0; i < n; i++ ) {
        h_a[i] = i;
        h_b[i] = i;
    }

    // Copy host vectors to device
    cudaMemcpy( d_a, h_a, bytes, cudaMemcpyHostToDevice);
    cudaMemcpy( d_b, h_b, bytes, cudaMemcpyHostToDevice);

    int blockSize, gridSize;

    // Number of threads in each thread block
    blockSize = 1024;

    // Number of thread blocks in grid
    gridSize = (int)ceil((float)n/blockSize);
```

```
    // Execute the kernel
    vecAdd<<<gridSize, blockSize>>>(d_a, d_b, d_c, n);

    // Copy array back to host
    cudaMemcpy( h_c, d_c, bytes, cudaMemcpyDeviceToHost );

    // Sum up vector c and print result divided by n, this should equal 1
within error
    double sum = 0;
    for(i=0; i<n; i++)
        printf(" %f + %f =%f\n",h_a[i],h_b[i],h_c[i]);
    //printf("final result: %f\n", sum/(double)n);

    // Release device memory
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);

    // Release host memory
    free(h_a);
    free(h_b);
    free(h_c);

    return 0;
}
```

Pgogram 6

```
//  Multiply two matrices A * B = C
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
//Thread  block  size
#define BLOCK_SIZE 3
#define WA 3
//  Matrix A  width
#define HA 3
//  Matrix  A  height
#define WB 3
//  Matrix  B  width
#define HB WA
//  Matrix  B  height
#define WC WB
//  Matrix  C  width
#define HC HA
//  Matrix  C  height
```

```
//Allocates a  matrix  with  random  float  entries.
void randomInit(float * data ,int size)
{
      for(int i = 0; i < size; ++i)
            //data[i] = rand() / (float) RAND_MAX;
            data[i] = i;
}


//  CUDA  Kernel
__global__ void matrixMul(float* C,float* A,float* B,int wA,int wB)
{
// 2D  Thread  ID
int tx = threadIdx.x;
int ty = threadIdx.y;
//  value  stores  the  element  that  is  computed  by  the thread
float value = 0;
for(int i = 0; i < wA; ++i)
{
float elementA = A[ty * wA + i];
float elementB = B[i * wB + tx];
value  +=  elementA * elementB;
}
//  Write  the  matrix  to  device  memory  each
//  thread  writes  one  element
C[ty * wA + tx] = value;
}



//  Program  main
int main(int argc ,char** argv)
{
//  set  seed  for  rand()
srand(2006);
//  1.  allocate  host  memory  for  matrices  A and B
unsigned  int size_A = WA * HA;
unsigned  int mem_size_A =sizeof(float) * size_A;
float* h_A = (float*)  malloc(mem_size_A);
unsigned  int size_B = WB * HB;
unsigned  int mem_size_B =sizeof(float) * size_B;
float * h_B = (float*)  malloc(mem_size_B);
//  2.  initialize  host  memory
randomInit(h_A, size_A);
randomInit(h_B, size_B);
//  3.  print  out A and B
printf("\n\nMatrix A\n");
for(int i = 0; i < size_A; i++)
{
printf("%f ", h_A[i]);
if(((i + 1) % WA) == 0)
printf("\n");
}
printf("\n\nMatrix B\n");
```

```
for(int i = 0; i < size_B; i++)
{
printf
("%f ", h_B[i]);
if(((i + 1) % WB) == 0)
printf("\n");
}
//  4.  allocate host  memory  for  the  result  C
unsigned  int size_C = WC * HC;
unsigned  int mem_size_C =sizeof(float) * size_C;
float * h_C = (float *)  malloc(mem_size_C);

//  8.  allocate  device  memory
float* d_A;
float* d_B;
cudaMalloc((void**) &d_A, mem_size_A);
cudaMalloc((void**) &d_B, mem_size_B);
//9.  copy  host  memory  to  device
cudaMemcpy(d_A, h_A,mem_size_A ,cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B,mem_size_B ,cudaMemcpyHostToDevice);
//  10.  allocate  device  memory  for  the  result
float* d_C;
cudaMalloc((void**) &d_C, mem_size_C);


//  5.  perform  the  calculation
//     setup  execution  parameters
dim3  threads(BLOCK_SIZE , BLOCK_SIZE);
dim3  grid(WC / threads.x, HC / threads.y);
//    execute  the  kernel
matrixMul<<< grid , threads  >>>(d_C, d_A,d_B, WA, WB);




//  11.  copy  result  from  device  to  host
cudaMemcpy(h_C, d_C, mem_size_C ,cudaMemcpyDeviceToHost);
//  6.  print  out  the  results
printf("\n\n Matrix C ( Results ) \n ");
for(int i = 0;i<size_C; i ++){
      printf("%f",h_C[i]);
      if(((i+ 1) % WC) == 0)
            printf("\n");
}
printf("\n");
// 7.clean up memory
cudaFree(d_A);
cudaFree(d_B);
cudaFree(d_C);
free(h_A);
free(h_B);
free(h_C);
```

```
}
```

Program7

```
// Copyright 2012 NVIDIA Corporation
//
// Licensed under the Apache License, Version 2.0 (the "License");
// you may not use this file except in compliance with the License.
// You may obtain a copy of the License at
//
//     http://www.apache.org/licenses/LICENSE-2.0
//
// Unless required by applicable law or agreed to in writing, software
// distributed under the License is distributed on an "AS IS" BASIS,
// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
implied.
// See the License for the specific language governing permissions and
// limitations under the License.

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

// Convenience function for checking CUDA runtime API results
// can be wrapped around any runtime API call. No-op in release builds.
inline
cudaError_t checkCuda(cudaError_t result)
{
#if defined(DEBUG) || defined(_DEBUG)
  if (result != cudaSuccess) {
    fprintf(stderr, "CUDA Runtime Error: %s\n",
cudaGetErrorString(result));
    assert(result == cudaSuccess);
  }
#endif
  return result;
}

const int TILE_DIM = 32;
const int BLOCK_ROWS = 8;
const int NUM_REPS = 100;

// Check errors and print GB/s
void postprocess(const float *ref, const float *res, int n, float ms)
{
  bool passed = true;
  for (int i = 0; i < n; i++)
    if (res[i] != ref[i]) {
      printf("%d %f %f\n", i, res[i], ref[i]);
      printf("%25s\n", "*** FAILED ***");
      passed = false;
      break;
    }
  if (passed)
```

```
    printf("%20.2f\n", 2 * n * sizeof(float) * 1e-6 * NUM_REPS / ms );
}

// simple copy kernel
// Used as reference case representing best effective bandwidth.
__global__ void copy(float *odata, const float *idata)
{
  int x = blockIdx.x * TILE_DIM + threadIdx.x;
  int y = blockIdx.y * TILE_DIM + threadIdx.y;
  int width = gridDim.x * TILE_DIM;

  for (int j = 0; j < TILE_DIM; j+= BLOCK_ROWS)
    odata[(y+j)*width + x] = idata[(y+j)*width + x];
}

// copy kernel using shared memory
// Also used as reference case, demonstrating effect of using shared
memory.
__global__ void copySharedMem(float *odata, const float *idata)
{
  __shared__ float tile[TILE_DIM * TILE_DIM];

  int x = blockIdx.x * TILE_DIM + threadIdx.x;
  int y = blockIdx.y * TILE_DIM + threadIdx.y;
  int width = gridDim.x * TILE_DIM;

  for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS)
     tile[(threadIdx.y+j)*TILE_DIM + threadIdx.x] = idata[(y+j)*width +
x];

  __syncthreads();

  for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS)
     odata[(y+j)*width + x] = tile[(threadIdx.y+j)*TILE_DIM +
threadIdx.x];
}

// naive transpose
// Simplest transpose; doesn't use shared memory.
// Global memory reads are coalesced but writes are not.
__global__ void transposeNaive(float *odata, const float *idata)
{
  int x = blockIdx.x * TILE_DIM + threadIdx.x;
  int y = blockIdx.y * TILE_DIM + threadIdx.y;
  int width = gridDim.x * TILE_DIM;

  for (int j = 0; j < TILE_DIM; j+= BLOCK_ROWS)
    odata[x*width + (y+j)] = idata[(y+j)*width + x];
}

// coalesced transpose
// Uses shared memory to achieve coalesing in both reads and writes
// Tile width == #banks causes shared memory bank conflicts.
__global__ void transposeCoalesced(float *odata, const float *idata)
```

```
{
  __shared__ float tile[TILE_DIM][TILE_DIM];

  int x = blockIdx.x * TILE_DIM + threadIdx.x;
  int y = blockIdx.y * TILE_DIM + threadIdx.y;
  int width = gridDim.x * TILE_DIM;

  for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS)
     tile[threadIdx.y+j][threadIdx.x] = idata[(y+j)*width + x];

  __syncthreads();

  x = blockIdx.y * TILE_DIM + threadIdx.x;  // transpose block offset
  y = blockIdx.x * TILE_DIM + threadIdx.y;

  for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS)
     odata[(y+j)*width + x] = tile[threadIdx.x][threadIdx.y + j];
}


// No bank-conflict transpose
// Same as transposeCoalesced except the first tile dimension is padded
// to avoid shared memory bank conflicts.
__global__ void transposeNoBankConflicts(float *odata, const float *idata)
{
  __shared__ float tile[TILE_DIM][TILE_DIM+1];

  int x = blockIdx.x * TILE_DIM + threadIdx.x;
  int y = blockIdx.y * TILE_DIM + threadIdx.y;
  int width = gridDim.x * TILE_DIM;

  for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS)
     tile[threadIdx.y+j][threadIdx.x] = idata[(y+j)*width + x];

  __syncthreads();

  x = blockIdx.y * TILE_DIM + threadIdx.x;  // transpose block offset
  y = blockIdx.x * TILE_DIM + threadIdx.y;

  for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS)
     odata[(y+j)*width + x] = tile[threadIdx.x][threadIdx.y + j];
}

int main(int argc, char **argv)
{
  const int nx = 1024;
  const int ny = 1024;
  const int mem_size = nx*ny*sizeof(float);

  dim3 dimGrid(nx/TILE_DIM, ny/TILE_DIM, 1);
  dim3 dimBlock(TILE_DIM, BLOCK_ROWS, 1);

  int devId = 0;
  if (argc > 1) devId = atoi(argv[1]);
```

```
  cudaDeviceProp prop;
  checkCuda( cudaGetDeviceProperties(&prop, devId));
  printf("\nDevice : %s\n", prop.name);
  printf("Matrix size: %d %d, Block size: %d %d, Tile size: %d %d\n",
         nx, ny, TILE_DIM, BLOCK_ROWS, TILE_DIM, TILE_DIM);
  printf("dimGrid: %d %d %d. dimBlock: %d %d %d\n",
         dimGrid.x, dimGrid.y, dimGrid.z, dimBlock.x, dimBlock.y,
dimBlock.z);

  checkCuda( cudaSetDevice(devId) );

  float *h_idata = (float*)malloc(mem_size);
  float *h_cdata = (float*)malloc(mem_size);
  float *h_tdata = (float*)malloc(mem_size);
  float *gold    = (float*)malloc(mem_size);

  float *d_idata, *d_cdata, *d_tdata;
  checkCuda( cudaMalloc(&d_idata, mem_size) );
  checkCuda( cudaMalloc(&d_cdata, mem_size) );
  checkCuda( cudaMalloc(&d_tdata, mem_size) );

  // check parameters and calculate execution configuration
  if (nx % TILE_DIM || ny % TILE_DIM) {
    printf("nx and ny must be a multiple of TILE_DIM\n");
    goto error_exit;
  }

  if (TILE_DIM % BLOCK_ROWS) {
    printf("TILE_DIM must be a multiple of BLOCK_ROWS\n");
    goto error_exit;
  }

  // host
  for (int j = 0; j < ny; j++)
    for (int i = 0; i < nx; i++)
      h_idata[j*nx + i] = j*nx + i;

  // correct result for error checking
  for (int j = 0; j < ny; j++)
    for (int i = 0; i < nx; i++)
      gold[j*nx + i] = h_idata[i*nx + j];

  // device
  checkCuda( cudaMemcpy(d_idata, h_idata, mem_size,
cudaMemcpyHostToDevice) );

  // events for timing
  cudaEvent_t startEvent, stopEvent;
  checkCuda( cudaEventCreate(&startEvent) );
  checkCuda( cudaEventCreate(&stopEvent) );
  float ms;

  // ------------
```

```
  // time kernels
  // ------------
  printf("%25s%25s\n", "Routine", "Bandwidth (GB/s)");

  // ----
  // copy
  // ----
  printf("%25s", "copy");
  checkCuda( cudaMemset(d_cdata, 0, mem_size) );
  // warm up
  copy<<<dimGrid, dimBlock>>>(d_cdata, d_idata);
  checkCuda( cudaEventRecord(startEvent, 0) );
  for (int i = 0; i < NUM_REPS; i++)
     copy<<<dimGrid, dimBlock>>>(d_cdata, d_idata);
  checkCuda( cudaEventRecord(stopEvent, 0) );
  checkCuda( cudaEventSynchronize(stopEvent) );
  checkCuda( cudaEventElapsedTime(&ms, startEvent, stopEvent) );
  checkCuda( cudaMemcpy(h_cdata, d_cdata, mem_size,
cudaMemcpyDeviceToHost) );
  postprocess(h_idata, h_cdata, nx*ny, ms);

  // -------------
  // copySharedMem
  // -------------
  printf("%25s", "shared memory copy");
  checkCuda( cudaMemset(d_cdata, 0, mem_size) );
  // warm up
  copySharedMem<<<dimGrid, dimBlock>>>(d_cdata, d_idata);
  checkCuda( cudaEventRecord(startEvent, 0) );
  for (int i = 0; i < NUM_REPS; i++)
     copySharedMem<<<dimGrid, dimBlock>>>(d_cdata, d_idata);
  checkCuda( cudaEventRecord(stopEvent, 0) );
  checkCuda( cudaEventSynchronize(stopEvent) );
  checkCuda( cudaEventElapsedTime(&ms, startEvent, stopEvent) );
  checkCuda( cudaMemcpy(h_cdata, d_cdata, mem_size,
cudaMemcpyDeviceToHost) );
  postprocess(h_idata, h_cdata, nx * ny, ms);

  // --------------
  // transposeNaive
  // --------------
  printf("%25s", "naive transpose");
  checkCuda( cudaMemset(d_tdata, 0, mem_size) );
  // warmup
  transposeNaive<<<dimGrid, dimBlock>>>(d_tdata, d_idata);
  checkCuda( cudaEventRecord(startEvent, 0) );
  for (int i = 0; i < NUM_REPS; i++)
     transposeNaive<<<dimGrid, dimBlock>>>(d_tdata, d_idata);
  checkCuda( cudaEventRecord(stopEvent, 0) );
  checkCuda( cudaEventSynchronize(stopEvent) );
  checkCuda( cudaEventElapsedTime(&ms, startEvent, stopEvent) );
  checkCuda( cudaMemcpy(h_tdata, d_tdata, mem_size,
cudaMemcpyDeviceToHost) );
  postprocess(gold, h_tdata, nx * ny, ms);
```

```
  // ------------------
  // transposeCoalesced
  // ------------------
  printf("%25s", "coalesced transpose");
  checkCuda( cudaMemset(d_tdata, 0, mem_size) );
  // warmup
  transposeCoalesced<<<dimGrid, dimBlock>>>(d_tdata, d_idata);
  checkCuda( cudaEventRecord(startEvent, 0) );
  for (int i = 0; i < NUM_REPS; i++)
     transposeCoalesced<<<dimGrid, dimBlock>>>(d_tdata, d_idata);
  checkCuda( cudaEventRecord(stopEvent, 0) );
  checkCuda( cudaEventSynchronize(stopEvent) );
  checkCuda( cudaEventElapsedTime(&ms, startEvent, stopEvent) );
  checkCuda( cudaMemcpy(h_tdata, d_tdata, mem_size,
cudaMemcpyDeviceToHost) );
  postprocess(gold, h_tdata, nx * ny, ms);

  // ------------------------
  // transposeNoBankConflicts
  // ------------------------
  printf("%25s", "conflict-free transpose");
  checkCuda( cudaMemset(d_tdata, 0, mem_size) );
  // warmup
  transposeNoBankConflicts<<<dimGrid, dimBlock>>>(d_tdata, d_idata);
  checkCuda( cudaEventRecord(startEvent, 0) );
  for (int i = 0; i < NUM_REPS; i++)
     transposeNoBankConflicts<<<dimGrid, dimBlock>>>(d_tdata, d_idata);
  checkCuda( cudaEventRecord(stopEvent, 0) );
  checkCuda( cudaEventSynchronize(stopEvent) );
  checkCuda( cudaEventElapsedTime(&ms, startEvent, stopEvent) );
  checkCuda( cudaMemcpy(h_tdata, d_tdata, mem_size,
cudaMemcpyDeviceToHost) );
  postprocess(gold, h_tdata, nx * ny, ms);

error_exit:
  // cleanup
  checkCuda( cudaEventDestroy(startEvent) );
  checkCuda( cudaEventDestroy(stopEvent) );
  checkCuda( cudaFree(d_tdata) );
  checkCuda( cudaFree(d_cdata) );
  checkCuda( cudaFree(d_idata) );
  free(h_idata);
  free(h_tdata);
  free(h_cdata);
  free(gold);
}
```

Program7

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

```
#include <cuda_runtime.h>

__global__ void global_reduce_kernel(float * d_out, float * d_in)
{
    int myId = threadIdx.x + blockDim.x * blockIdx.x;
    int tid  = threadIdx.x;

    // do reduction in global mem
    for (unsigned int s = blockDim.x / 2; s > 0; s >>= 1)
    {
        if (tid < s)
        {
            d_in[myId] += d_in[myId + s];
        }
        __syncthreads();        // make sure all adds at one stage are
done!
    }

    // only thread 0 writes result for this block back to global mem
    if (tid == 0)
    {
        d_out[blockIdx.x] = d_in[myId];
    }
}

__global__ void shmem_reduce_kernel(float * d_out, const float * d_in)
{
    // sdata is allocated in the kernel call: 3rd arg to <<<b, t, shmem>>>
    extern __shared__ float sdata[];

    int myId = threadIdx.x + blockDim.x * blockIdx.x;
    int tid  = threadIdx.x;

    // load shared mem from global mem
    sdata[tid] = d_in[myId];
    __syncthreads();            // make sure entire block is loaded!

    // do reduction in shared mem
    for (unsigned int s = blockDim.x / 2; s > 0; s >>= 1)
    {
        if (tid < s)
        {
            sdata[tid] += sdata[tid + s];
        }
        __syncthreads();        // make sure all adds at one stage are
done!
    }

    // only thread 0 writes result for this block back to global mem
    if (tid == 0)
    {
        d_out[blockIdx.x] = sdata[0];
    }
}
```

```
void reduce(float * d_out, float * d_intermediate, float * d_in,
            int size, bool usesSharedMemory)
{
    // assumes that size is not greater than maxThreadsPerBlock^2
    // and that size is a multiple of maxThreadsPerBlock
    const int maxThreadsPerBlock = 1024;
    int threads = maxThreadsPerBlock;
    int blocks = size / maxThreadsPerBlock;
    if (usesSharedMemory)
    {
        shmem_reduce_kernel<<<blocks, threads, threads * sizeof(float)>>>
            (d_intermediate, d_in);
    }
    else
    {
        global_reduce_kernel<<<blocks, threads>>>
            (d_intermediate, d_in);
    }
    // now we're down to one block left, so reduce it
    threads = blocks; // launch one thread for each block in prev step
    blocks = 1;
    if (usesSharedMemory)
    {
        shmem_reduce_kernel<<<blocks, threads, threads * sizeof(float)>>>
            (d_out, d_intermediate);
    }
    else
    {
        global_reduce_kernel<<<blocks, threads>>>
            (d_out, d_intermediate);
    }
}

int main(int argc, char **argv)
{
    int deviceCount;
    cudaGetDeviceCount(&deviceCount);
    if (deviceCount == 0) {
        fprintf(stderr, "error: no devices supporting CUDA.\n");
        exit(EXIT_FAILURE);
    }
    int dev = 0;
    cudaSetDevice(dev);

    cudaDeviceProp devProps;
    if (cudaGetDeviceProperties(&devProps, dev) == 0)
    {
        printf("Using device %d:\n", dev);
        printf("%s; global mem: %dB; compute v%d.%d; clock: %d kHz\n",
               devProps.name, (int)devProps.totalGlobalMem,
               (int)devProps.major, (int)devProps.minor,
               (int)devProps.clockRate);
    }
```

```
const int ARRAY_SIZE = 5;
const int ARRAY_BYTES = ARRAY_SIZE * sizeof(float);

// generate the input array on the host
float h_in[ARRAY_SIZE];
float sum = 0.0f;
for(int i = 0; i < ARRAY_SIZE; i++) {
    // generate random float in [-1.0f, 1.0f]
    //h_in[i] = -1.0f + (float)random()/((float)RAND_MAX/2.0f);
    sum += h_in[i];
  h_in[i]=i;
}

// declare GPU memory pointers
float * d_in, * d_intermediate, * d_out;

// allocate GPU memory
cudaMalloc((void **) &d_in, ARRAY_BYTES);
cudaMalloc((void **) &d_intermediate, ARRAY_BYTES); // overallocated
cudaMalloc((void **) &d_out, sizeof(float));

// transfer the input array to the GPU
cudaMemcpy(d_in, h_in, ARRAY_BYTES, cudaMemcpyHostToDevice);

int whichKernel = 0;
if (argc == 2) {
    whichKernel = atoi(argv[1]);
}

cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
// launch the kernel
switch(whichKernel) {
case 0:
    printf("Running global reduce\n");
    cudaEventRecord(start, 0);
    //for (int i = 0; i < 100; i++)
    //{
        reduce(d_out, d_intermediate, d_in, ARRAY_SIZE, false);
    //}
    cudaEventRecord(stop, 0);
    break;
case 1:
    printf("Running reduce with shared mem\n");
    cudaEventRecord(start, 0);
    //for (int i = 0; i < 100; i++)
    //{
        reduce(d_out, d_intermediate, d_in, ARRAY_SIZE, true);
    //}
    cudaEventRecord(stop, 0);
    break;
default:
```

```
        fprintf(stderr, "error: ran no kernel\n");
        exit(EXIT_FAILURE);
    }
    cudaEventSynchronize(stop);
    float elapsedTime;
    cudaEventElapsedTime(&elapsedTime, start, stop);
    elapsedTime /= 100.0f;      // 100 trials

    // copy back the sum from GPU
    float h_out;
    cudaMemcpy(&h_out, d_out, sizeof(float), cudaMemcpyDeviceToHost);

    printf("average time elapsed: %f\n", elapsedTime);
      printf("The reduce sum is %f\n",h_out);
    // free GPU memory allocation
    cudaFree(d_in);
    cudaFree(d_intermediate);
    cudaFree(d_out);

    return 0;
}
```