Programming GPUs

Lecture 6

Branch Execution and Reduction Algorithms

Juan J. Durillo
Divergent Control Flow Reduction Algorithms

Section 1 Divergent Control Flow
- Instructions are issued in lockstep in wavefronts/warps for AMD/nVidia
- What happen with “if” branches?
  - Each work item can execute a different path from others in the wavefront
How does a GPU handle threads going on different execution paths when the same instruction has to be executed by all the work items in a wavefront?

**Predication** is a method for mitigating the costs associated with conditional branches

- Beneficial in case of branches with short sections of code
- Compilers may replace “switch” or “if then else” statements by using branch predication
A predicate is a condition code that is set to true or false based on a conditional.

Both cases of the conditional flow get scheduled for execution:
- Instructions with a true predicate are committed.
- Instructions with a false predicate do not write results or read operands.
- This divergence is masked by hardware.

Benefits of performance only for very short conditionals:
- Predication is made at the level of warp/wavefront.
- When the whole wavefront/warp is evaluated to the same value of the condition, only one of the branches is required to be executed.
Predication for GPUs

Time for `Do_Some_Work` = $t_1$ (if case)
Time for `Do_Other_Work` = $t_2$ (else case)

if (tid % 2 == 0)

- `Do_Some_Work()`

Squash invalid results, invert mask

Green colored threads have valid results

Total Time taken = $t_{\text{start}} + t_1 + t_2$
Section 2 | Reduction Algorithms
So far we saw only embarrassingly parallel problems

- No dependence among data
- Thread with id $i$ accesses only to memory $A[i]$
- We use synchronization (fences) to synchronize because of memory accesses optimization
  - Not because the problem needed it

Unfortunately, this is not the case in most of real world problems

- Data show dependence among
To reduce an array to a single element

Example: adding all the elements in an array

```c
float reduce_sum(float* input, int length)
{
    float accumulator = input[0];
    for(int i = 1; i < length; i++)
        accumulator += input[i];
    return accumulator;
}
```
Reduction

- It is a fundamental data-parallel primitive
  - Used in many applications, from database to physical simulation and machine learning
- There are many different kinds of reduction depending on
  - The type of data being reduced
  - The operator which is being used to perform the reduction (+, *, max, min, ...)
- It is very easy to implement!
  - Difficult to do it right
  - It is a nice example of optimization!
Reduction

- If our reduction operator gives some flexibility in terms of what order the operations must be performed, we can parallelize a sequential reduction

- Associacivity
  - If an operator allows us to regroup the operations and still get the same result
  - E.g., \((10+20)+30 = (10 + (20+30))\)

- Commutativity
  - If an operator allows us to reorder the operations and still get the same result
  - E.g., \(((10+20) + 30) = ((30 + 10) + 20)\)
Reduction

- Taking advantage of associativity
- The idea for applying reduction is to use a tree based approach within each work group
- The code should be able to use multiple work groups if necessary
  - To process very large arrays
  - To keep all the GPU cores busy
  - Each work group reduces a portion of the whole array/vector
How do we proceed with large arrays that do not fit in a work group?

- We do not have synchronization among different work groups
- Avoid synchronization by decomposing computation into multiple kernel invocations
What is our optimization goal?

- We should strive to reach GPU peak performance
- Reductions usually have very low arithmetic intensity
- Our goal should be to get the peak bandwidth
```c
kernel__ void reduce0(global__ int *g_idata, global__ int *g_odata) {
    local__ int sdata[];

    // each thread loads one element from global to shared mem
    unsigned int tid = get_local_id(0);
    unsigned int i = get_group_id(0)*get_work_dim(0) + get_local_id(0);
    sdata[tid] = g_idata[i];
    barrier();

    // do reduction in shared mem
    for(unsigned int s=1; s < get_work_dim(0); s *= 2) {
        if (tid % (2*s) == 0) {
            sdata[tid] += sdata[tid + s];
        }
        barrier();
    }

    // write result for this block to global mem
    if (tid == 0) g_odata[get_group_id(0)] = sdata[0];
}
```
Reduction

Divergent Control Flow
Reduction Algorithms

Step 1
Stride 1
Thread IDs
10 1 8 -1 0 -2 3 5 -2 -3 2 7 0 11 0 2
Values
11 1 7 -1 -2 -2 8 5 -5 -3 9 7 11 11 2 2

Step 2
Stride 2
Thread IDs
0
Values
18 1 7 -1 6 -2 8 5 4 -3 9 7 13 11 2 2

Step 3
Stride 4
Thread IDs
0
Values
24 1 7 -1 6 -2 8 5 17 -3 9 7 13 11 2 2

Step 4
Stride 8
Thread IDs
0
Values
41 1 7 -1 6 -2 8 5 17 -3 9 7 13 11 2 2
Reduction

Problems:
- Highly divergent branch
- Operator % is very inefficient
Old Code

```c
for (unsigned int s=1; s < get_work_dim(0); s *= 2) {
    if (tid % (2*s) == 0) {
        sdata[tid] += sdata[tid + s];
    }
    barrier();
}
```

New One

```c
for (unsigned int s=1; s < get_work_dim(0); s *= 2) {
    int index = 2 * s * tid;
    if (index < get_work_dim(0)) {
        sdata[index] += sdata[index + s];
    }
    barrier();
}
```
Reduction

Step 1
Stride 1
Thread IDs
Values

Step 2
Stride 2
Thread IDs
Values

Step 3
Stride 4
Thread IDs
Values

Step 4
Stride 8
Thread IDs
Values
Problems:
- Bank memory conflicts

Solution:
- Take advantage of commutativity
Old Code

```c
for (unsigned int s=1; s < get_work_dim(0); s *= 2) {
    int index = 2 * s * tid;

    if (index < get_work_dim(0)) {
        sdata[index] += sdata[index + s];
    }
    barrier();
}
```

New One

```c
for (unsigned int s=get_work_dim(0); s>0; s>>=1) {
    if (tid < s) {
        sdata[tid] += sdata[tid + s];
    }
    barrier();
}
```
Reduction Algorithms

Step 1
Stride 8

Thread IDs

Values
8  1  8 -1  0 -2  3  5 -2 -3  2  7  0  11  0  2

Step 2
Stride 4

Thread IDs

Values
8  7  10  6  9  3  7 -2 -3  2  7  0  11  0  2

Step 3
Stride 2

Thread IDs

Values
21  20  13  13  0  9  3  7 -2 -3  2  7  0  11  0  2

Step 4
Stride 1

Thread IDs

Values
41  20  13  13  0  9  3  7 -2 -3  2  7  0  11  0  2
Problems:

- Half of the threads in the group are not doing anything
- Waste of resources
Halve the number of blocks, and replace single load:

```c
// each thread loads one element from global to local mem
unsigned int tid = get_local_id(0);
unsigned int i = get_group_id(0)*get_work_dim(0) +
get_local_id(0);
sdata[tid] = g_idata[i];
barriers();
```

With two loads and first add of the reduction:

```c
// perform first level of reduction,
// reading from global memory, writing to local memory
unsigned int tid = get_local_id(0);
unsigned int i = get_group_id(0)*get_work_dim(0)*2,
get_local_id(0);
sdata[tid] = g_idata[i] + g_idata[i+get_local_id(0)];
barrier();
```