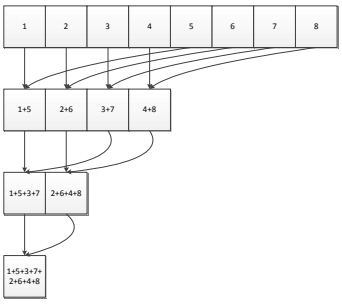
08e.cu

```
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;
}
```

1

08e.cu

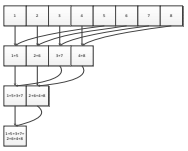
- A very simple reduction implemented using CUDA.



2

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);
}
```



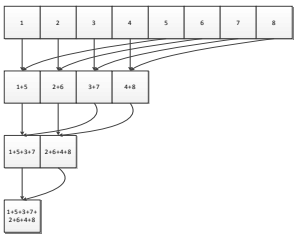
p: # of working theads
q: # of elements to be reduced.
nThreads: # of elements to be processed by finalReduction()

This is the host code controlling the overall reduction process.

3

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];
}
```



4

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];
  }
}
```

This function reduces the final few elements (<32) by using one thread.

```
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

5

__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.*

6

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.
- Check out /opt/cuda/sdk/C/src/reduction for more in-depth discussions in efficient reduction in CUDA.

7

07?.cu

Summing Matrix^2

8

Overall computation

Randomly generate n matrices of size s-by-s.

$a_1, a_2, a_3, a_4, \ldots a_n$

ans={0};

for i in 1, 2, 3, … n

ans += $a_i * a_i^T$

end for

9

07a.cpp

```
void gemv(const size_t s, DT *a, DT *out) {
    for(size_t i=0;i<s;i++) {
        for(size_t j=0;j<s;j++) {
            for(size_t k=0;k<s;k++) {
                out[i*s+j] += a[i*s+k] * a[k*s+j];
            }
        }
    }
}
void addMatrices(const size_t n, const size_t s, DT
**a, DT *out) {
    for(size_t i=0;i<n;i++) {
        gemv(s, a[i], out);
    }
}
```

10

07b.cu

```
void addMatricesGPU(const size_t n, const size_t s, DT **a, DT
*out) {
  DT *tmp, *ans;
  size_t bytes = s*s*sizeof(DT);
  cudaMalloc(&tmp, bytes);
  cudaMalloc(&ans, bytes);
  cudaMemset(ans, 0, bytes);
  const int threads=16;
  dim3 blocks(threads, threads);
  dim3 grids((s+threads-1)/threads,(s+threads-1)/threads);
  for(size_t i=0;i<n;i++) {
    cudaMemcpy(tmp, a[i], bytes, cudaMemcpyHostToDevice);
    addKernel<<<grids, blocks>>>(s, tmp, ans);
  }
  cudaMemcpy(out, ans, bytes, cudaMemcpyDeviceToHost);
  cudaFree(tmp);
  cudaFree(ans);
}
```

```
for(size_t i=0;i<n;i++) {
    gemv(s, a[i], out);
}
```

11

07b.cu

```
__global__
void addKernel(size_t s, DT *a, DT *ans) {
  size_t i = blockIdx.y*blockDim.y + threadIdx.y;
  size_t j = blockIdx.x*blockDim.x + threadIdx.x;
  if(i<s && j<s) {
    for(size_t k=0;k<s;k++) {
      ans[i*s+j] += a[i*s+k] * a[k*s+j];
    }
  }
}
```

```
void gemv(const size_t s, DT *a, DT *out) {
  for(size_t i=0;i<s;i++) {
    for(size_t j=0;j<s;j++) {
      for(size_t k=0;k<s;k++) {
        out[i*s+j] += a[i*s+k] * a[k*s+j];
      }
    }
  }
}
```

Heterogeneous Computing

- All previous examples are done either using CPU or using GPU.
- When GPU does it work, CPU does nothing, and vice versa.
- Most CUDA APIs are now non-blocking / asynchronous:
  - Kernel launches
  - Functions for copying memory and are suffixed with Async
  - Functions performing device ←→ device memory copies
  - Functions set memory (memset)
- CUDA introduced so called streams, which defines series of "commands" to be issued to CUDA devices. These commands are issued with very little delay to CPU (asynchronously, non-blocking).
- By using asynchronous APIs, we can make CPU and GPU work at the same time.

13

07c.cu

```
void addMatricesGPU(const size_t n, const size_t s, DT
**a, DT *out) {
  DT *tmp, *tmp1, *ans;
  size_t i, bytes = s*s*sizeof(DT);
  cudaMalloc(&tmp, bytes);
  cudaMalloc(&ans, bytes);
  cudaMallocHost(&tmp1, bytes);
```

14

07c.cu

```
// Define work to be done at GPU
const double ratio = 3.0/4;
const int threads=16;
dim3 blocks(threads, threads);
dim3 grids( (s+threads-1)/threads, (s+threads-1)/threads);
cudaStream_t stream;
cudaStreamCreate(&stream);
cudaMemsetAsync(ans, 0, bytes, stream);
for(i=0;i<n*ratio;i++) {
  cudaMemcpyAsync(tmp,a[i],bytes,cudaMemcpyHostToDevice,stream);
  addKernel<<<grids, blocks, 0, stream>>>(s, tmp, ans);
}
cudaMemcpyAsync(tmp1,ans,bytes,cudaMemcpyDeviceToHost,stream);

// CPU does rest of the work
for(;i<n;i++) {
  gemv(s, a[i], out);
}
```

15

07c.cu

```
  // Merge results
  cudaStreamSynchronize(stream);
  for(i=0;i<bytes;i++) {
    out[i] += tmp1[i];
  }

  cudaFree(tmp);
  cudaFree(ans);
  cudaFree(tmp1);
  cudaStreamDestroy(stream);
}
```

16

Asynchronous Operations

- Concurrent data transfer & kernel launch (limited)
  - Compute capability 1.1+
- Concurrent heterogeneous kernel launches
  - Compute capability 2.0+, 4 max
- Concurrent data transfers (uploads & downloads)
  - Compute capability 2.0+
- Streams
  - A sequence of commands that execute in order.
  - Different streams, on the other hand, may execute their commands out of order with respect to one another or concurrently; this behavior is not guaranteed and should therefore not be relied upon for correctness (e.g. inter-kernel communication is undefined).
- Events
  - For timing purposes

17

Multiple CUDA Devices

- It is possible to use multiple CUDA devices at the same time on one host.
  - cudaGetDeviceCount(&deviceCount);
  - cudaSetDevice(device);

18

07d.cu

```
void addMatricesGPU(const size_t n, const size_t s, DT **a, DT
*out) {
  int deviceCount;
  cudaGetDeviceCount(&deviceCount);
  addMatricesGPUn(n, s, a, out, deviceCount);
}

void addMatricesGPUn(const size_t n, const size_t s, DT **a,
DT* const out, const int deviceCount=1) {
  DT *tmp[deviceCount], *ans[deviceCount];
  DT *tmpAns[deviceCount];
  const size_t bytes = s*s*sizeof(DT);
  const int threads=16;
  dim3 blocks(threads, threads);
  dim3 grids( (s+threads-1)/threads, (s+threads-1)/threads);
```


07d.cu

```
omp_set_num_threads(deviceCount);
#pragma omp parallel for
for(int device=0;device<deviceCount;device++) {
  cudaSetDevice(device);
  if(device!=0) cudaMallocHost(&tmpAns[device], bytes);
  else          tmpAns[0] = out;
  cudaMalloc(&tmp[device], bytes);
  cudaMalloc(&ans[device], bytes);
  cudaMemset(ans[device], 0, bytes);
  for(size_t i=device;i<n;i+=deviceCount) {
    cudaMemcpy(tmp[device],a[i],bytes,cudaMemcpyHostToDevice);
    addKernel<<<grids, blocks>>>(s, tmp[device], ans[device]);
  }
  cudaMemcpy(tmpAns[device],ans[device],bytes, cudaMemcpyDeviceToHost);
  cudaFree(tmp[device]); cudaFree(ans[device]);
  cudaThreadSynchronize();
}
```


07d.cu

```
  // merge results from multiple GPUs on CPU
  for(size_t device=1;device<deviceCount;device++) {
    for(size_t i=0;i<s*s;i++) {
      out[i] += tmpAns[device][i];
    }
    cudaFree(tmpAns[device]);
  }
}
```


Summary

- Devices → running multiple threads arranged in blocks, which are arranged in grid. The arrangement of blocks and grids replaces for-loop in conventional programming.
- Threads run kernels. GPU requires running multiple threads to be advantageous.
- Memory transfer operations are essential to perform useful tasks on GPU and obtain results calculated on GPU back to CPU for data storage and display.
- Need to learn to use different types of memory and need to be aware of efficient memory access patterns required for different devices.
- The last example uses OpenMP to generate multiple threads, each threads works with one CUDA device.

Progress Report