/\*

Parallel reduction kernels

\*/

#ifndef \_REDUCE\_KERNEL\_H\_

#define \_REDUCE\_KERNEL\_H\_

#include <stdio.h>

#ifdef \_\_DEVICE\_EMULATION\_\_

#define EMUSYNC \_\_syncthreads()

#else

#define EMUSYNC

#endif

// Utility class used to avoid linker errors with extern

// unsized shared memory arrays with templated type

template<class T>

struct SharedMemory

{

\_\_device\_\_ inline operator T\*()

{

extern \_\_shared\_\_ int \_\_smem[];

return (T\*)\_\_smem;

}

\_\_device\_\_ inline operator const T\*() const

{

extern \_\_shared\_\_ int \_\_smem[];

return (T\*)\_\_smem;

}

};

// specialize for double to avoid unaligned memory

// access compile errors

template<>

struct SharedMemory<double>

{

\_\_device\_\_ inline operator double\*()

{

extern \_\_shared\_\_ double \_\_smem\_d[];

return (double\*)\_\_smem\_d;

}

\_\_device\_\_ inline operator const double\*() const

{

extern \_\_shared\_\_ double \_\_smem\_d[];

return (double\*)\_\_smem\_d;

}

};

/\*

Parallel sum reduction using shared memory

- takes log(n) steps for n input elements

- uses n threads

- only works for power-of-2 arrays

\*/

/\* This reduction interleaves which threads are active by using the modulo

operator. This operator is very expensive on GPUs, and the interleaved

inactivity means that no whole warps are active, which is also very

inefficient \*/

template <class T>

\_\_global\_\_ void

reduce0(T \*g\_idata, T \*g\_odata, unsigned int n)

{

T \*sdata = SharedMemory<T>();

// load shared mem

unsigned int tid = threadIdx.x;

unsigned int i = blockIdx.x\*blockDim.x + threadIdx.x;

sdata[tid] = (i < n) ? g\_idata[i] : 0;

\_\_syncthreads();

// do reduction in shared mem

for(unsigned int s=1; s < blockDim.x; s \*= 2) {

// modulo arithmetic is slow!

if ((tid % (2\*s)) == 0) {

sdata[tid] += sdata[tid + s];

}

\_\_syncthreads();

}

// write result for this block to global mem

if (tid == 0) g\_odata[blockIdx.x] = sdata[0];

}

/\* This version uses contiguous threads, but its interleaved

addressing results in many shared memory bank conflicts. \*/

template <class T>

\_\_global\_\_ void

reduce1(T \*g\_idata, T \*g\_odata, unsigned int n)

{

T \*sdata = SharedMemory<T>();

// load shared mem

unsigned int tid = threadIdx.x;

unsigned int i = blockIdx.x\*blockDim.x + threadIdx.x;

sdata[tid] = (i < n) ? g\_idata[i] : 0;

\_\_syncthreads();

// do reduction in shared mem

for(unsigned int s=1; s < blockDim.x; s \*= 2)

{

int index = 2 \* s \* tid;

if (index < blockDim.x)

{

sdata[index] += sdata[index + s];

}

\_\_syncthreads();

}

// write result for this block to global mem

if (tid == 0) g\_odata[blockIdx.x] = sdata[0];

}

/\*

This version uses sequential addressing -- no divergence or bank conflicts.

\*/

template <class T>

\_\_global\_\_ void

reduce2(T \*g\_idata, T \*g\_odata, unsigned int n)

{

T \*sdata = SharedMemory<T>();

// load shared mem

unsigned int tid = threadIdx.x;

unsigned int i = blockIdx.x\*blockDim.x + threadIdx.x;

sdata[tid] = (i < n) ? g\_idata[i] : 0;

\_\_syncthreads();

// 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();

}

// write result for this block to global mem

if (tid == 0) g\_odata[blockIdx.x] = sdata[0];

}

/\*

This version uses n/2 threads --

it performs the first level of reduction when reading from global memory

\*/

template <class T>

\_\_global\_\_ void

reduce3(T \*g\_idata, T \*g\_odata, unsigned int n)

{

T \*sdata = SharedMemory<T>();

// perform first level of reduction,

// reading from global memory, writing to shared memory

unsigned int tid = threadIdx.x;

unsigned int i = blockIdx.x\*(blockDim.x\*2) + threadIdx.x;

sdata[tid] = (i < n) ? g\_idata[i] : 0;

if (i + blockDim.x < n)

sdata[tid] += g\_idata[i+blockDim.x];

\_\_syncthreads();

// 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();

}

// write result for this block to global mem

if (tid == 0) g\_odata[blockIdx.x] = sdata[0];

}

/\*

This version unrolls the last warp to avoid synchronization where it

isn't needed

\*/

template <class T, unsigned int blockSize>

\_\_global\_\_ void

reduce4(T \*g\_idata, T \*g\_odata, unsigned int n)

{

// now that we are using warp-synchronous programming (below)

// we need to declare our shared memory volatile so that the compiler

// doesn't reorder stores to it and induce incorrect behavior.

volatile T \*sdata = SharedMemory<T>();

// perform first level of reduction,

// reading from global memory, writing to shared memory

unsigned int tid = threadIdx.x;

unsigned int i = blockIdx.x\*(blockDim.x\*2) + threadIdx.x;

sdata[tid] = (i < n) ? g\_idata[i] : 0;

if (i + blockSize < n)

sdata[tid] += g\_idata[i+blockSize];

\_\_syncthreads();

// do reduction in shared mem

for(unsigned int s=blockDim.x/2; s>32; s>>=1)

{

if (tid < s)

{

sdata[tid] += sdata[tid + s];

}

\_\_syncthreads();

}

#ifndef \_\_DEVICE\_EMULATION\_\_

if (tid < 32)

#endif

{

if (blockSize >= 64) { sdata[tid] += sdata[tid + 32]; EMUSYNC; }

if (blockSize >= 32) { sdata[tid] += sdata[tid + 16]; EMUSYNC; }

if (blockSize >= 16) { sdata[tid] += sdata[tid + 8]; EMUSYNC; }

if (blockSize >= 8) { sdata[tid] += sdata[tid + 4]; EMUSYNC; }

if (blockSize >= 4) { sdata[tid] += sdata[tid + 2]; EMUSYNC; }

if (blockSize >= 2) { sdata[tid] += sdata[tid + 1]; EMUSYNC; }

}

// write result for this block to global mem

if (tid == 0) g\_odata[blockIdx.x] = sdata[0];

}

/\*

This version is completely unrolled. It uses a template parameter to achieve

optimal code for any (power of 2) number of threads. This requires a switch

statement in the host code to handle all the different thread block sizes at

compile time.

\*/

template <class T, unsigned int blockSize>

\_\_global\_\_ void

reduce5(T \*g\_idata, T \*g\_odata, unsigned int n)

{

// now that we are using warp-synchronous programming (below)

// we need to declare our shared memory volatile so that the compiler

// doesn't reorder stores to it and induce incorrect behavior.

volatile T \*sdata = SharedMemory<T>();

// perform first level of reduction,

// reading from global memory, writing to shared memory

unsigned int tid = threadIdx.x;

unsigned int i = blockIdx.x\*(blockSize\*2) + threadIdx.x;

sdata[tid] = (i < n) ? g\_idata[i] : 0;

if (i + blockSize < n)

sdata[tid] += g\_idata[i+blockSize];

\_\_syncthreads();

// do reduction in shared mem

if (blockSize >= 512) { if (tid < 256) { sdata[tid] += sdata[tid + 256]; } \_\_syncthreads(); }

if (blockSize >= 256) { if (tid < 128) { sdata[tid] += sdata[tid + 128]; } \_\_syncthreads(); }

if (blockSize >= 128) { if (tid < 64) { sdata[tid] += sdata[tid + 64]; } \_\_syncthreads(); }

#ifndef \_\_DEVICE\_EMULATION\_\_

if (tid < 32)

#endif

{

if (blockSize >= 64) { sdata[tid] += sdata[tid + 32]; EMUSYNC; }

if (blockSize >= 32) { sdata[tid] += sdata[tid + 16]; EMUSYNC; }

if (blockSize >= 16) { sdata[tid] += sdata[tid + 8]; EMUSYNC; }

if (blockSize >= 8) { sdata[tid] += sdata[tid + 4]; EMUSYNC; }

if (blockSize >= 4) { sdata[tid] += sdata[tid + 2]; EMUSYNC; }

if (blockSize >= 2) { sdata[tid] += sdata[tid + 1]; EMUSYNC; }

}

// write result for this block to global mem

if (tid == 0) g\_odata[blockIdx.x] = sdata[0];

}

/\*

This version adds multiple elements per thread sequentially. This reduces the overall

cost of the algorithm while keeping the work complexity O(n) and the step complexity O(log n).

(Brent's Theorem optimization)

\*/

template <class T, unsigned int blockSize, bool nIsPow2>

\_\_global\_\_ void

reduce6(T \*g\_idata, T \*g\_odata, unsigned int n)

{

// now that we are using warp-synchronous programming (below)

// we need to declare our shared memory volatile so that the compiler

// doesn't reorder stores to it and induce incorrect behavior.

volatile T \*sdata = SharedMemory<T>();

// perform first level of reduction,

// reading from global memory, writing to shared memory

unsigned int tid = threadIdx.x;

unsigned int i = blockIdx.x\*blockSize\*2 + threadIdx.x;

unsigned int gridSize = blockSize\*2\*gridDim.x;

T mySum = 0;

// we reduce multiple elements per thread. The number is determined by the

// number of active thread blocks (via gridDim). More blocks will result

// in a larger gridSize and therefore fewer elements per thread

while (i < n)

{

mySum += g\_idata[i];

// ensure we don't read out of bounds -- this is optimized away for powerOf2 sized arrays

if (nIsPow2 || i + blockSize < n)

mySum += g\_idata[i+blockSize];

i += gridSize;

}

// each thread puts its local sum into shared memory

sdata[tid] = mySum;

\_\_syncthreads();

// do reduction in shared mem

if (blockSize >= 512) { if (tid < 256) { sdata[tid] = mySum = mySum + sdata[tid + 256]; } \_\_syncthreads(); }

if (blockSize >= 256) { if (tid < 128) { sdata[tid] = mySum = mySum + sdata[tid + 128]; } \_\_syncthreads(); }

if (blockSize >= 128) { if (tid < 64) { sdata[tid] = mySum = mySum + sdata[tid + 64]; } \_\_syncthreads(); }

#ifndef \_\_DEVICE\_EMULATION\_\_

if (tid < 32)

#endif

{

if (blockSize >= 64) { sdata[tid] = mySum = mySum + sdata[tid + 32]; EMUSYNC; }

if (blockSize >= 32) { sdata[tid] = mySum = mySum + sdata[tid + 16]; EMUSYNC; }

if (blockSize >= 16) { sdata[tid] = mySum = mySum + sdata[tid + 8]; EMUSYNC; }

if (blockSize >= 8) { sdata[tid] = mySum = mySum + sdata[tid + 4]; EMUSYNC; }

if (blockSize >= 4) { sdata[tid] = mySum = mySum + sdata[tid + 2]; EMUSYNC; }

if (blockSize >= 2) { sdata[tid] = mySum = mySum + sdata[tid + 1]; EMUSYNC; }

}

// write result for this block to global mem

if (tid == 0)

g\_odata[blockIdx.x] = sdata[0];

}

extern "C"

bool isPow2(unsigned int x);

////////////////////////////////////////////////////////////////////////////////

// Wrapper function for kernel launch

////////////////////////////////////////////////////////////////////////////////

template <class T>

void

reduce(int size, int threads, int blocks,

int whichKernel, T \*d\_idata, T \*d\_odata)

{

dim3 dimBlock(threads, 1, 1);

dim3 dimGrid(blocks, 1, 1);

int smemSize = (threads <= 32) ? 2 \* threads \* sizeof(T) : threads \* sizeof(T);

// choose which of the optimized versions of reduction to launch

switch (whichKernel)

{

case 0:

reduce0<T><<< dimGrid, dimBlock, smemSize >>>(d\_idata, d\_odata, size);

break;

case 1:

reduce1<T><<< dimGrid, dimBlock, smemSize >>>(d\_idata, d\_odata, size);

break;

case 2:

reduce2<T><<< dimGrid, dimBlock, smemSize >>>(d\_idata, d\_odata, size);

break;

case 3:

reduce3<T><<< dimGrid, dimBlock, smemSize >>>(d\_idata, d\_odata, size);

break;

case 4:

switch (threads)

{

case 512:

reduce4<T, 512><<< dimGrid, dimBlock, smemSize >>>(d\_idata, d\_odata, size); break;

case 256:

reduce4<T, 256><<< dimGrid, dimBlock, smemSize >>>(d\_idata, d\_odata, size); break;

case 128:

reduce4<T, 128><<< dimGrid, dimBlock, smemSize >>>(d\_idata, d\_odata, size); break;

case 64:

reduce4<T, 64><<< dimGrid, dimBlock, smemSize >>>(d\_idata, d\_odata, size); break;

case 32:

reduce4<T, 32><<< dimGrid, dimBlock, smemSize >>>(d\_idata, d\_odata, size); break;

case 16:

reduce4<T, 16><<< dimGrid, dimBlock, smemSize >>>(d\_idata, d\_odata, size); break;

case 8:

reduce4<T, 8><<< dimGrid, dimBlock, smemSize >>>(d\_idata, d\_odata, size); break;

case 4:

reduce4<T, 4><<< dimGrid, dimBlock, smemSize >>>(d\_idata, d\_odata, size); break;

case 2:

reduce4<T, 2><<< dimGrid, dimBlock, smemSize >>>(d\_idata, d\_odata, size); break;

case 1:

reduce4<T, 1><<< dimGrid, dimBlock, smemSize >>>(d\_idata, d\_odata, size); break;

}

break;

case 5:

switch (threads)

{

case 512:

reduce5<T, 512><<< dimGrid, dimBlock, smemSize >>>(d\_idata, d\_odata, size); break;

case 256:

reduce5<T, 256><<< dimGrid, dimBlock, smemSize >>>(d\_idata, d\_odata, size); break;

case 128:

reduce5<T, 128><<< dimGrid, dimBlock, smemSize >>>(d\_idata, d\_odata, size); break;

case 64:

reduce5<T, 64><<< dimGrid, dimBlock, smemSize >>>(d\_idata, d\_odata, size); break;

case 32:

reduce5<T, 32><<< dimGrid, dimBlock, smemSize >>>(d\_idata, d\_odata, size); break;

case 16:

reduce5<T, 16><<< dimGrid, dimBlock, smemSize >>>(d\_idata, d\_odata, size); break;

case 8:

reduce5<T, 8><<< dimGrid, dimBlock, smemSize >>>(d\_idata, d\_odata, size); break;

case 4:

reduce5<T, 4><<< dimGrid, dimBlock, smemSize >>>(d\_idata, d\_odata, size); break;

case 2:

reduce5<T, 2><<< dimGrid, dimBlock, smemSize >>>(d\_idata, d\_odata, size); break;

case 1:

reduce5<T, 1><<< dimGrid, dimBlock, smemSize >>>(d\_idata, d\_odata, size); break;

}

break;

case 6:

default:

if (isPow2(size))

{

switch (threads)

{

case 512:

reduce6<T, 512, true><<< dimGrid, dimBlock, smemSize >>>(d\_idata, d\_odata, size); break;

case 256:

reduce6<T, 256, true><<< dimGrid, dimBlock, smemSize >>>(d\_idata, d\_odata, size); break;

case 128:

reduce6<T, 128, true><<< dimGrid, dimBlock, smemSize >>>(d\_idata, d\_odata, size); break;

case 64:

reduce6<T, 64, true><<< dimGrid, dimBlock, smemSize >>>(d\_idata, d\_odata, size); break;

case 32:

reduce6<T, 32, true><<< dimGrid, dimBlock, smemSize >>>(d\_idata, d\_odata, size); break;

case 16:

reduce6<T, 16, true><<< dimGrid, dimBlock, smemSize >>>(d\_idata, d\_odata, size); break;

case 8:

reduce6<T, 8, true><<< dimGrid, dimBlock, smemSize >>>(d\_idata, d\_odata, size); break;

case 4:

reduce6<T, 4, true><<< dimGrid, dimBlock, smemSize >>>(d\_idata, d\_odata, size); break;

case 2:

reduce6<T, 2, true><<< dimGrid, dimBlock, smemSize >>>(d\_idata, d\_odata, size); break;

case 1:

reduce6<T, 1, true><<< dimGrid, dimBlock, smemSize >>>(d\_idata, d\_odata, size); break;

}

}

else

{

switch (threads)

{

case 512:

reduce6<T, 512, false><<< dimGrid, dimBlock, smemSize >>>(d\_idata, d\_odata, size); break;

case 256:

reduce6<T, 256, false><<< dimGrid, dimBlock, smemSize >>>(d\_idata, d\_odata, size); break;

case 128:

reduce6<T, 128, false><<< dimGrid, dimBlock, smemSize >>>(d\_idata, d\_odata, size); break;

case 64:

reduce6<T, 64, false><<< dimGrid, dimBlock, smemSize >>>(d\_idata, d\_odata, size); break;

case 32:

reduce6<T, 32, false><<< dimGrid, dimBlock, smemSize >>>(d\_idata, d\_odata, size); break;

case 16:

reduce6<T, 16, false><<< dimGrid, dimBlock, smemSize >>>(d\_idata, d\_odata, size); break;

case 8:

reduce6<T, 8, false><<< dimGrid, dimBlock, smemSize >>>(d\_idata, d\_odata, size); break;

case 4:

reduce6<T, 4, false><<< dimGrid, dimBlock, smemSize >>>(d\_idata, d\_odata, size); break;

case 2:

reduce6<T, 2, false><<< dimGrid, dimBlock, smemSize >>>(d\_idata, d\_odata, size); break;

case 1:

reduce6<T, 1, false><<< dimGrid, dimBlock, smemSize >>>(d\_idata, d\_odata, size); break;

}

}

break;

}

}

// Instantiate the reduction function for 3 types

template void

reduce<int>(int size, int threads, int blocks,

int whichKernel, int \*d\_idata, int \*d\_odata);

template void

reduce<float>(int size, int threads, int blocks,

int whichKernel, float \*d\_idata, float \*d\_odata);

template void

reduce<double>(int size, int threads, int blocks,

int whichKernel, double \*d\_idata, double \*d\_odata);

#endif // #ifndef \_REDUCE\_KERNEL\_H\_

// Utilities and system includes

#include <shrUtils.h>

#include "cutil\_inline.h"

#include <algorithm>

// includes, project

#include "reduction.h"

class CPrecisionTimer

{

LARGE\_INTEGER lFreq, lStart;

public:

CPrecisionTimer()

{

QueryPerformanceFrequency(&lFreq);

}

inline void Start()

{

QueryPerformanceCounter(&lStart);

}

inline double Stop()

{

// Return duration in miliseconds...

LARGE\_INTEGER lEnd;

QueryPerformanceCounter(&lEnd);

return (double(lEnd.QuadPart - lStart.QuadPart) / lFreq.QuadPart\*1000);

}

};

enum ReduceType

{

REDUCE\_INT,

REDUCE\_FLOAT,

REDUCE\_DOUBLE

};

////////////////////////////////////////////////////////////////////////////////

// declaration, forward

template <class T>

void runTest( int argc, char\*\* argv, ReduceType datatype);

#define MAX\_BLOCK\_DIM\_SIZE 65535

#ifdef WIN32

#define strcasecmp strcmpi

#endif

extern "C"

bool isPow2(unsigned int x)

{

return ((x&(x-1))==0);

}

////////////////////////////////////////////////////////////////////////////////

// Program main

////////////////////////////////////////////////////////////////////////////////

int

main( int argc, char\*\* argv)

{

shrSetLogFileName ("reduction.txt");

shrLog("%s Starting...\n\n", argv[0]);

char \*typeChoice;

cutGetCmdLineArgumentstr( argc, (const char\*\*) argv, "type", &typeChoice);

if (0 == typeChoice)

{

typeChoice = (char\*)malloc(4 \* sizeof(char));

strcpy(typeChoice, "int");

}

ReduceType datatype = REDUCE\_INT;

if (!strcasecmp(typeChoice, "float"))

datatype = REDUCE\_FLOAT;

else if (!strcasecmp(typeChoice, "double"))

datatype = REDUCE\_DOUBLE;

else

datatype = REDUCE\_INT;

cudaDeviceProp deviceProp;

deviceProp.major = 1;

deviceProp.minor = 0;

int minimumComputeVersion = 10;

if (datatype == REDUCE\_DOUBLE)

{

deviceProp.minor = 3;

minimumComputeVersion = 13;

}

int dev;

if( cutCheckCmdLineFlag(argc, (const char\*\*)argv, "device") )

{

cutilDeviceInit(argc, argv);

cutilSafeCallNoSync(cudaGetDevice(&dev));

}

else

{

cutilSafeCallNoSync(cudaChooseDevice(&dev, &deviceProp));

}

cutilSafeCallNoSync(cudaGetDeviceProperties(&deviceProp, dev));

if((deviceProp.major \* 10 + deviceProp.minor) >= minimumComputeVersion)

{

shrLog("Using Device %d: %s\n\n", dev, deviceProp.name);

cutilSafeCallNoSync(cudaSetDevice(dev));

}

else

{

shrLog("Error: the selected device does not support compute capability %d.%d.\n\n",

minimumComputeVersion / 10, minimumComputeVersion % 10);

shrLog("FAILED");

cudaThreadExit();

cutilExit(argc, argv);

}

shrLog("Reducing array of type %s\n\n", typeChoice);

switch (datatype)

{

default:

case REDUCE\_INT:

runTest<int>( argc, argv, datatype);

break;

case REDUCE\_FLOAT:

runTest<float>( argc, argv, datatype);

break;

case REDUCE\_DOUBLE:

runTest<double>( argc, argv, datatype);

break;

}

cudaThreadExit();

shrEXIT(argc, (const char\*\*)argv);

}

////////////////////////////////////////////////////////////////////////////////

//! Compute sum reduction on CPU

//! We use Kahan summation for an accurate sum of large arrays.

//! http://en.wikipedia.org/wiki/Kahan\_summation\_algorithm

//!

//! @param data pointer to input data

//! @param size number of input data elements

////////////////////////////////////////////////////////////////////////////////

template<class T>

T reduceCPU(T \*data, int size)

{

T sum = data[0];

T c = (T)0.0;

for (int i = 1; i < size; i++)

{

T y = data[i] - c;

T t = sum + y;

c = (t - sum) - y;

sum = t;

}

return sum;

}

unsigned int nextPow2( unsigned int x ) {

--x;

x |= x >> 1;

x |= x >> 2;

x |= x >> 4;

x |= x >> 8;

x |= x >> 16;

return ++x;

}

#define MIN(x,y) ((x < y) ? x : y)

////////////////////////////////////////////////////////////////////////////////

// Compute the number of threads and blocks to use for the given reduction kernel

// For the kernels >= 3, we set threads / block to the minimum of maxThreads and

// n/2. For kernels < 3, we set to the minimum of maxThreads and n. For kernel

// 6, we observe the maximum specified number of blocks, because each thread in

// that kernel can process a variable number of elements.

////////////////////////////////////////////////////////////////////////////////

void getNumBlocksAndThreads(int whichKernel, int n, int maxBlocks, int maxThreads, int &blocks, int &threads)

{

if (whichKernel < 3)

{

threads = (n < maxThreads) ? nextPow2(n) : maxThreads;

blocks = (n + threads - 1) / threads;

}

else

{

threads = (n < maxThreads\*2) ? nextPow2((n + 1)/ 2) : maxThreads;

blocks = (n + (threads \* 2 - 1)) / (threads \* 2);

}

if (whichKernel == 6)

blocks = MIN(maxBlocks, blocks);

}

////////////////////////////////////////////////////////////////////////////////

// This function performs a reduction of the input data multiple times and

// measures the average reduction time.

////////////////////////////////////////////////////////////////////////////////

template <class T>

T benchmarkReduce(int n,

int numThreads,

int numBlocks,

int maxThreads,

int maxBlocks,

int whichKernel,

int testIterations,

bool cpuFinalReduction,

int cpuFinalThreshold,

unsigned int timer,

T\* h\_odata,

T\* d\_idata,

T\* d\_odata)

{

T gpu\_result = 0;

bool needReadBack = true;

for (int i = 0; i < testIterations; ++i)

{

gpu\_result = 0;

cudaThreadSynchronize();

cutilCheckError( cutStartTimer( timer));

// execute the kernel

reduce<T>(n, numThreads, numBlocks, whichKernel, d\_idata, d\_odata);

// check if kernel execution generated an error

cutilCheckMsg("Kernel execution failed");

if (cpuFinalReduction)

{

// sum partial sums from each block on CPU

// copy result from device to host

cutilSafeCallNoSync( cudaMemcpy( h\_odata, d\_odata, numBlocks\*sizeof(T), cudaMemcpyDeviceToHost) );

for(int i=0; i<numBlocks; i++)

{

gpu\_result += h\_odata[i];

}

needReadBack = false;

}

else

{

// sum partial block sums on GPU

int s=numBlocks;

int kernel = whichKernel;

while(s > cpuFinalThreshold)

{

int threads = 0, blocks = 0;

getNumBlocksAndThreads(kernel, s, maxBlocks, maxThreads, blocks, threads);

reduce<T>(s, threads, blocks, kernel, d\_odata, d\_odata);

if (kernel < 3)

s = (s + threads - 1) / threads;

else

s = (s + (threads\*2-1)) / (threads\*2);

}

if (s > 1)

{

// copy result from device to host

cutilSafeCallNoSync( cudaMemcpy( h\_odata, d\_odata, s \* sizeof(T), cudaMemcpyDeviceToHost) );

for(int i=0; i < s; i++)

{

gpu\_result += h\_odata[i];

}

needReadBack = false;

}

}

cudaThreadSynchronize();

cutilCheckError( cutStopTimer(timer) );

}

if (needReadBack)

{

// copy final sum from device to host

cutilSafeCallNoSync( cudaMemcpy( &gpu\_result, d\_odata, sizeof(T), cudaMemcpyDeviceToHost) );

}

return gpu\_result;

}

////////////////////////////////////////////////////////////////////////////////

// This function calls benchmarkReduce multiple times for a range of array sizes

// and prints a report in CSV (comma-separated value) format that can be used for

// generating a "shmoo" plot showing the performance for each kernel variation

// over a wide range of input sizes.

////////////////////////////////////////////////////////////////////////////////

template <class T>

void shmoo(int minN, int maxN, int maxThreads, int maxBlocks, ReduceType datatype)

{

// create random input data on CPU

unsigned int bytes = maxN \* sizeof(T);

T \*h\_idata = (T\*) malloc(bytes);

for(int i = 0; i < maxN; i++) {

// Keep the numbers small so we don't get truncation error in the sum

if (datatype == REDUCE\_INT)

h\_idata[i] = (T)(rand() & 0xFF);

else

h\_idata[i] = (rand() & 0xFF) / (T)RAND\_MAX;

}

int maxNumBlocks = MIN( maxN / maxThreads, MAX\_BLOCK\_DIM\_SIZE);

// allocate mem for the result on host side

T\* h\_odata = (T\*) malloc(maxNumBlocks\*sizeof(T));

// allocate device memory and data

T\* d\_idata = NULL;

T\* d\_odata = NULL;

cutilSafeCallNoSync( cudaMalloc((void\*\*) &d\_idata, bytes) );

cutilSafeCallNoSync( cudaMalloc((void\*\*) &d\_odata, maxNumBlocks\*sizeof(T)) );

// copy data directly to device memory

cutilSafeCallNoSync( cudaMemcpy(d\_idata, h\_idata, bytes, cudaMemcpyHostToDevice) );

cutilSafeCallNoSync( cudaMemcpy(d\_odata, h\_idata, maxNumBlocks\*sizeof(T), cudaMemcpyHostToDevice) );

// warm-up

#ifndef \_\_DEVICE\_EMULATION\_\_

for (int kernel = 0; kernel < 7; kernel++)

{

reduce<T>(maxN, maxThreads, maxNumBlocks, kernel, d\_idata, d\_odata);

}

int testIterations = 100;

#else

int testIterations = 1;

#endif

unsigned int timer = 0;

cutilCheckError( cutCreateTimer( &timer));

// print headers

shrLog("Time in milliseconds for various numbers of elements for each kernel\n\n\n");

shrLog("Kernel");

for (int i = minN; i <= maxN; i \*= 2)

{

shrLog("; %d", i);

}

for (int kernel = 0; kernel < 7; kernel++)

{

shrLog("\n%d", kernel);

for (int i = minN; i <= maxN; i \*= 2)

{

cutResetTimer(timer);

int numBlocks = 0;

int numThreads = 0;

getNumBlocksAndThreads(kernel, i, maxBlocks, maxThreads, numBlocks, numThreads);

float reduceTime;

if( numBlocks <= MAX\_BLOCK\_DIM\_SIZE ) {

benchmarkReduce(i, numThreads, numBlocks, maxThreads, maxBlocks, kernel,

testIterations, false, 1, timer, h\_odata, d\_idata, d\_odata);

reduceTime = cutGetAverageTimerValue(timer);

} else {

reduceTime = -1.0;

}

shrLog("; %.5f", reduceTime);

}

}

// cleanup

cutilCheckError(cutDeleteTimer(timer));

free(h\_idata);

free(h\_odata);

cutilSafeCallNoSync(cudaFree(d\_idata));

cutilSafeCallNoSync(cudaFree(d\_odata));

}

template <class T>

void shmooCPU(int minN, int maxN, ReduceType datatype)

{

// create random input data on CPU

unsigned int bytes = maxN \* sizeof(T);

T \*h\_idata = (T\*) malloc(bytes);

for(int i = 0; i < maxN; i++) {

// Keep the numbers small so we don't get truncation error in the sum

if (datatype == REDUCE\_INT)

h\_idata[i] = (T)(rand() & 0xFF);

else

h\_idata[i] = (rand() & 0xFF) / (T)RAND\_MAX;

}

// allocate mem for the result on host side

unsigned int timer = 0;

// print headers

shrLog("Time in milliseconds for various numbers of elements\n\n");

for (int i = minN; i <= maxN; i \*= 2)

{

shrLog("%d ;", i);

}

for (int i = minN; i <= maxN; i \*= 2)

{

CPrecisionTimer\* cputimer = new CPrecisionTimer();

cputimer->Start();

reduceCPU<T>(h\_idata, i);

double cputime = cputimer->Stop();

shrLog("; %f", cputime);

}

// cleanup

free(h\_idata);

}

////////////////////////////////////////////////////////////////////////////////

// The main function whihc runs the reduction test.

////////////////////////////////////////////////////////////////////////////////

template <class T>

void

runTest( int argc, char\*\* argv, ReduceType datatype)

{

int size = 1<<22; // number of elements to reduce

int maxThreads = 256; // number of threads per block

int whichKernel = 6;

int maxBlocks = 64;

bool cpuFinalReduction = false;

int cpuFinalThreshold = 1;

cutGetCmdLineArgumenti( argc, (const char\*\*) argv, "n", &size);

cutGetCmdLineArgumenti( argc, (const char\*\*) argv, "threads", &maxThreads);

cutGetCmdLineArgumenti( argc, (const char\*\*) argv, "kernel", &whichKernel);

cutGetCmdLineArgumenti( argc, (const char\*\*) argv, "maxblocks", &maxBlocks);

shrLog("%d elements\n", size);

shrLog("%d threads (max)\n", maxThreads);

cpuFinalReduction = (cutCheckCmdLineFlag( argc, (const char\*\*) argv, "cpufinal") == CUTTrue);

cutGetCmdLineArgumenti( argc, (const char\*\*) argv, "cputhresh", &cpuFinalThreshold);

bool runShmoo = false;

if (runShmoo)

{

shmooCPU<T>(1, 1048576, datatype);

}

else

{

// create random input data on CPU

unsigned int bytes = size \* sizeof(T);

T \*h\_idata = (T \*) malloc(bytes);

for(int i=0; i<size; i++)

{

// Keep the numbers small so we don't get truncation error in the sum

if (datatype == REDUCE\_INT)

h\_idata[i] = (T)(rand() & 0xFF);

else

h\_idata[i] = (rand() & 0xFF) / (T)RAND\_MAX;

}

int numBlocks = 0;

int numThreads = 0;

getNumBlocksAndThreads(whichKernel, size, maxBlocks, maxThreads, numBlocks, numThreads);

if (numBlocks == 1) cpuFinalThreshold = 1;

// allocate mem for the result on host side

T\* h\_odata = (T\*) malloc(numBlocks\*sizeof(T));

shrLog("%d blocks\n\n", numBlocks);

// allocate device memory and data

T\* d\_idata = NULL;

T\* d\_odata = NULL;

cutilSafeCallNoSync( cudaMalloc((void\*\*) &d\_idata, bytes) );

cutilSafeCallNoSync( cudaMalloc((void\*\*) &d\_odata, numBlocks\*sizeof(T)) );

// copy data directly to device memory

cutilSafeCallNoSync( cudaMemcpy(d\_idata, h\_idata, bytes, cudaMemcpyHostToDevice) );

cutilSafeCallNoSync( cudaMemcpy(d\_odata, h\_idata, numBlocks\*sizeof(T), cudaMemcpyHostToDevice) );

#ifndef \_\_DEVICE\_EMULATION\_\_

// warm-up

reduce<T>(size, numThreads, numBlocks, whichKernel, d\_idata, d\_odata);

int testIterations = 100;

#else

int testIterations = 1;

#endif

unsigned int timer = 0;

cutilCheckError( cutCreateTimer( &timer));

T gpu\_result = 0;

gpu\_result = benchmarkReduce<T>(size, numThreads, numBlocks, maxThreads, maxBlocks,

whichKernel, testIterations, cpuFinalReduction,

cpuFinalThreshold, timer,

h\_odata, d\_idata, d\_odata);

double reduceTime = cutGetAverageTimerValue(timer);

shrLogEx(LOGBOTH | MASTER, 0, "Reduction, Throughput = %.4f GB/s, Time = %.5f ms, Size = %u, NumDevsUsed = %d, Workgroup = %u\n",

1.0e-9 \* (size \* sizeof(int))/reduceTime, reduceTime, size, 1, numThreads);

// compute reference solution

CPrecisionTimer\* cputimer = new CPrecisionTimer();

cputimer->Start();

T cpu\_result = reduceCPU<T>(h\_idata, size);

double cputime = cputimer->Stop();

if (datatype == REDUCE\_INT)

{

shrLog("\nGPU result = %d", gpu\_result);

shrLog("\nGPU time = %.5f s \n", reduceTime);

shrLog("\nCPU result = %d", cpu\_result);

shrLog("\nCPU time = %f \n", cputime);

shrLog("%s\n\n", (gpu\_result == cpu\_result) ? "PASSED" : "FAILED");

}

else

{

shrLog("\nGPU result = %f\n", gpu\_result);

shrLog("CPU result = %f\n\n", cpu\_result);

double threshold = (datatype == REDUCE\_FLOAT) ? 1e-8 \* size : 1e-12;

double diff = abs((double)gpu\_result - (double)cpu\_result);

shrLog("%s\n\n", (diff < threshold) ? "PASSED" : "FAILED");

}

// cleanup

cutilCheckError( cutDeleteTimer(timer) );

free(h\_idata);

free(h\_odata);

cutilSafeCallNoSync(cudaFree(d\_idata));

cutilSafeCallNoSync(cudaFree(d\_odata));

}

}