Lecture 09

CUDA (III)

1

Review

- Matrix addition
  - Pinned memory
  - 2D memory layouts
- Vector scaling
  - Effect of pinned memory
  - Various optimization techniques

2

Today's Outline

- **Simple Counting**
  - Race condition
  - Atomic operation
  - Reduction operation

3

05?.cu

Simple Moving Average

4

Simple Moving Average

- Data: 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2
- Window size: 2
  - 1, 1.5, 1.5, 1.5,1.5, 1.5, …
- Windows size: 3
  - 1, 2, 4/3, 5/3, 4/3, 5/3, 4/3, 5/3, …
- Window size: 4
  - 1, 2, 1.5, 1.5, 1.5, 1.5, 1.5, …
- Most numbers in the series is referenced as many times as the window size!

Example: moving average
Vanilla C code vs. CUDA Kernel

```
void
movingAverage(const int n,
  const float *v, const int ws,
  float *out)
{
  int i, j;

  float sum;

  for(i=0;i<ws-1;i++) {
    out[i]= v[i];
  }
  for(;i<n;i++) {
    sum = 0;
    for(j=0;j<ws;j++) {
      sum += v[i-j];
    }
    out[i] = sum/ws;
  }
}
```

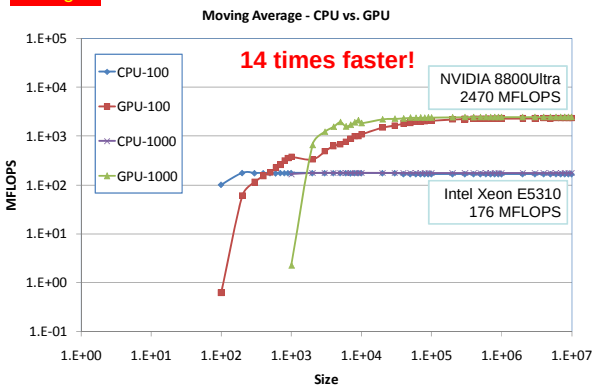
```
__global__
void mavg_device(const float n,
  const float *v, const int ws,
  float *out)
{
  int i=blockIdx.x*blockDim.x
        + threadIdx.x, j;
  float sum;

  if(i < ws-1) {
    out[i] = v[i];
  }
  else if(i < n) {
    sum = 0;
    for(j=0;j<ws;j++) {
      sum += v[i-j];
    }
    out[i] = sum/ws;
  }
}
```

Example: moving average
CUDA code - interface

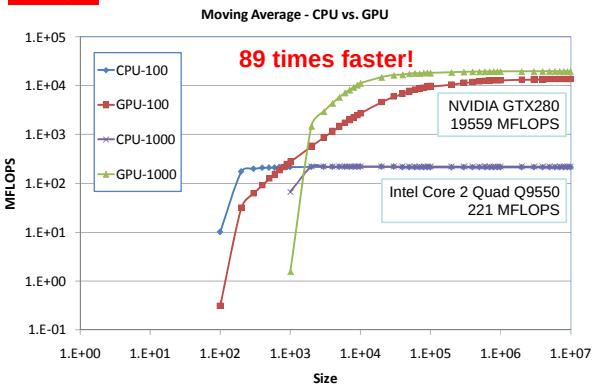
```
void movingAverage_int(const int n, const float *v, const int ws,
       float *out)
{
  float *v_d, *out_d;
  int bytes = n*sizeof(float);

  cudaMalloc((void**) &v_d, bytes);
  cudaMalloc((void**) &out_d, bytes);
  cudaMemcpy(v_d, v, n*sizeof(float), cudaMemcpyHostToDevice);

  dim3 blocks(THREAD_PER_BLOCK);
  dim3 grids (n/THREAD_PER_BLOCK+1);
  mavg_device<<<grids, blocks>>> (n, v_d, ws, out_d);

  cudaMemcpy((void*)out,(void*)out_d,bytes,cudaMemcpyDeviceToHost);
  cudaFree(v_d);
  cudaFree(out_d);
}
```

Both *v and *out are allocated using cudaMallocHost

Config #1



Config #2



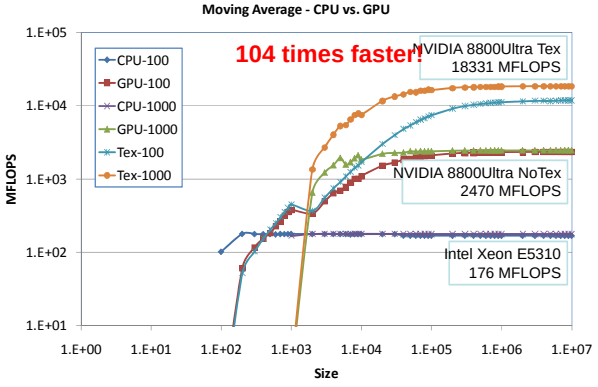
Example: moving average
Optimization #2 – use texture memory

```
void movingAverage_int(const int n, const float *v, const int ws,
       float *out)
{
  float *v_d, *out_d;
  int bytes = n * sizeof(float);

  cudaMalloc((void**) &v_d  , bytes);
  cudaMalloc((void**) &out_d, bytes);
  cudaMemcpy(v_d, v, n*sizeof(float), cudaMemcpyHostToDevice);

  cudaBindTexture(0, v_Tex, v_d, n*sizeof(float));

  dim3 blocks(THREAD_PER_BLOCK);
  dim3 grids (n/THREAD_PER_BLOCK+1);
  mavg_device<<<grids, blocks>>> (n, v_d, ws, out_d);

  cudaMemcpy((void*)out,(void*)out_d,bytes,cudaMemcpyDeviceToHost);
  cudaFree(v_d);
  cudaFree(out_d);
}
```

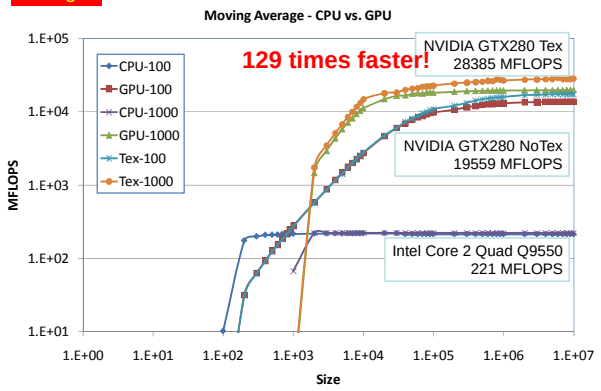
Example: moving average
Optimization #2 – use texture memory

```
texture<float, 1, cudaReadModeElementType> v_Tex;

__global__ void
mavg_device(const float n, const float *v, const int ws, float *out)
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j;
  float sum;

  if(i < ws-1) {
    out[i] = tex1Dfetch(v_Tex, i);
  }
  else if(i < n) {
    sum = 0;
    for(j=0;j<ws;j++) {
      sum += tex1Dfetch(v_Tex, i-j);
    }
    out[i] = sum/ws;
  }
}
```

Config #1



Config #2



Lesson Learned from Simple Moving Average

- Moving average is different from vector scaling in terms of "arithmetic intensity" (number of computations per memory reference)
- Moving average on CPU is bounded by memory bandwidth.
  - 176MFLOPS vs. 6.4GFLOPS theoretical peak.
- As GPU has much higher memory bandwidth, significant performance boost can be obtained!
- Use texture memory (read-only, cached) can further enhance performance. →100x faster than CPU version!
  - Note on the latest iterations of CUDA devices (e.g. Tesla C2050, GTX460, …) cache memory is introduced on the device. Texture memory may no longer provide any benefit!

06?.cu

Vector Reversal

15

Vector reversal

```
void reverse(const int n, float *a, float *b) {
  for(int i=0;i<n;i++) {
    b[i] = a[n-i-1];
  }
}
```

Is this CPU-bound or memory-bound?

```
// simple/naïve version
__global__
void reverse_dev(const int n, float *a, float *b) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if(i<n) {
    b[i] = a[n-i-1];
  }
}
```

16

Coalescing memory access

- A coordinated memory access by "half-warp" (16 threads)
  - A warp is 32 threads – the minimal size of the threads processed by a multiprocessor.
  - Threads within a block are further organized into warps to be processed by one multiprocessor.
- A contiguous region of global memory:
  - 64 bytes: each thread reads a word: int, float, …
  - 128 bytes: each thread reads a double-word: int2, float2, …
  - 256 bytes: each thread reads a quad-word: int4, float4, …
- Additional restriction:
  - Starting address for a region must be a multiple of region size
  - The k-th thread in a half-warp must access the k-th element in a block being read
- Exception: not all threads need to participate

17

Naïve vector reversal

- Results in non-coalescing memory access on CUDA 1.0 and 1.1 device!
- Non-coalescing memory access results in reduced effective memory bandwidth.
- For compute capabilities >=1.2, the rules are much relaxed. (i.e. the simple version has good performance)
- Refer to Appendix F in CUDA_C Programming Guide (4.0rc2) for such information.

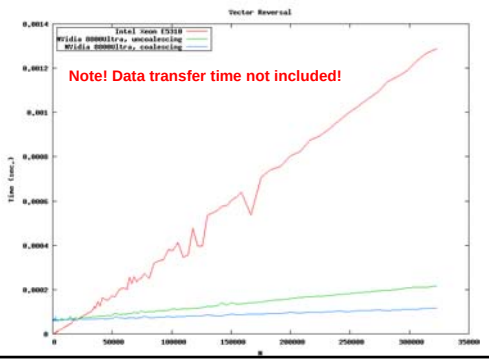
18

Vector reversal

```
// better version for CUDA 1.0 devices
__global__
void reverse_dev(const int n, float *a, float *b) {
  const int head = blockIdx.x * blockDim.x;
  __shared__ float smem[NX];
  const bool process = (head+threadIdx.x) < n;
  if(process) {
    float *from = a + n - head - blockDim.x;
    smem[blockDim.x-threadIdx.x-1]=from[threadIdx.x];
  }
  __syncthreads();
  if(process) {
    b[head+threadIdx.x] = smem[threadIdx.x];
  }
}
```

19

Performance



20

CUDA-C

new declspecs

- __shared__ float smem[NX];
  - *Declare a memory region that is shared by threads within the same block*
- Note shared memory usage have other issues known as bank conflict.
  - *Read Section 5.3.2.3 , Section F3.3 and F4.3 of the programming guide for more info.*

Variable declaration	Memory	Scope	Lifetime
`__device__ __local__ int LocalVar;`	local	thread	thread
`__device__ __shared__ int SharedVar;`	shared	block	block
`__device__ int GlobalVar;`	global	grid	application
`__device__ __constant__ int ConstantVar;`	constant	grid	Application

- *__device__ is optional when used with __local__, __shared__, or __constant__*
- *Automatic variables without any qualifier reside in a register, except arrays that reside in local memory.*

21

CUDA Profiling

```
export CUDA_PROFILE=1
export CUDA_PROFILE_CSV=1
export CUDA_PROFILE_LOG=06b.log
export CUDA_PROFILE_CONFIG=CUDA_PROFILE_CONFIG

Then run the program normally.
salloc -pCuda
srun./06b.exe 1000000
srun./06c.exe 1000000
exit
```

```
gld_incoherent
gst_incoherent
divergent_branch
warp_serialize
```

22

Profiling results

```
# CUDA_PROFILE_LOG_VERSION 2.0
# CUDA_DEVICE 0 GeForce 8800 Ultra
# CUDA_PROFILE_CSV 1
# TIMESTAMPFACTOR fffff6f628f9b160
method,gputime,cputime,occupancy,gld_incoherent,gst_incoherent,divergent_branch,warp_serialize
memcpyHtoD,1297.856,1337.000
_Z11reverse_deviPfS_,575.520,614.000,1.000,125056,0,0,0
memcpyDtoH,1258.464,1270.000
```

```
# CUDA_PROFILE_LOG_VERSION 2.0
# CUDA_DEVICE 0 GeForce 8800 Ultra
# CUDA_PROFILE_CSV 1
# TIMESTAMPFACTOR fffff6f6b209bab8
method,gputime,cputime,occupancy,gld_incoherent,gst_incoherent,divergent_branch,warp_serialize
memcpyHtoD,1297.824,1322.000
_Z11reverse_devmPfS_,222.176,274.000,1.000,0,0,0,0
memcpyDtoH,1257.568,1270.000
```

23

08?.cu

Simple Counting ?!

24

08a.cpp

```
int main(int argc, char **argv) {
  if(argc!=2) {
    cerr << argv[0] << " [n]";
    return 255;
  }
  const int n = atoi(argv[1]);
  int *a = new int[n];
  for(int i=0;i<n;i++) a[i] = rand();
  int ans = count(n, a);
  cout << ans << endl;

  delete []a;
  return 0;
}
```

```
int count(int n, int *a) {
  int ans=0;
  for(int i=0;i<n;i++) {
    if(a[i] % 3 == 0) ans++;
  }
  return ans;
}
```

08b.cu

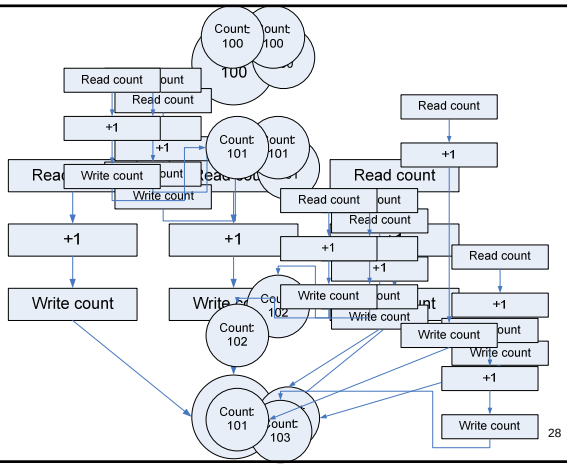
```
int count(int n, int *a) {
  int ans=0, *b;
  const int nThreads=512;
  cudaMalloc(&b, (n+1)*sizeof(int));
  cudaMemcpy(b+1,a, n*sizeof(int), cudaMemcpyHostToDevice);
  cudaMemcpy(b,&ans,1*sizeof(int), cudaMemcpyHostToDevice);
  kernel<<<(n+nThreads-1)/nThreads, nThreads>>> (n, b);
  cudaMemcpy(&ans,b,1*sizeof(int), cudaMemcpyDeviceToHost);
  cudaFree(b);
  return ans;
}
```

```
__global__
void kernel(int n, int *a) {
  int i=blockIdx.x*blockDim.x+threadIdx.x;
  if(i<n) {
    if(a[i+1] % 3 == 0) a[0]++;
  }
}
```

Race condition

- 08b.cu, the rather straightforward translation gives wrong results due to **race condition**.
- **Race condition**: A race condition is a flaw in a system or process whereby the output of the process is unexpectedly and critically dependent on the sequence or timing of other events. (Wikipedia).
- Race condition often occurs on parallel programs when multiple threads / processes try to modify the same variable.

27



28

08b.cu

- Source of the problem:
  read – modify – write on a[0] variable in 08b.cu
- Solution: coordinated access on a[0].
- Atomic operation: Atomic operations ensure only one thread can update a variable at a time. (See Sec. B.11 in Programming Guide).
  - Compute capability 1.1+: atomic operation in global memory
  - Compute capability 1.2+: atomic operation in shared memory
- In CUDA, most atomic operations only work for integer data. Other data types can be accomplished using atomicCAS function.

29

Atomic operation in CUDA

- Arithmetic functions
  - atomicAdd, atomicSub, atomicExch, atomicMin, atomicMax, atomicInc, atomicDec, atomicCAS (compare and swap)
- Bitwise functions
  - atomicAnd, atomicOr, atomicXor

30

08c.cu

```
__global__
void kernel(int n, int *a) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if(i<n) {
    if(a[i+1] % 3 == 0) atomicAdd(a, 1);
  }
}
```

Note: atomicInc(a, 1) gives wrong result! atomicInc has the following prototype: int atomicInc(int *address, int val);
It evaluates **((old >= val) ? 0 : (old+1))**, and returns old.

08d.cu

- Atomic operation is one solution, but it requires compute capability 1.1+. Also, atomic operation synchronizes many threads. (Synchronization usually means slow in parallel programming).
- Another solution, is to do a reduction operation.
- Recall MPI_Reduce

Thrust

- Thrust is a CUDA library of parallel algorithms with an interface resembling the C++ Standard Template Library (STL).
- This library is included since CUDA SDK 4.0 (currently 4.2).
- Provides algorithms such as searching, copying, reductions, reordering (partitioning, stream compaction), sorting, …

http://code.google.com/p/thrust/

08d.cu

```
__global__
void kernel(int n, int *a) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if(i<n) {
    a[i] = (a[i]%3==0);
  }
}
```


08d.cu

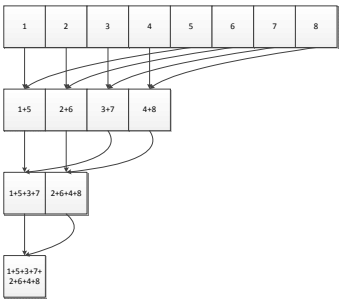
```
#include "thrust/reduce.h"
#include "thrust/device_ptr.h"
int count(int n, int *a) {
  const int nThreads=512;
  int *b;
  cudaMalloc(&b, n*sizeof(int));
  cudaMemcpy(b,a,n*sizeof(int),cudaMemcpyHostToDevice);
  kernel<<<(n+nThreads-1)/nThreads,nThreads>>>(n,b);
  thrust::device_ptr<int> vec(b);
  int ans = thrust::reduce(vec,vec+n,(int)0,
                thrust::plus<int>());
  cudaFree(b);
  return ans;
}
```


08e.cu

```
int count(int n, int *a) {
  int ans=0, *b;
  const int nThreads=512;

  cudaMalloc(&b, n*sizeof(int));
  cudaMemcpy(b, a, n*sizeof(int), cudaMemcpyHostToDevice);

  kernel<<< (n+nThreads-1)/nThreads, nThreads >>> (n, b);
  doReduction(n, b);

  cudaMemcpy(&ans, b, 1*sizeof(int), cudaMemcpyDeviceToHost);
  cudaFree(b);

  return ans;
}
```

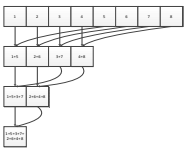
08e.cu

- A very simple reduction implemented using CUDA.



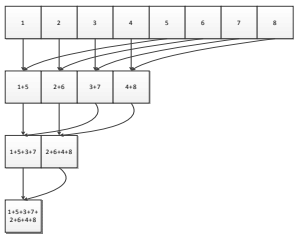
08e.cu

```
void doReduction(int n, int *data) {
  int nThreads=32;
  int q = n;
  int p = (q+1)>>1; // (n+1) / 2

  while(q>nThreads) {
    int nBlocks=(p+nThreads-1)/nThreads;
    reductionKernel <<<nBlocks, nThreads>>> (p, q, data);
    cudaThreadSynchronize();
    q=p;
    p=(q+1)>>1;
  }

  finalReduction<<<1, q, q*sizeof(int)>>> (data);
}
```



08e.cu

```
__global__
void reductionKernel(int p, int q, int *data) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if(i+p<q) data[i] += data[i+p];
}
```



08e.cu

```
__global__
void finalReduction(int *data) {
  int i = threadIdx.x;
  extern __shared__  int shmem[];
  shmem[i] = data[i];
  __syncthreads();
  if(threadIdx.x==0) {
    for(int j=1;j<blockDim.x;j++) shmem[0] += shmem[j];
    data[0] = shmem[0];
  }
}
```

```
Invoked by:  finalReduction<<<1, q, q*sizeof(int)>>> (data);
```

Demonstrates the use of shared memory
It is necessary to __syncthreads() to ensure all data are loaded into shmem[]
Use one thread in the thread block to sum data stored in the shared memory

__syncthreads

- void __syncthreads()
  - *waits until all threads in the thread block have reached this point and all global and shared memory accesses made by these threads prior to* ***__syncthreads()*** *are visible to all threads in the block.*
  - *That means some processors are forced idle to wait for others.*
  - *Used in this example to make sure the data have be saved into the shared memory.*
  - ***__syncthreads()*** *is allowed in conditional code (if, switch, …) but only if the conditional evaluates* ***identically*** *across the entire thread block, otherwise the code execution is likely to hang or produce unintended side effects.*

Summary

- Simple counting isn't that simple in parallel computing.
- Race condition occurs when multiple threads try to update a shared variable.
- Atomic operation can be one solution to cure race condition. But synchronized access to one shared variable serializes multiple threads and reduces performance.
- Reduction is a common pattern in parallel computing. It should be noted the computed result may be different than the serial version due to different order of operations.

Assignment #6

Due: 5/15/2012

Assignment #6

This program will continue your saga to find collision pairs of spheres.
You are to write programs to:

1) Get a user-specified N, D, Seed from command line arguments.
2) On host, specify random seed (Seed) using srand(), and then generate N random spheres (3 coordinate components + radius), centers of these spheres should be in the region of [-D:D, -D:D, -D:D] and each sphere is numbered incrementally starting from 0. (of course, you need to use dynamic arrays to store all the data!)
3) Write a global function that takes arguments 1) N, the number of spheres, and 2) data for spheres. (i.e. x, y, z, r). This function will then launch a kernel to count the number of pairs of spheres that collides.
   - **There are several challenges on writing this function (to be discussed in the next lecture!) Better start working on this one early!**

Discuss

1. Discuss the efficiency of storing N sphere data using various data layout strategies. Which of the following gives the best performance?
   1. Option 1: double xyz[N][3], r[N];
   2. Option 2: double xyzr[N][4];
   3. Option 3: double x[N], y[N], z[N], r[N];
   4. You may also explorer other possibilities (e.g. using struct & class)
2. Is CUDA version faster or slower than the CPU version? How much faster/slower is your CUDA version than the pure CPU version?

Note

- All discussions need to consider different program sizes (e.g. N=10, 50, 100, 200, 500, 1000, 2000, 5000, 10000, …)
- All discussions should be facilitated by charts, and then tell stories from these charts.
- All your work must reside in folder HW06 in your home directory in our cluster system (ssh into 140.118.5.6:222)
- All your programs must be made by simply typing make in HW06 folder (i.e. you must write a Makefile)
- For evaluating performance or efficiency, use the timing class that is provided to you:
  - /home/courses/stopWatch.h, /home/courses/stopWatch.o