```
//
// 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;
}
```