Exercise Session 12

Data Processing on Modern Hardware
263-3502-00L — Fall Semester 2012

Cagri Balkesen
cagri.balkesen@inf.ethz.ch

Department of Computer Science ETH Zurich, Switzerland

13 December 2012

Execution Time CPU-only

You had to implement the following query on your GPU:

```
SELECT SUM(quantity * extendedprice)
FROM   lineitem
WHERE  suppkey<Z
```

- Data set: 6,001,215 rows (scale factor = 1)
- Experiment on : Intel(R) Core(TM) i5 CPU 750@2.67GHz
- Selection : Z = 30
- Reference CPU-only implementation: **5.782 ms**
- Now we will refine GPU solution step-by-step

Selection Kernel (simpleselect)

```
__global__ void
simpleselect(uint32_t *suppkey, int64_t *quantity,
    int64_t *extendedprice, int64_t *agg_data, uint32_t numrows,
    int32_t Z) {
        int idx = blockIdx.x * blockDim.x + threadIdx.x;
        agg_data[idx] = 0;
        if(idx < numrows) {
            if(suppkey[idx] < Z) {
                agg_data[idx] = quantity[idx]*extendedprice[idx];
            }
        }
}
```

- Only does selection and multiplication on the GPU
- Entire sum is computed on the CPU

Invoking the Kernel

```
cudaMalloc((void**)&suppkey_d, numrows*sizeof(uint32_t));
...
cudaMemcpy(suppkey_d, suppkey_h, numrows*sizeof(uint32_t),
cudaMemcpyHostToDevice);
...
int blocksize = 256;
int numblocks = (numrows+blocksize-1)/blocksize;
dim3 dimgrid(numblocks);
dim3 dimblock(blocksize);
simpleselect<<<dimgrid,dimblock>>>(suppkey_d, quantity_d,
    extendedprice_d, agg_data_d, numrows, Z);
cudaThreadSynchronize();

cudaMemcpy(agg_data_h, agg_data_d, numrows*sizeof(int64_t),
    cudaMemcpyDeviceToHost);
agg_result = 0;
for (int i=0; i<numrows; i++) {
    agg_result += agg_data_h[i];
}
```

Performance Measurements (`simpleselect`)

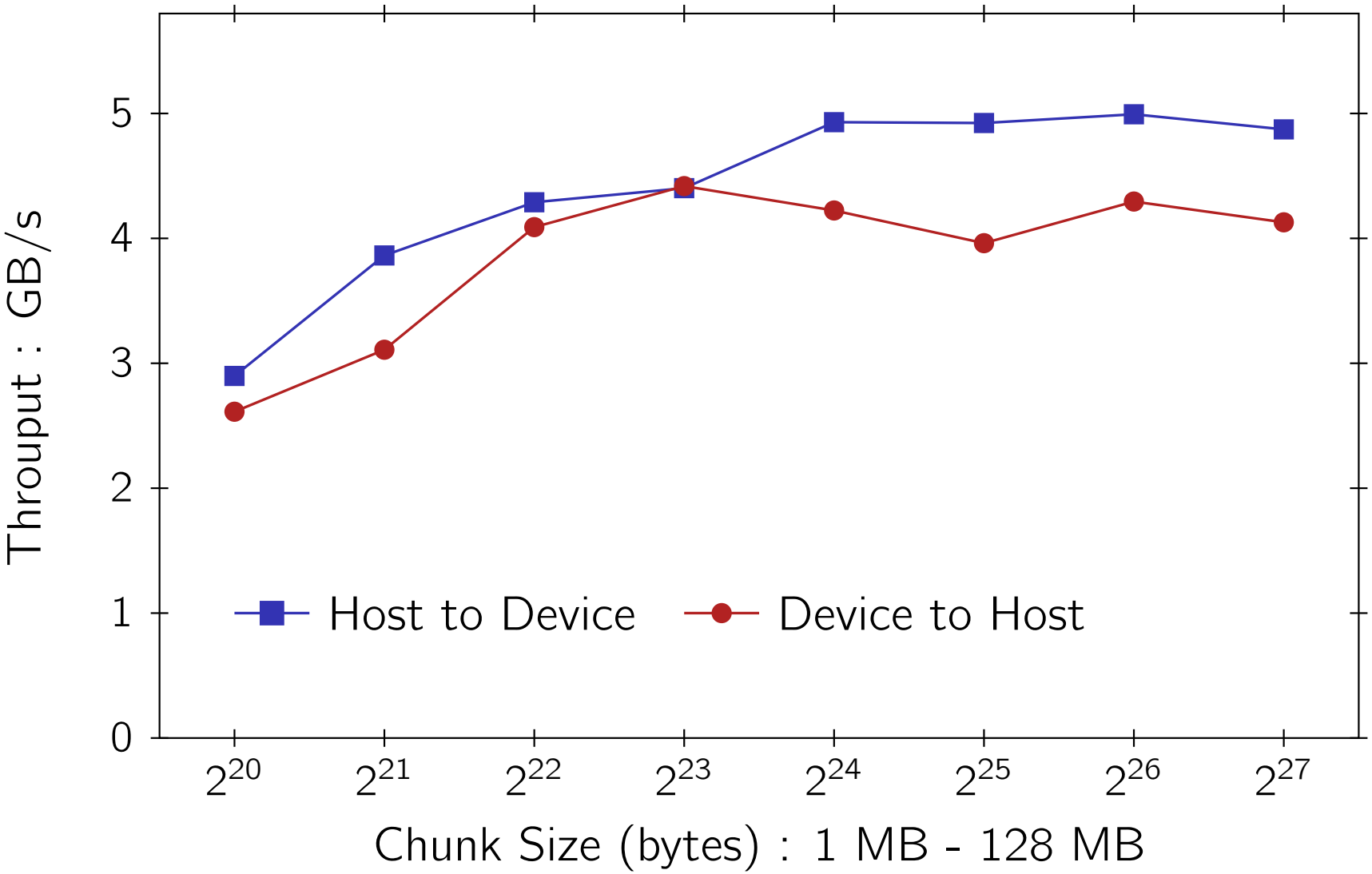
- Reference CPU-only implementation: **5.782 ms**
- GPU Experiments on : GeForce 9500 GT
- 4 (MP) x 8 (Cores/MP) = 32 CUDA Cores
- Total execution time = **59.45 ms** ↯
- cudaMemcpy overhead = **46.335 ms** ↯
- Kernel execution and sum on CPU = **13.115 ms** ↯

Bandwidth Test

```
cudaMalloc : 48 MB
Execution time: 38.009000 ms
cudaMemcpy (HostToDevice) : 48 MB
Execution time: 8.943000 ms
Throughput: 5.37 GB/s
cudaMemcpy (DeviceToHost) : 48 MB
Execution time: 13.957000 ms
Throughput: 3.4 GB/s
```

Alternative:
/NVIDI_GPU_Computing_SDK/C/bin/linux/release/bandwidthTest

./bandwidthTest



First Reduction Step on GPU (selectandsum)

```
__global__ void
selectandsum(uint32_t *suppkey, int64_t *quantity, int64_t *extendedprice,
        int64_t *agg_data, uint32_t numrows, int32_t Z) {
  extern __shared__ int64_t sagg[];
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  /* evaluate predicate */
  if (idx < numrows && suppkey[idx] < Z) {
    sagg[threadIdx.x] = quantity[idx]*extendedprice[idx];
  } else {
    sagg[threadIdx.x] = 0;
  }
   __syncthreads();
  /* compute sum (1st level of reduction) */
  for (unsigned int s=1; s<blockDim.x; s*=2) {
      int i = 2*s*threadIdx.x;
      if (i < blockDim.x)
          sagg[i] += sagg[i+s];
      __syncthreads();
  }
    if (threadIdx.x == 0)
      agg_data[blockIdx.x] = sagg[0];
}
```

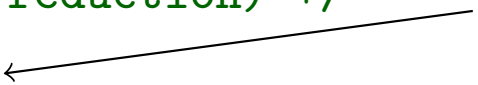
Performance Measurements (`selectandsum`)

- Reference CPU-only implementation: **5.782 ms**
- Total execution time = **60.126 ms** ↯
- cudaMemcpy overhead = **24.080 ms** ✓
  - [cudaprof] Before : Device → Host = 21.380 ms
  - [cudaprof] Now : Device → Host = 0.131 ms ✓
- Kernel execution and sum on CPU = **36.046 ms** ↯
  - [cudaprof] warp serialize : 6,223,004

Optimizing Bank Conflicts (selectandsumopt1)

- Get rid of strided access to shared memory

```
/* compute sum (1st level of reduction) */
__syncthreads();
for (unsigned int s=blockDim.x/2; s>0; s>>=1) {
  if (threadIdx.x < s) {
    sagg[threadIdx.x] += sagg[threadIdx.x+s];
  }
  __syncthreads();
}
```



blocksize needs to be a power of 2

Performance Measurements (`selectandsumopt1`)

- Reference CPU-only implementation: **5.782 ms**
- Total execution time = **40.892 ms** ✓
- cudaMemcpy overhead = 24.128 ms (unchanged)
- Kernel execution and remaining sum on CPU
  - Before : **36.046 ms** ↯
  - Now : **16.764 ms** ✓
- warp serialize
  - [cudaprof] Before : 6,223,004
  - [cudaprof] Now : 993,635

Loop unrolling (`selectandsumopt2`)

- Bottleneck : instruction overhead → addr. arithmetic, loop overhead
- Solution : unroll last 6 iterations
  - when s <= 32, we have only one warp left
  - no need for `__syncthreads()`
  - no need for `if(i < blockDim.x)`

```
for (unsigned int s=blockDim.x/2; s>32; s>>=1) {
  ...
}
if (threadIdx.x < 32) {
  sagg[threadIdx.x] += sagg[threadIdx.x+32];
  sagg[threadIdx.x] += sagg[threadIdx.x+16];
  sagg[threadIdx.x] += sagg[threadIdx.x+8];
  sagg[threadIdx.x] += sagg[threadIdx.x+4];
  sagg[threadIdx.x] += sagg[threadIdx.x+2];
  sagg[threadIdx.x] += sagg[threadIdx.x+1];
}
```

Performance Measurements (`selectandsumopt2`)

- Reference CPU-only implementation: **5.782 ms**
- Total execution time = **35.71 ms** ✓
- Kernel execution and remaining sum on CPU
  - Before : **16.764 ms** ↯
  - Now : **12.939 ms** ✓
- Number of instructions
  - [cudaprof] Before : 53,548,768 ↯
  - [cudaprof] Now : 3,283,837 ✓

Idle Threads (selectandsumopt3)

```
for (unsigned int s=blockDim.x/2; s>0; s>>=1) {
  if (threadIdx.x < s) {
    sagg[threadIdx.x] += sagg[threadIdx.x+s];
  }
...
```

- Half of the threads are idle on first loop iteration!
- Solution : halve number of blocks and instead of one load do two loads and first add of reduction.

```
 int idx = blockIdx.x*blockDim.x*2+threadIdx.x;
  if (idx < numrows && suppkey[idx] < Z) {
  sagg[threadIdx.x] = quantity[idx]*extendedprice[idx];
} else {
  sagg[threadIdx.x] = 0;
}
  if ((idx+blockDim.x) < numrows && suppkey[idx+blockDim.x] < Z) {
  sagg[threadIdx.x] += quantity[idx+blockDim.x]*
                       extendedprice[idx+blockDim.x];
}
```

Performance Measurements (`selectandsumopt3`)

- Reference CPU-only implementation: **5.782 ms**
- Total execution time = **30.808 ms** ✓
- Kernel execution and remaining sum on CPU
  - Before : **12.93 ms** ↯
  - Now : **8.181 ms** ✓

More Reduction on GPU (reduce)

- We are still computing sum of 11,722 ((6001215+1)÷256÷2) values on CPU.
- After invoking the reduction kernel there will be only 23 values left!
- Kernel invocation:

```
int numaggregates = numblocks;
numblocks = (numblocks+2*blocksize-1)/(2*blocksize);
dim3 reducedimgrid(numblocks);
reduce<<<reducedimgrid,dimblock,blocksize*sizeof(int64_t)>>>
    (agg_data_d,numaggregates);
cudaCheckError("Kernel Execution Failed!");
cudaThreadSynchronize();
```

Performance Measurements (`reduce`))

- Reference CPU-only implementation: **5.782 ms**
- Total execution time = **31.091 ms** ↯
- Kernel(s) execution and remaining sum on CPU
  - Before : **8.181 ms**
  - Now : **9.116 ms** ↯
- Overhead of calling the kernel for further reduction is not worth the overhead!

Optimizing Occupancy

- selectandsumopt3 (fastest) → 2/3 occupancy
  - Registers : 14 per thread
  - Shared memory : (48 + 16) + 256*8 = 2,112 bytes
  - Block size = 256
  - Capability = 1.1
- Occupancy calculator → bound by registers
- Solution: compile with `--maxrregcount 10` flag
- Occupancy calculator → optimal block size = 128/256/384
  - Occupancy = 1
- Total execution time = **28.609 ms** ✓
- Kernel(s) execution and remaining sum on CPU
  - Before : **8.181 ms**
  - Now : **5.993 ms** ✓

Optimizing Data Transfer

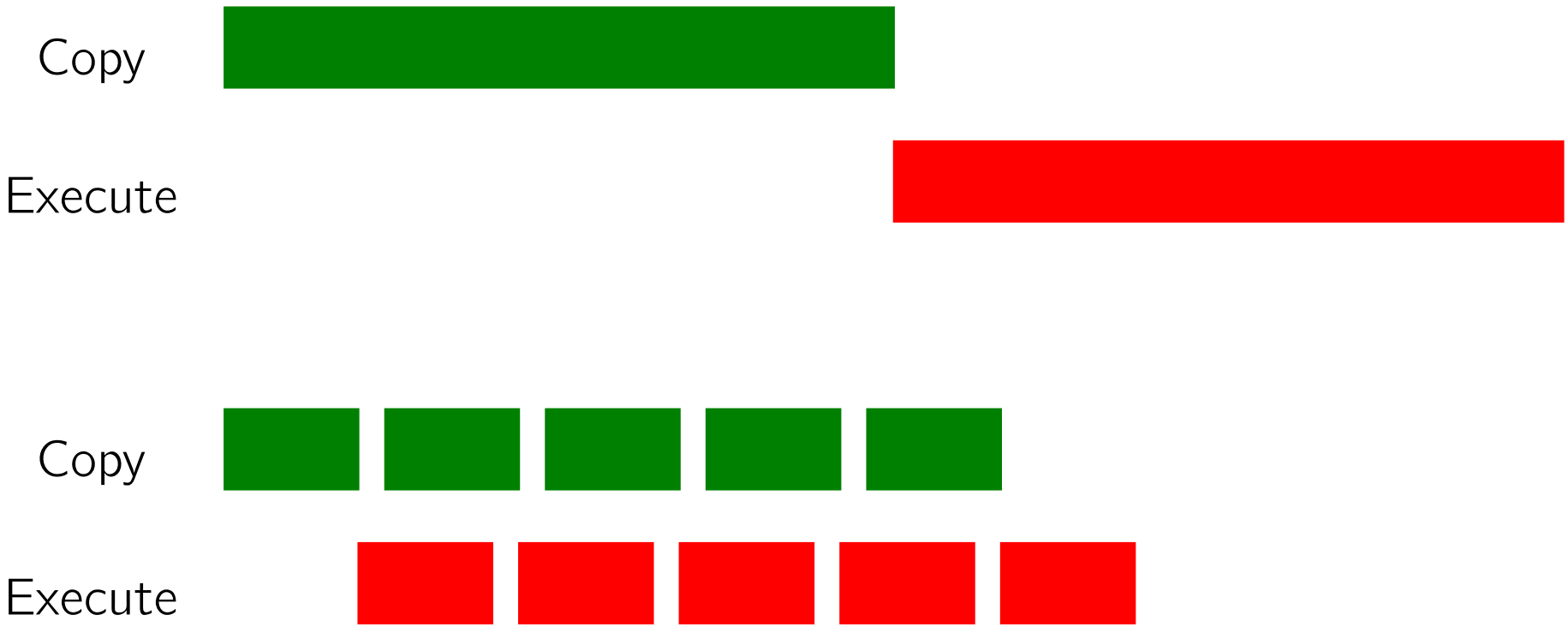
- Paged Memory → total execution time = **28.609 ms** ↯
- Pinned Memory → total execution time = **26.486 ms**
  - ▷ `cudaMallocHost(), cudaFreeHost()`
- Asynchronous data transfer and kernel invocation

```
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);
cudaMemcpyAsync(a_d, a_h, size, cudaMemcpyHostToDevice,
stream1);
...
kernel<<<dimGrid,dimBlock,sharedMem,stream1>>>(a_d);
...
```

- Idea : run multiple streams asynchronously that operate on different parts of the data

Comparing to the traditional way

```
cudaMemcpy(a_d, a_h, size, cudaMemcpyHostToDevice);
kernel<<<dimGrid,dimBlock,sharedMem>>>(a_d);
```



Conclusions

- Reference CPU-only implementation: **5.782 ms**
- Best GPU variant : **26.486 ms** ↯
  - Main cost due to data transfer
  - But also computation is slower → **5.993 ms** ↯
- Heavier arithmetic computation required so that cost of data transfer can be amortized
- We improved GPU variant from **60.126 ms** → **26.486 ms**
- We improved kernel performance from **36.046 ms** → **5.993 m**
- Parallel reduction → "Use Case: easy to implement harder to get it right" (Mark Harris, NVIDIA)