Lecture 11

CUDA (III)

1

Review

- Matrix addition
  - Pinned memory
  - 2D memory layouts
- Vector scaling
  - Effect of pinned memory
  - Various optimization techniques
- Simple Moving Average
  - Arithmetic Intensity
  - Use of texture memory

2

Today's Outline

- **Simple Counting**
  - Race condition
  - Atomic operation
  - Reduction operation
- **Vector Reversal**
  - Coalescing memory access
  - Use of shared memory for coalescing memory access
  - Profiling

3

08?.cu

Simple Counting ?!

4

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;
}
```

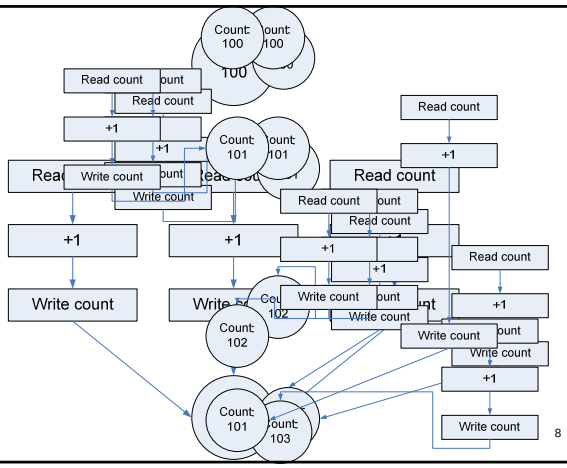
08b.cu

```
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);
  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.



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.

Atomic operation in CUDA

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

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 will be included in CUDA SDK 4.0 (currently RC2, not official yet!)
- 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

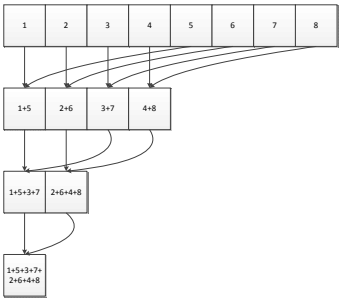
```
#include "thrust/reduce.h"
#include "thrust/device_ptr.h"
int count(int n, int *a) {
  const int nThreads=512;
  int *b;
  cudaMalloc(&b, n*sizeof(int));
  cudaMemcpy(b,a,n*sizeof(int),cudaMemcpyHostToDevice);
  kernel<<<(n+nThreads-1)/nThreads,nThreads>>>(n,b);
  thrust::device_ptr<int> vec(b);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


08e.cu

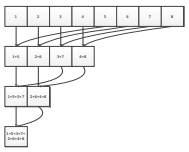
```
int count(int n, int *a) {
  int ans=0, *b;
  const int nThreads=512;

  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.



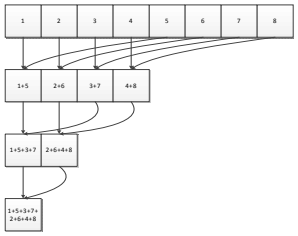
08e.cu



```
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.

06?.cu

Vector Reversal

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];
  }
}
```

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

25

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.

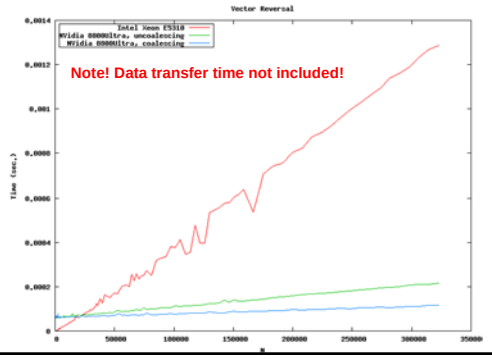
26

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];
  }
}
```

27

Performance



28

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.*

29

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
```

30

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
```

Upcoming

- Lectures
  - One more set of CUDA examples
    - Asynchronous operations
    - CPU + GPU computation
    - Using multiple CUDA devices
  - OpenMP (3 weeks)
  - Parallel Sorting
- Assignments, Exams
  - Term Project Parallel Version due (6/7/2011)
  - Discussions on Assignments (**When**?)
  - Final exam (6/21/2011)
  - Final presentation (6/28/2011),

32