```
/*
grid size = 16385 block size = 1024
Processing 16777216 elements...
Host CPU Processing time: 46.452999 (ms)
CUDA Processing time: 15.079000 (ms)
Speedup: 3.080642X
Test PASSED
cal7 + aggre2 + downSweep2
*/
```

In our final version, we will test if it can be divided by 1024 which is our block size. If it is not a power of 2 size, every time before any thread make any action, the thread has to check if its global id is smaller than number of elements. Only when global id is smaller than number of elements, the threads will make action. Also, the global id of the last element in the last block will be the (number of elements - 1). From the fact that we can handle 160000, our algo. won't fail when the number of elements is not power of 2.

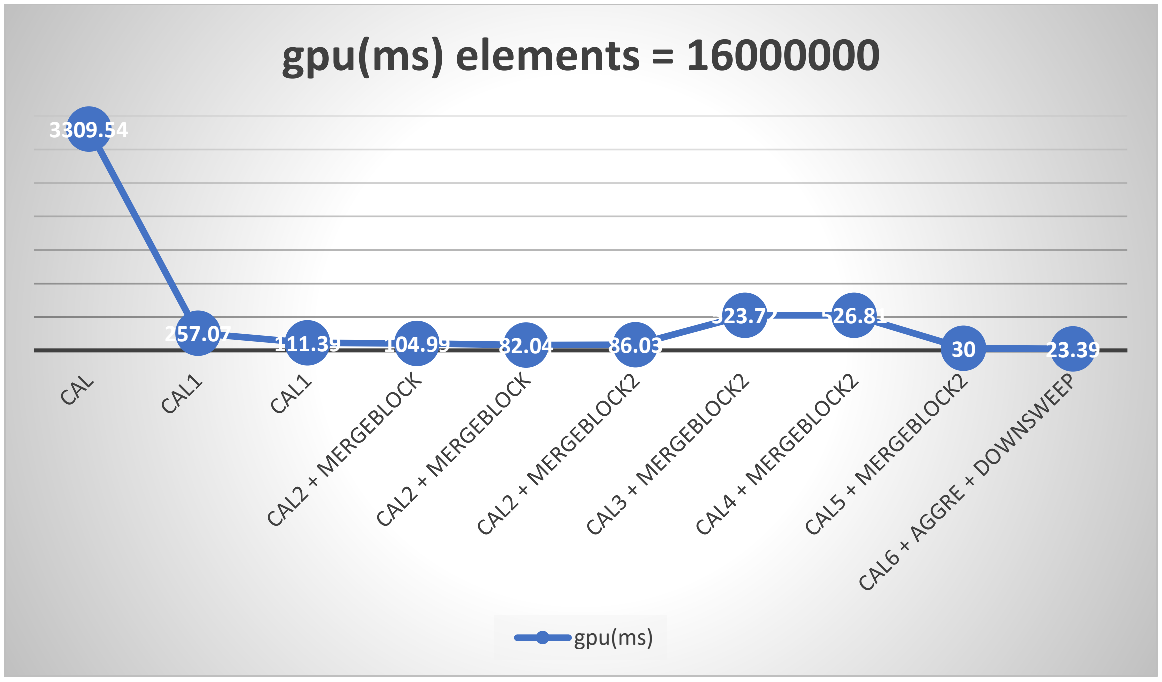
We have tried many ways to solve the problem and the results shows that Blelloch's algo is the best. We can know the difference by comparing the versions that before using cal5 and using cal5 and after cal5. In order to speed up, we also use template to handle when number of elements can't be divided to 1024, because in this situation our last block need to consider more situation. We also notice that there are a lot of variable or conditions are used several times. We try to make these variables be const variables and reuse them. Besides, we also use bit operation to handle division and multiplication. By using these techniques, we improve our speed up from 1.89 to 2.58. After these, we add padding in our sharing memory to avoid as much as possible bank conflict, this improves our speed up from 2.58 to 3.08. Unfortunately, we still can't improve our performance to be five times faster than cpu. Our guess might be there are still some bank conflict that we can avoid.

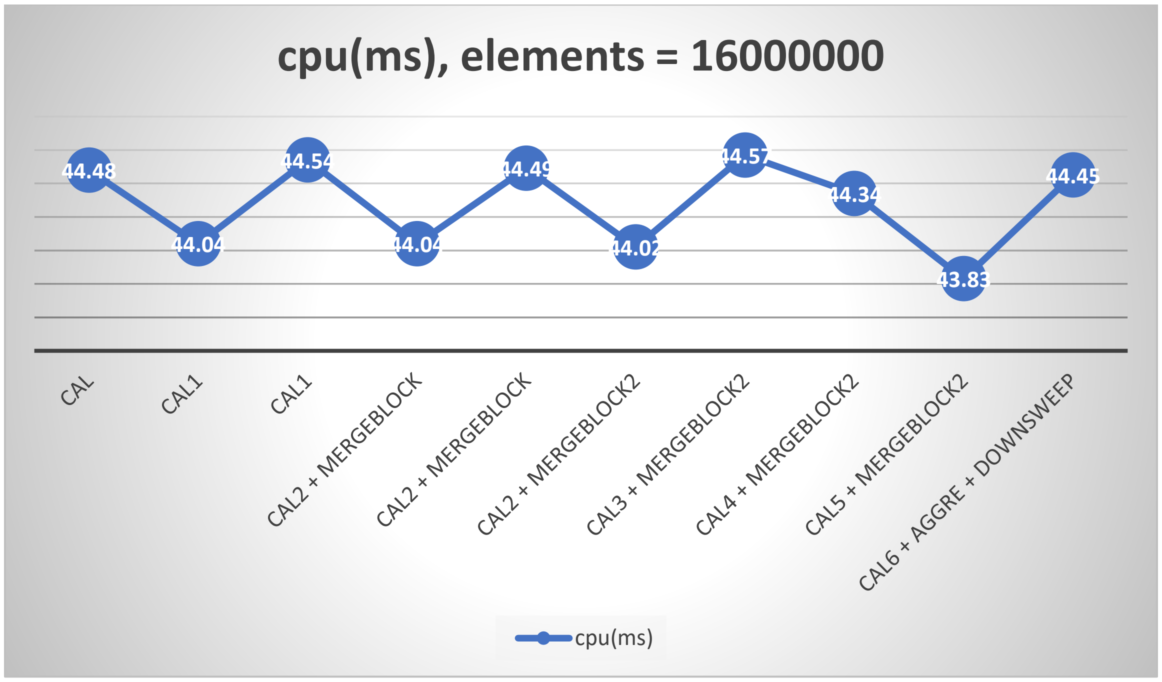
For the FLOPS rate of CPU, because CPU implement a O(n) algorithm to solve this problem, FLOPS = 16777216 / (46.453 / 1000). For the FLOPS rate of GPU, because GPU implement a O(2*n(for upswep and down swep) + (n/1024) (for the aggre)) algorithm to solve this problem, FLOPS = (2 + 1/1024) * 16777216 / (15.079 / 1000). For Intel(R) Xeon(R) CPU E5-1620 0 @ 3.60GHz, the limit of theoretic performance is 3.96 GFLOPS/core, from https://lhcathome.cern.ch/lhcathome/cpu_list.php. For NVIDIA GeForce GTX 680 ,the limit of theoretic FP32 (float) performance is 3,250 GFLOPS, from https://www.techpowerup.com/gpu-specs/geforce-gtx-680.c342. For out GPU implementation, the bottlenecks are the time read from global memory to shared memory, reading shared memory probably with some bank conflict, and from shared memory to global memory.

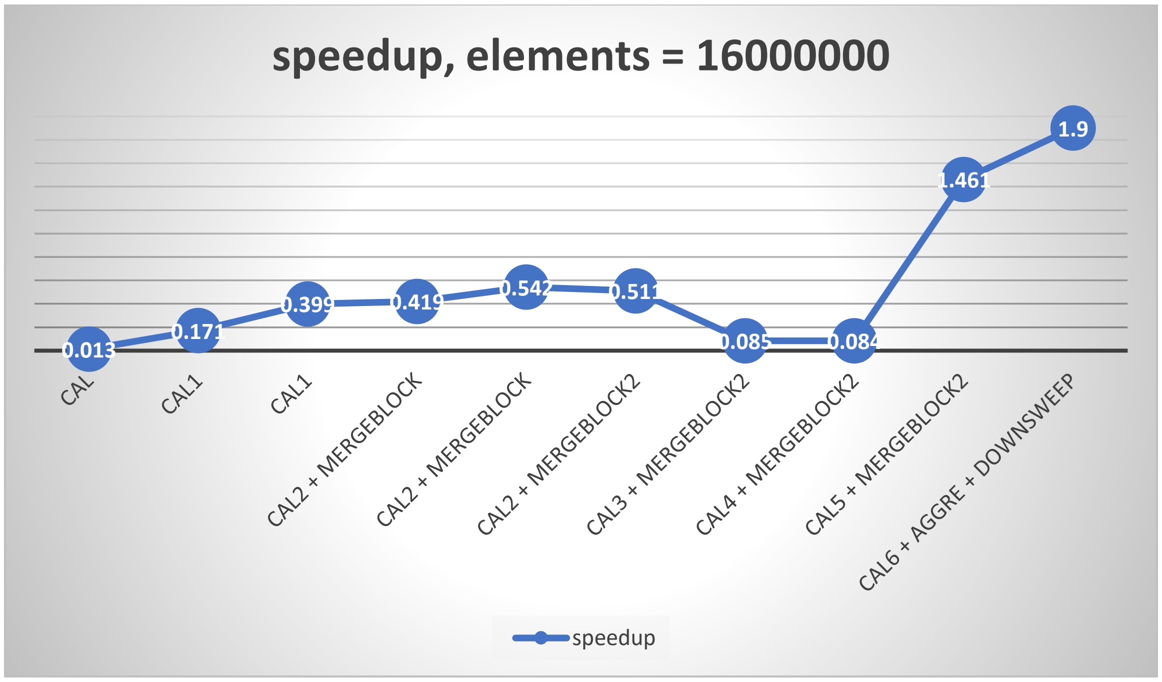
version	Elements	Grid size and block size	Gpu time	Cpu time	Speedup
Cal	16000000	1, 1	3309.541992 (ms)	44.480000 (ms)	0.013440X
Cal1	16000000	1, 256	257.076996 (ms)	44.043999 (ms)	0.171326X

Cal1	16000000	1, 1024	111.398003 (ms)	44.544998 (ms)	0.399873X
cal2 + mergeblock	16000000	256, 1	104.995003 (ms)	44.049999 (ms)	0.419544X
cal2 + mergeblock	16000000	1024,1	82.049004 (ms)	44.497002 (ms)	0.542322X
cal2 + mergeblock2	16000000	1024, 1	86.033997 (ms)	44.027000 (ms)	0.511740X
cal3 + mergeblock2	16000000	1024, 1024	523.723022 (ms)	44.577000 (ms)	0.085116X
Cal4 + mergeblock2	16000000	1024, 1024	526.818970 (ms)	44.347000 (ms)	0.084179X
Cal5 + mergeblock2	16000000	1024, 1024	30.003000 (ms)	43.834999 (ms)	1.461020X
Cal6 + aggre + downSweep	16000000	numElement/1024+1, 1024	23.393000 (ms)	44.452999 (ms)	1.900269X
Cal6 + aggre + downSweep	16777216	num/1024 or num/1024 +1, 1024	24.499001 (ms)	46.369999 (ms)	1.892730X
Cal6 + aggre + downSweep (use template and reuse variables that use several times)	16777216	num/1024 or num/1024 +1, 1024	17.938000 (ms)	46.398998 (ms)	2.586632X
Cal6 + aggre + downSweep (use template and reuse variables that use several times and padding)	16777216	num/1024 = 16385 or num/1024 +1, 1024	15.079000 (ms)	46.452999 (ms)	3.080642X

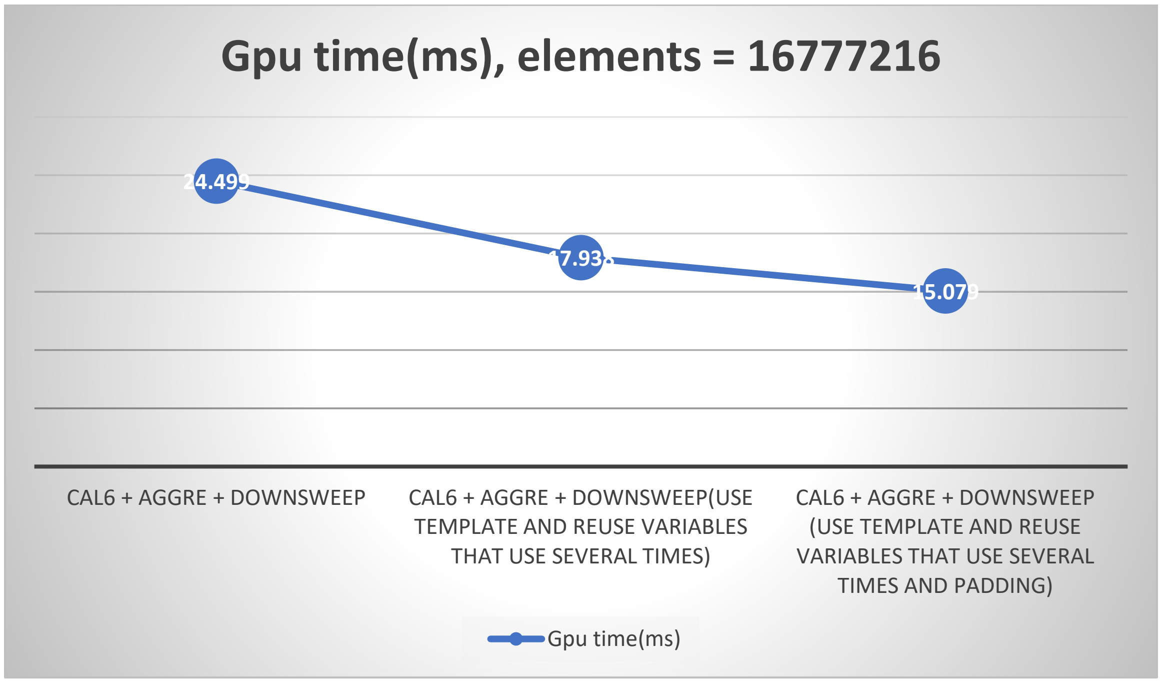
Performance when elements = 16000000

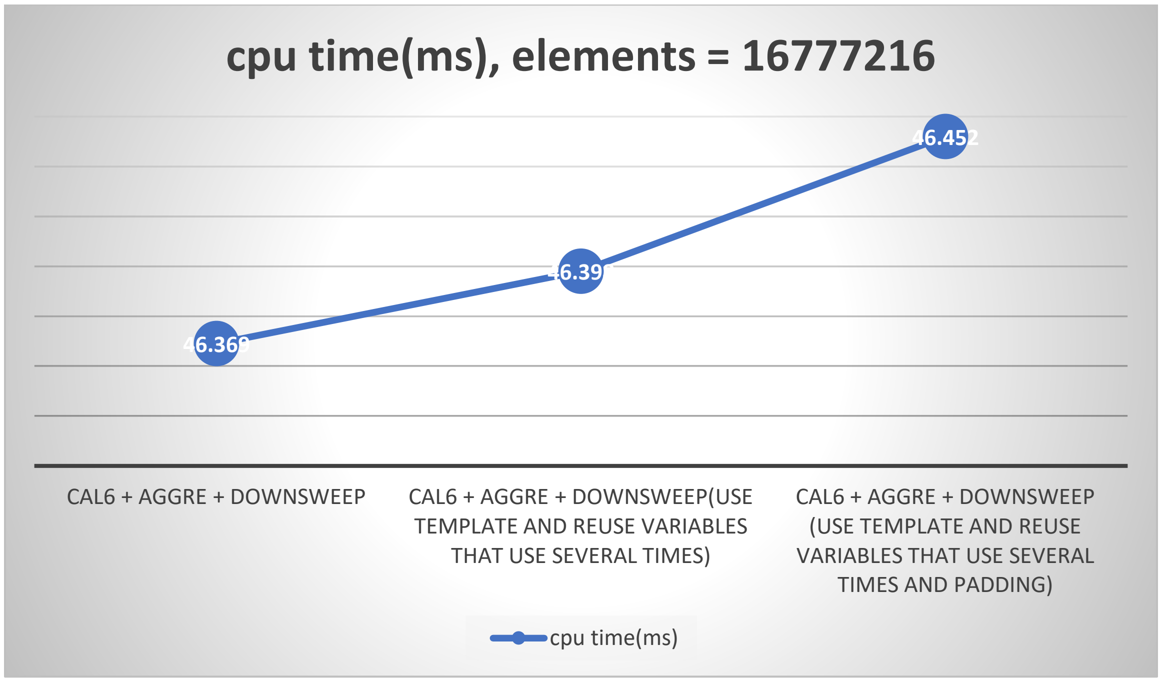

gpu(ms) elements = 16000000
3309.54
257.07
111.39
104.99
82.04
86.03
123.72
126.81
30
23.39
CAL
CAL1
CAL1
CAL2 + MERGEBLOCK
CAL2 + MERGEBLOCK
CAL2 + MERGEBLOCK2
CAL3 + MERGEBLOCK2
CAL4 + MERGEBLOCK2
CAL5 + MERGEBLOCK2
CAL6 + AGGRE + DOWNSWEEP
gpu(ms)

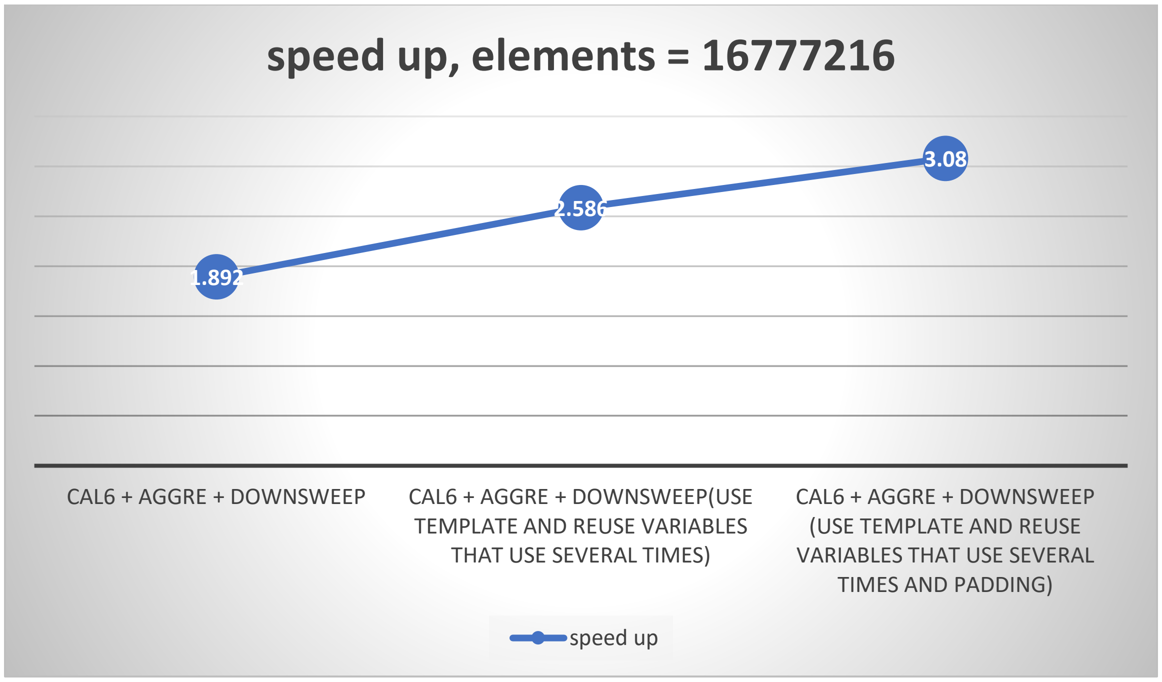

cpu(ms), elements = 16000000
44.48
44.04
44.54
44.04
44.49
44.02
44.57
44.34
43.83
44.45
CAL
CAL1
CAL1
CAL2 + MERGEBLOCK
CAL2 + MERGEBLOCK
CAL2 + MERGEBLOCK2
CAL3 + MERGEBLOCK2
CAL4 + MERGEBLOCK2
CAL5 + MERGEBLOCK2
CAL6 + AGGRE + DOWNSWEEP
cpu(ms)


speedup, elements = 16000000
0.013
0.171
0.399
0.419
0.542
0.511
0.085
0.084
1.461
1.9
CAL
CAL1
CAL1
CAL2 + MERGEBLOCK
CAL2 + MERGEBLOCK
CAL2 + MERGEBLOCK2
CAL3 + MERGEBLOCK2
CAL4 + MERGEBLOCK2
CAL5 + MERGEBLOCK2
CAL6 + AGGRE + DOWNSWEEP
speedup

Performance when elements = 16777216






speed up, elements = 16777216
1.892
2.586
3.08
CAL6 + AGGRE + DOWNSWEEP
CAL6 + AGGRE + DOWNSWEEP(USE TEMPLATE AND REUSE VARIABLES THAT USE SEVERAL TIMES)
CAL6 + AGGRE + DOWNSWEEP (USE TEMPLATE AND REUSE VARIABLES THAT USE SEVERAL TIMES AND PADDING)
speed up

Performance output detail

```
//single thread, cal<<<(1, 1, 1), BLOCK_SIZE>>>(outArray, inArray, numElements);
/*
Processing 16000000 elements...
Host CPU Processing time: 44.480000 (ms)
CUDA Processing time: 3309.541992 (ms)
Speedup: 0.013440X
Test PASSED
*/
Cal
//implement outArray[i] = outArray[i-1] + inArray[i-1];
Using global memory
```

```
//BLOCK_SIZE threads
/*block size = 256
Processing 16000000 elements...
Host CPU Processing time: 44.043999 (ms)
CUDA Processing time: 257.076996 (ms)
Speedup: 0.171326X
Test PASSED
*/
/*block size = 1024
Processing 16000000 elements...
Host CPU Processing time: 44.544998 (ms)
CUDA Processing time: 111.398003 (ms)
Speedup: 0.399873X
Test PASSED
*/
Cal1
// implement parallel initialize output, split jobs to threads equally, and each
threads aggregate sum in their field, and all threads helps the data was handled
by thread1 to add the sum was handled by thread0, and then do the data was
handled by thread2 and so on.
Using global memory
```

```
//BLOCK_SIZE threads
//Grid size = 1024 but block Size = 1 and need to call mergeBlock
//to be used after cal2
/*
Grid size = 1024
Processing 16000000 elements...
Host CPU Processing time: 44.497002 (ms)
```

```
CUDA Processing time: 82.049004 (ms)
Speedup: 0.542322X
Test PASSED
*/
/*
Grid size = 256
Processing 16000000 elements...
Host CPU Processing time: 44.049999 (ms)
CUDA Processing time: 104.995003 (ms)
Speedup: 0.419544X
Test PASSED
*/
cal2 + mergeblock
//similar to cal1 but split the jobs to grids, and we need a merge block to
handle the aggregation because we have to do devicesynchronized()
megreblock uses 1024 threads
Using global memory
```

```
//BLOCK_SIZE threads
//Grid size = 1024 but block Size = 1 and need to call mergeBlock
/*
grid size = 1024
Processing 16000000 elements...
Host CPU Processing time: 44.027000 (ms)
CUDA Processing time: 86.033997 (ms)
Speedup: 0.511740X
Test PASSED
*/
cal2 + mergeblock2
mergeblock2 uses share memory without padding
megreblock2 uses 1024 threads
```

```
//to be used after cal3
/*
grid size = 1024
block size = 1024
Processing 16000000 elements...
Host CPU Processing time: 44.577000 (ms)
CUDA Processing time: 523.723022 (ms)
Speedup: 0.085116X
Test PASSED
*/
cal3 + mergeblock2
```

```
//split jobs to grid size times block size, each thread calculates its own jobs,
and in the end thread 0 will aggregate the whole block and merge it in
mergeblock2
megreblock2 uses 1024 threads
cal3 uses global memory, megreblock2 uses shared memory without padding
```

```
to be used after cal4
grid size = 1024, block size = 1024
Processing 16000000 elements...
Host CPU Processing time: 44.347000 (ms)
CUDA Processing time: 526.818970 (ms)
Speedup: 0.084179X
Test PASSED
Cal4 + mergeblock2
Similar to cal3 + mergeblock2, but all the threads in the block will do
aggregation
Cal4 uses global memory, mergeblock2 uses shared memory
```

```
/*
to be used after cal5 and before mergeblock2
grid size = 1024, block size = 1024
Processing 16000000 elements...
Host CPU Processing time: 43.834999 (ms)
CUDA Processing time: 30.003000 (ms)
Speedup: 1.461020X
Test PASSED
*/
Cal5 + upAndDownSweep + mergeblock2
//implement Blelloch’s algo.
Cal5 and upAndDownSweep uses global memory, mergeblock2 uses shared memory
```

```
/*
to be used after cal6, aggre
grid size = num/1024+1, block size = 1024
Processing 16000000 elements...
Host CPU Processing time: 44.452999 (ms)
CUDA Processing time: 23.393000 (ms)
Speedup: 1.900269X
Test PASSED
*/
Cal6 + aggre + downSweep
```

```
Similar to cal5 but we divide blelloch’s algo to three parts, and before we do
downsweep, we first add the sum of previous block into the first element of the
block
Use shared memory without padding
```

```
/*
Processing 16777216 elements...
to be used after cal6, aggre
grid size = num/1024+1, block size = 1024
Host CPU Processing time: 46.369999 (ms)
CUDA Processing time: 24.499001 (ms)
Speedup: 1.892730X
Test PASSED
*/
Cal6 + aggre + downSweep
```

```
/*
Processing 16777216 elements...
to be used after cal6 aggre
grid size = num/1024 or num/1024 +1, block size = 1024
Host CPU Processing time: 46.398998 (ms)
CUDA Processing time: 17.938000 (ms)
Speedup: 2.586632X
Test PASSED
*/
Cal6 + aggre + downSweep
Using template and use bit operation and reuse variable that will be used several
times
```

```
/*
grid size = 16385 block size = 1024
Processing 16777216 elements...
Host CPU Processing time: 46.452999 (ms)
CUDA Processing time: 15.079000 (ms)
Speedup: 3.080642X
Test PASSED
cal7 + aggre2 + downSweep2
*/
```