EuroCC@Turkey
Parallel Computing on GPUs with CUDA

Dr. Özcan DÜLGER

Computer Engineering, Middle East Technical University

Computer Engineering, Artvin Coruh University

27 April 2022

Contents

- Debugging and Profiling Performance
- **Performance Optimization and Efficiency**
- Some Libraries and Remaining Issues
- CUDA Samples

Sabancı Üniversitesi

ORTA DOĞU TEKNİK ÜNİVERSİTESİ
MIDDLE EAST TECHNICAL UNIVERSITY

TESLA K40

Property	Value	Property	Value
Architecture	Kepler	Global Memory	11520 MB
Number of SMX	15	Shared Memory	49152 Byte
CUDA Core	2880	L2 Cache	1572864 Byte
Core Clock	745 MHz	Segment Size	128 Byte
Max. Thread / SMX	2048	Warp Size	32
Max. Thread / Block	1024	Max. Block / SMX	16

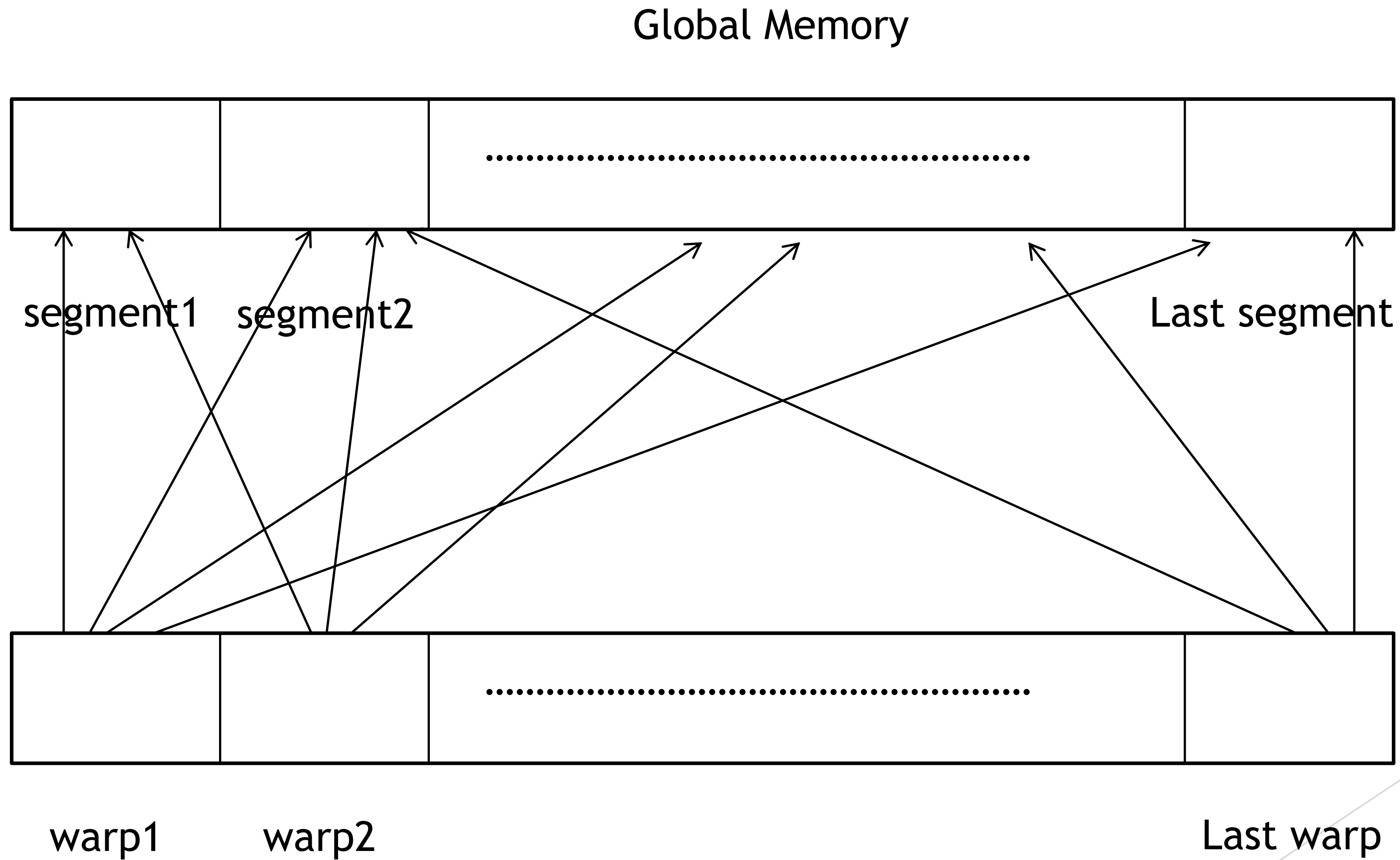
Performance Optimization and Efficiency

- Memory Coalesced Access to Global Memory
- Warp Divergence
- Device Occupancy and SM Efficiency
- Efficient Parallel Reduction Algorithm

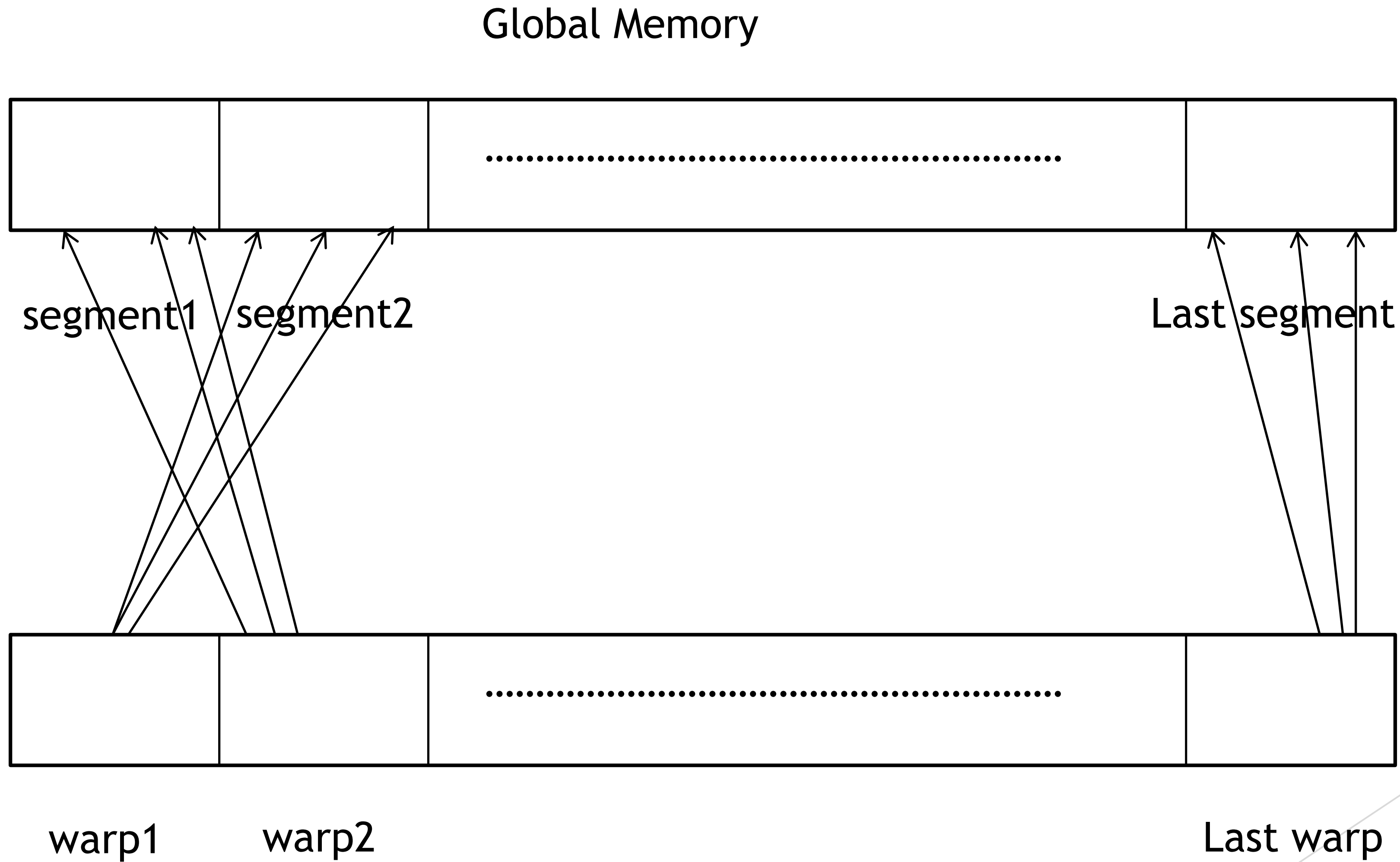
Memory Coalesced Access

- Reading from or writing to global memory performs segment by segment
- The threads in a warp are physically related to each other. That means a warp completes its instruction when all the threads in the warp complete the instruction
- In global memory operations, if the threads in the warp access to the different segments of the global memory, the operations become serial

None Coalesced Access



Memory Coalesced Access



Example of Memory Coalesced Access

```
1 __global__ void full_coalesced_access(float *A,float *B,float *C)
2 {
3     unsigned int tid = blockDim.x*blockIdx.x+threadIdx.x;//Global thread id
4     C[tid] = A[tid] + B[tid];//Vector addition
5 }
6
7 __global__ void non_coalesced_access(float *A,float *B,float *C,curandInitializer RNGs,unsigned int NP)
8 {
9     unsigned int tid = blockDim.x*blockIdx.x+threadIdx.x;//Global thread id
10
11    curandState_t state;// State of the generator
12    RNGs.load(state,tid);// Loading the state
13    unsigned int index,i;
14
15    for(i=0;i<100;i++)
16    {
17        index = curand(&state)/NP;// Generate a random number between 0 and data_size-1
18        C[tid] = A[index] + B[index];//Vector addition
19    }
20 }
21
22 int main(int argc, char **argv)
23 {
24    unsigned int data_size = 4194304;//Data size
25    float *A_host,*B_host,*C_host ;//Host Arrays
26    float *A_GPU,*B_GPU,*C_GPU;//Device Arrays
27
28    for(int counter = 0;counter < data_size; counter++)
29    {
30        A_host[counter] = counter+1;//Assigning numbers from 1 to size
31        B_host[counter] = counter+2;//Assigning numbers from 2 to size+1
32    }
33
34    cudaMemcpy(A_GPU,A_host,sizeof(float)*data_size,cudaMemcpyHostToDevice);
35    cudaMemcpy(B_GPU,B_host,sizeof(float)*data_size,cudaMemcpyHostToDevice);
36
37    unsigned int NTB = 1024;//Number of threads in a block
38    unsigned int NP_data_size = (unsigned long int)pow(2,32)/data_size;// Number of partitions for 'data_size'
39
40    dim3 threadsPerBlock(NTB);//Number of threads in a block
41    dim3 numBlocks(data_size/NTB);//Number of blocks in a grid
42
43    curandInitializer RNGs(data_size);//Creating generators for each thread in 'non_coalesced_access' kernel
44    unsigned int clck = clock();
45    initialize_RNGs<<<numBlocks,threadsPerBlock>>>(RNGs,clck);//Initializing the states of each generator
46
47    full_coalesced_access<<<numBlocks,threadsPerBlock>>>(A_GPU,B_GPU,C_GPU);//Launching 'full_coalesced_access' kernel
48    non_coalesced_access<<<numBlocks,threadsPerBlock>>>(A_GPU,B_GPU,C_GPU,RNGs,NP_data_size);//Launching 'non_coalesced_access' kernel
49    cudaDeviceSynchronize();//Waits until the kernel completes its run
50 }
```

Load in one transaction

Load the state of the generator of each thread from global memory as a coalesced way

Load in at most 32 transactions

Note: We use cudaEventRecord in order to measure the kernel execution times

XORWOW Generators

```
1 __global__ void full_coalesced_access(float *A,float *B,float *C)
2 {
3     unsigned int tid = blockDim.x*blockIdx.x+threadIdx.x;//Global thread id
4     C[tid] = A[tid] + B[tid];//Vector addition
5 }
6
7 __global__ void non_coalesced_access(float *A,float *B,float *C,curandInitializer RNGs,unsigned int NP)
8 {
9     unsigned int tid = blockDim.x*blockIdx.x+threadIdx.x;//Global thread id
10
11    curandState_t state;// State of the generator
12    RNGs.load(state,tid);// Loading the state
13    unsigned int index,i;
14
15    for(i=0;i<100;i++)
16    {
17        index = curand(&state)/NP;// Generate a random number between 0 and data_size-1
18        C[tid] = A[index] + B[index];//Vector addition
19    }
20 }
21
22 int main(int argc, char **argv)
23 {
24    unsigned int data_size = 32768;//Data size
25    float *A_host,*B_host,*C_host ;//Host Arrays
26    float *A_GPU,*B_GPU,*C_GPU;//Device Arrays
```

Metrics:
gld_transactions: Number of global memory load transactions
gld_transactions_per_request: Average number of global memory load transactions performed for each global memory load

```
Exec. Time of 'full_coalesced_access' kernel = 9.504e-06
Exec. Time of 'non_coalesced_access' kernel = 0.00168371
Speed Up = 177.158X
```

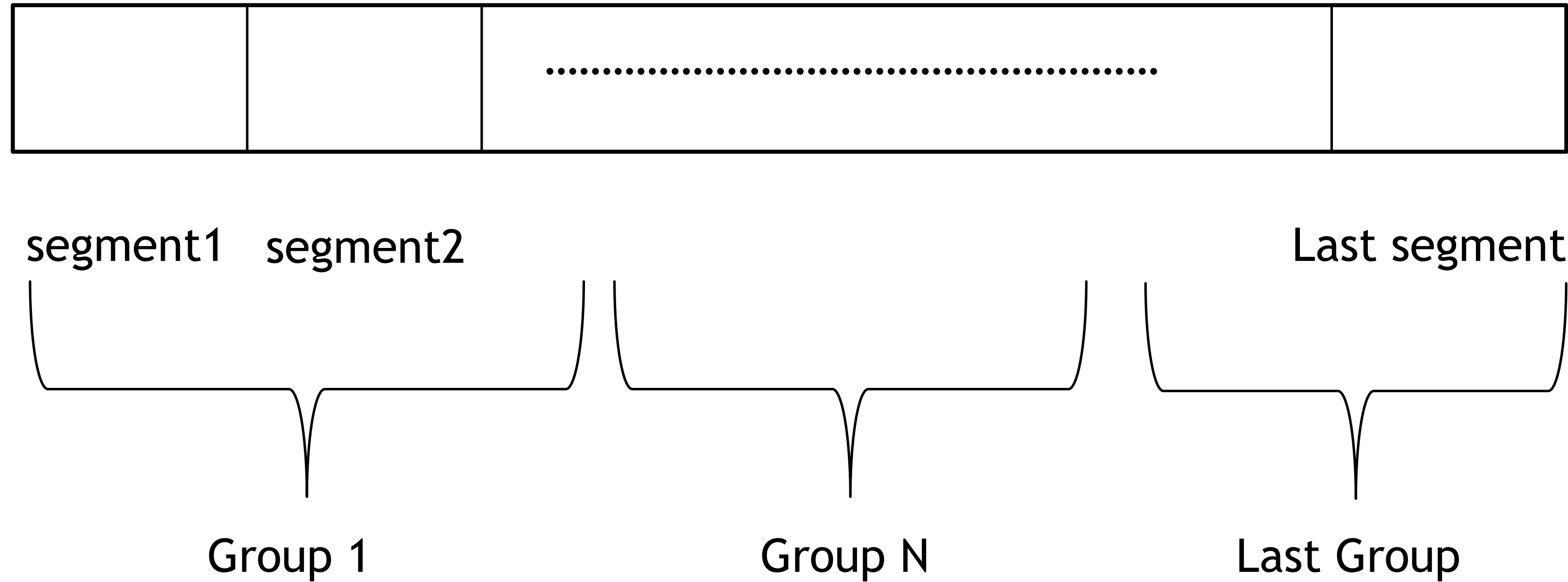
```
==11093== Profiling result:
==11093== Metric result:
Invocations                    Metric Name                    Metric Description         Min        Max        Avg
Device "Tesla K40c (0)"
    Kernel: full_coalesced_access(float*, float*, float*)
          1                    gld_transactions              Global Load Transactions       2048       2048       2048
          1        gld_transactions_per_request  Global Load Transactions Per Request   1.000000   1.000000   1.000000
    Kernel: non_coalesced_access(float*, float*, float*, curandInitializer, unsigned int)
          1                    gld_transactions              Global Load Transactions    6461574    6461574    6461574
          1        gld_transactions_per_request  Global Load Transactions Per Request  30.631703  30.631703  30.631703
```

```
1 __global__ void full_coalesced_access(float *A,float *B,float *C)
2 {
3     unsigned int tid = blockDim.x*blockIdx.x+threadIdx.x;//Global thread id
4     C[tid] = A[tid] + B[tid];//Vector addition
5 }
6
7 __global__ void non_coalesced_access(float *A,float *B,float *C,curandInitializer RNGs,unsigned int NP)
8 {
9     unsigned int tid = blockDim.x*blockIdx.x+threadIdx.x;//Global thread id
10
11    curandState_t state;// State of the generator
12    RNGs.load(state,tid);// Loading the state
13    unsigned int index,i;
14
15    for(i=0;i<100;i++)
16    {
17        index = curand(&state)/NP;// Generate a random number between 0 and data_size-1
18        C[tid] = A[index] + B[index];//Vector addition
19    }
20 }
21
22 int main(int argc, char **argv)
23 {
24    unsigned int data_size = 4194304;//Data size
25    float *A_host,*B_host,*C_host ;//Host Arrays
```

```
Exec. Time of 'full_coalesced_access' kernel = 0.000290432
Exec. Time of 'non_coalesced_access' kernel = 0.516614
Speed Up = 1778.78X
```

```
==11325== Profiling result:
==11325== Metric result:
Invocations                               Metric Name                        Metric Description         Min         Max         Avg
Device "Tesla K40c (0)"
        Kernel: full_coalesced_access(float*, float*, float*)
          1                          gld_transactions                  Global Load Transactions      262144      262144      262144
          1              gld_transactions_per_request      Global Load Transactions Per Request    1.000000    1.000000    1.000000
        Kernel: non_coalesced_access(float*, float*, float*, curandInitializer, unsigned int)
          1                          gld_transactions                  Global Load Transactions   839548174   839548174   839548174
          1              gld_transactions_per_request      Global Load Transactions Per Request   31.093419   31.093419   31.093419
```

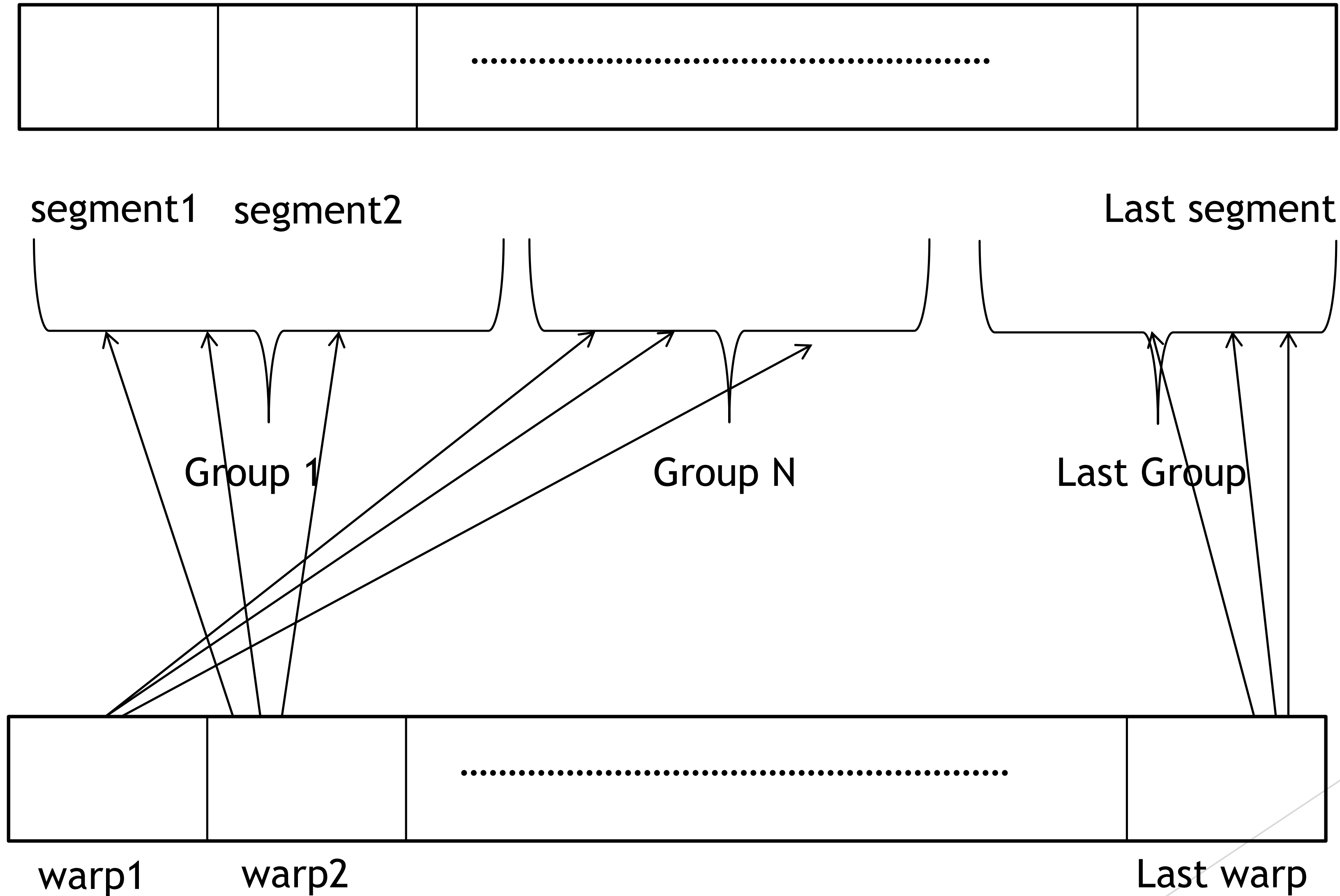
Grouping



- A group consists of contiguous segments
- The number of segments in the group can be
  - between 1 and 32 (warp size)

Dülger, Ö., Oğuztüzün, H. & Demirekler, M. Memory Coalescing Implementation of Metropolis Resampling on Graphics Processing Unit. J Sign Process Syst 90, 433–447 (2018)

Grouping



```
1 __global__ void semi_coalesced_access(float *A,float *B,float *C,curandInitializer RNGs1,curandInitializer RNGs2,unsigned int NPP_group_count,unsigned int NPP_group_size,unsigned int
  GS)
2 {
3     int tid = blockDim.x*blockIdx.x+threadIdx.x;//Global thread id
4
5     curandState_t state1,state2;// States of the generators
6     RNGs1.load(state1,tid);// Loading the state of first generator
7     RNGs2.load(state2,tid);// Loading the state of second generator
8
9     unsigned int GN = curand(&state2)/NPP_group_count;//Generate a random number between 0 and group_count-1 (Pick a random group)
10    unsigned int index,i;
11
12    for(i=0;i<100;i++)
13    {
14        index = (curand(&state1)/NPP_group_size) + (GN*GS);//Generate a random number between 0 and group_size-1 then shift the index (Pick a random data within the selected group)
15        C[tid] = A[index] + B[index];//Vector addition
16    }
17 }
18
19 int main(int argc, char **argv)
20 {
21    unsigned int data_size = 4194304;//Data size
22    float *A_host,*B_host,*C_host ;//Host Arrays
23    float *A_GPU,*B_GPU,*C_GPU;//Device Arrays
24
25    for(int counter = 0;counter < data_size; counter++)
26    {
27        A_host[counter] = counter+1;//Assigning numbers from 1 to size
28        B_host[counter] = counter+2;//Assigning numbers from 2 to size+1
29    }
30
31    cudaMemcpy(A_GPU,A_host,sizeof(float)*data_size,cudaMemcpyHostToDevice);
32    cudaMemcpy(B_GPU,B_host,sizeof(float)*data_size,cudaMemcpyHostToDevice);
33
34    unsigned int NTB = 1024;//Number of threads in a block
35    unsigned int NP_data_size = (unsigned long int)pow(2,32)/data_size;// Number of partitions for 'data_size'
36
37    unsigned int segment_size = 128;//Number of bytes of a segment
38    unsigned int group_size = 16*(segment_size/4);//Number of data in a group
39    unsigned int group_count = data_size/group_size;//Number of groups for 'data_size'
40    unsigned int NP_group_count = (unsigned long int)pow(2,32)/group_count;// Number of partitions for 'group_count'
41    unsigned int NP_group_size = (unsigned long int)pow(2,32)/group_size;// Number of partitions for 'group_size
42
43    dim3 threadsPerBlock(NTB);//Number of threads in a block
44    dim3 numBlocks(data_size/NTB);//Number of blocks in a grid
45
46    full_coalesced_access<<<numBlocks,threadsPerBlock>>>(A_GPU,B_GPU,C_GPU);//Launching 'full_coalesced_access' kernel
47    non_coalesced_access<<<numBlocks,threadsPerBlock>>>(A_GPU,B_GPU,C_GPU,RNGs,NP_data_size);//Launching 'non_coalesced_access' kernel
48    cudaDeviceSynchronize();//Waits until the kernel completes its run
49 }
```

- The number of segments is 16 in a group
- So the memory operations of a warp will perform at most 16 transactions

```
1 __global__ void semi_coalesced_access(float *A,float *B,float *C,curandInitializer RNGs1,curandInitializer RNGs2,unsigned int NPP_group_count,unsigned int NPP_group_size,unsigned int
  GS)
2 {
3     int tid = blockDim.x*blockIdx.x+threadIdx.x;//Global thread id
4
5     curandState_t state1,state2;// States of the generators
6     RNGs1.load(state1,tid);// Loading the state of first generator
7     RNGs2.load(state2,tid);// Loading the state of second generator
8
9     unsigned int GN = curand(&state2)/NPP_group_count;//Generate a random number between 0 and group_count-1 (Pick a random group)
10    unsigned int index,i;
11
12    for(i=0;i<100;i++)
13    {
14        index = (curand(&state1)/NPP_group_size) + (GN*GS);//Generate a random number between 0 and group_size-1 then shift the index (Pick a random data within the selected group)
15        C[tid] = A[index] + B[index];//Vector addition
16    }
17 }
18
19 int main(int argc, char **argv)
20 {
21    unsigned int data_size = 32768;//Data size
22    float *A_host,*B_host,*C_host ;//Host Arrays
```

```
Exec. Time of 'full_coalesced_access' kernel = 9.6e-06
Exec. Time of 'semi_coalesced_access' kernel = 0.00078848
Speed Up = 82.1333X
Exec. Time of 'non_coalesced_access' kernel = 0.00167981
Speed Up = 174.98X
```

```
==16800== Profiling result:
==16800== Metric result:
Invocations                               Metric Name                        Metric Description         Min         Max         Avg
Device "Tesla K40c (0)"
    Kernel: full_coalesced_access(float*, float*, float*)
          1                          gld_transactions                  Global Load Transactions        2048        2048        2048
          1                  gld_transactions_per_request      Global Load Transactions Per Request    1.000000    1.000000    1.000000
    Kernel: semi_coalesced_access(float*, float*, float*, curandInitializer, curandInitializer, unsigned int, unsigned int, unsigned int)
          1                          gld_transactions                  Global Load Transactions     2870742     2870742     2870742
          1                  gld_transactions_per_request      Global Load Transactions Per Request   13.413679   13.413679   13.413679
    Kernel: non_coalesced_access(float*, float*, float*, curandInitializer, unsigned int)
          1                          gld_transactions                  Global Load Transactions     6461532     6461532     6461532
          1                  gld_transactions_per_request      Global Load Transactions Per Request   30.631504   30.631504   30.631504
```

```
1  __global__ void semi_coalesced_access(float *A,float *B,float *C,curandInitializer RNGs1,curandInitializer RNGs2,unsigned int NPP_group_count,unsigned int NPP_group_size,unsigned int
   GS)
2  {
3      int tid = blockDim.x*blockIdx.x+threadIdx.x;//Global thread id
4
5      curandState_t state1,state2;// States of the generators
6      RNGs1.load(state1,tid);// Loading the state of first generator
7      RNGs2.load(state2,tid);// Loading the state of second generator
8
9      unsigned int GN = curand(&state2)/NPP_group_count;//Generate a random number between 0 and group_count-1 (Pick a random group)
10     unsigned int index,i;
11
12     for(i=0;i<100;i++)
13     {
14         index = (curand(&state1)/NPP_group_size) + (GN*GS);//Generate a random number between 0 and group_size-1 then shift the index (Pick a random data within the selected group)
15         C[tid] = A[index] + B[index];//Vector addition
16     }
17 }
18
19 int main(int argc, char **argv)
20 {
21     unsigned int data_size = 4194304;//Data size
22     float *A_host,*B_host,*C_host ;//Host Arrays
23     float *A_GPU,*B_GPU,*C_GPU;//Device Arrays
```

```
Exec. Time of 'full_coalesced_access' kernel = 0.00029232
Exec. Time of 'semi_coalesced_access' kernel = 0.22142
Speed Up = 757.457X
Exec. Time of 'non_coalesced_access' kernel = 0.516598
Speed Up = 1767.24X
```

```
==16562== Profiling result:
==16562== Metric result:
Invocations                               Metric Name                        Metric Description         Min         Max         Avg
Device "Tesla K40c (0)"
        Kernel: full_coalesced_access(float*, float*, float*)
          1                          gld_transactions                  Global Load Transactions      262144      262144      262144
          1              gld_transactions_per_request      Global Load Transactions Per Request    1.000000    1.000000    1.000000
        Kernel: semi_coalesced_access(float*, float*, float*, curandInitializer, curandInitializer, unsigned int, unsigned int, unsigned int)
          1                          gld_transactions                  Global Load Transactions   367433454   367433454   367433454
          1              gld_transactions_per_request      Global Load Transactions Per Request   13.412894   13.412894   13.412894
        Kernel: non_coalesced_access(float*, float*, float*, curandInitializer, unsigned int)
          1                          gld_transactions                  Global Load Transactions   839547696   839547696   839547696
          1              gld_transactions_per_request      Global Load Transactions Per Request   31.093401   31.093401   31.093401
```

Warp Divergence

- Some of the structures such as 'If-Else' structure are considered as a single instruction for a warp
- A warp completes such instructions when all the threads in the warp complete those instructions
- If the threads in a warp execute the different paths of 'If-Else' structure, executing these paths become serial
- if(tid %2 == 0)//tid is global thread id

```
        …………..
else
        …………..
```

- First the threads with even thread id in a warp execute 'if' path, and the remaining threads wait
- Then the threads with odd thread id in a warp execute the 'else' path, and the remaining threads wait

Warp Divergence

- It is important that all the threads in a warp execute the same path of the ‘if-Else’ structure
- This can be ensured by using warp id in the condition of the structure
- if((tid/32) %2 == 0)//tid is the global thread id

```
        …………..
else
        …………..
```

- The threads in a warp whose warp id is even execute the ‘if’ path
- The threads in a warp whose warp id is odd execute the ‘else’ path
- So executing the paths do not become serialized

```
1  __global__ void warp_no_divergence(float *A,float *B,float *C)//No branching for the warps in 'if-elseif' structure
2  {
3      unsigned int tid = blockDim.x*blockIdx.x+threadIdx.x;//Global thread id
4
5      for(unsigned int i=0;i<100;i++)
6      {
7          if( (tid/32) % 4 == 0)
8              C[tid] = A[tid] + B[tid];//Vector addition
9          else if( (tid/32) % 4 == 1)
10             C[tid] = A[tid] - B[tid];//Vector subtraction
11         else if( (tid/32) % 4 == 2)
12             C[tid] = A[tid] * B[tid];//Vector multiplication
13         else if( (tid/32) % 4 == 3)
14             C[tid] = A[tid] / B[tid];//Vector division
15     }
16 }
17
18 __global__ void warp_divergence(float *A,float *B,float *C)//Four different paths for the warps in 'if-elseif' structure
19 {
20     unsigned int tid = blockDim.x*blockIdx.x+threadIdx.x;//Global thread id
21
22     for(unsigned int i=0;i<100;i++)
23     {
24         if( tid % 4 == 0)
25             C[tid] = A[tid] + B[tid];//Vector addition
26         else if( tid % 4 == 1)
27             C[tid] = A[tid] - B[tid];//Vector subtraction
28         else if( tid % 4 == 2)
29             C[tid] = A[tid] * B[tid];//Vector multiplication
30         else if( tid % 4 == 3)
31             C[tid] = A[tid] / B[tid];//Vector division
32     }
33 }
34
35 int main(int argc, char **argv)
36 {
37     unsigned int data_size = 4194304;//Data size
38     float *A_host,*B_host,*C_host ;//Host Arrays
39     float *A_GPU,*B_GPU,*C_GPU;//Device Arrays
40
41     cudaMemcpy(A_GPU,A_host,sizeof(float)*data_size,cudaMemcpyHostToDevice);
42     cudaMemcpy(B_GPU,B_host,sizeof(float)*data_size,cudaMemcpyHostToDevice);
43
44     unsigned int NTB = 1024;//Number of threads in a block
45     dim3 threadsPerBlock(NTB);//Number of threads in a block
46     dim3 numBlocks(data_size/NTB);//Number of blocks in a grid
47
48     warp_no_divergence<<<numBlocks,threadsPerBlock>>>(A_GPU,B_GPU,C_GPU);//Launching 'warp_no_divergence' kernel
49     warp_divergence<<<numBlocks,threadsPerBlock>>>(A_GPU,B_GPU,C_GPU);//Launching 'warp_divergence' kernel
50     cudaDeviceSynchronize();//Waits until vector_add kernel completes its run
51 }
```

- Distribute the paths according to warp id
- First warp executes addition, second warp executes subtraction and so on

- Distribute the paths according to thread id
- First thread executes addition, second thread executes subtraction and so on

- Sufficiently number of paths (4)
- Sufficiently number of repetitions of the instruction (100)
- Memory operations are coalesced, so the divergence dominates the execution time
- We use **cudaEventRecord** in order to measure the kernel execution times

```
1 __global__ void warp_no_divergence(float *A,float *B,float *C)//No branching for the warps in 'if-elseif' structure
2 {
3     unsigned int tid = blockDim.x*blockIdx.x+threadIdx.x;//Global thread id
4 
5     for(unsigned int i=0;i<100;i++)
6     {
7         if( (tid/32) % 4 == 0)
8             C[tid] = A[tid] + B[tid];//Vector addition
9         else if( (tid/32) % 4 == 1)
10            C[tid] = A[tid] - B[tid];//Vector subtraction
11        else if( (tid/32) % 4 == 2)
12            C[tid] = A[tid] * B[tid];//Vector multiplication
13        else if( (tid/32) % 4 == 3)
14            C[tid] = A[tid] / B[tid];//Vector division
15    }
16 }
17 
18 __global__ void warp_divergence(float *A,float *B,float *C)//Four different paths for the warps in 'if-elseif' structure
19 {
20    unsigned int tid = blockDim.x*blockIdx.x+threadIdx.x;//Global thread id
21 
22    for(unsigned int i=0;i<100;i++)
23    {
24        if( tid % 4 == 0)
25            C[tid] = A[tid] + B[tid];//Vector addition
26        else if( tid % 4 == 1)
27            C[tid] = A[tid] - B[tid];//Vector subtraction
28        else if( tid % 4 == 2)
29            C[tid] = A[tid] * B[tid];//Vector multiplication
30        else if( tid % 4 == 3)
31            C[tid] = A[tid] / B[tid];//Vector division
32    }
33 }
34 
35 int main(int argc, char **argv)
36 {
37    unsigned int data_size = 4194304;//Data size
38    float *A_host,*B_host,*C_host ;//Host Arrays
39    float *A_CPU,*B_CPU,*C_CPU;//Device Arrays
```

```
Exec. Time of 'warp_no_divergence' kernel = 0.0124316
Exec. Time of 'warp_divergence' kernel = 0.0385476
Speed Up = 3.10078X
```

```
==23306== Profiling result:
==23306== Metric result:
Invocations                    Metric Name                 Metric Description        Min        Max        Avg
Device "Tesla K40c (0)"
    Kernel: warp_divergence(float*, float*, float*)
          1             warp_execution_efficiency    Warp Execution Efficiency     34.80%     34.80%     34.80%
    Kernel: warp_no_divergence(float*, float*, float*)
          1             warp_execution_efficiency    Warp Execution Efficiency    100.00%    100.00%    100.00%
```

Metric:

warp_execution_efficiency: Ratio of the average active threads per warp to the maximum number of threads per warp supported on a multiprocessor expressed as percentage

Occupancy

- is the ratio of active warps to the maximum number of resident warps supported on a multiprocessor
- is related with resource limitations of the SMX. These limitations are:
  - Maximum number of threads per multiprocessor (2048)
  - Maximum number of threads per block (1024)
  - Maximum number of blocks per multiprocessor (16)
  - Shared memory and registers
    - --ptxas-options=-v gives us the shared memory and register usage
- The main target is to find the optimum number of threads in a block in order to achieve maximum occupancy

Occupancy

- Set the block size as 64:
  - At most 16x64 (1024) threads can be active in a SMX
  - %50 theoretical occupancy
- Set the block size as 128:
  - At most 16x128 (2048) threads can be active in a SMX
  - %100 theoretical occupancy
- Set the block size as 1024:
  - At most 2x1024 (2048) threads can be active in a SMX
  - %100 theoretical occupancy

```
1 __global__ void vector_add(float *A,float *B,float *C)
2 {
3     int tid = blockDim.x*blockIdx.x+threadIdx.x;//Global thread id
4     for(int i=0;i<1000000;i++)
5     {
6         if( (tid/32) % 4 == 0)
7             C[tid] = A[tid] + B[tid];//Vector addition
8         else if( (tid/32) % 4 == 1)
9             C[tid] = A[tid] - B[tid];//Vector subtraction
10        else if( (tid/32) % 4 == 2)
11            C[tid] = A[tid] * B[tid];//Vector multiplication
12        else if( (tid/32) % 4 == 3)
13            C[tid] = A[tid] / B[tid];//Vector division
14    }
15 }
16
17 int main(int argc, char **argv)
18 {
19     int data_size;//Data size
20
21     float *A_host,*B_host,*C_host;//Host Arrays
22     float *A_GPU,*B_GPU,*C_GPU;//Device Arrays
23
24     int NTB;//Number of threads per block
25     if(data_size <= 1024)//Scenario 1 - Set NTB to data_size until 2^:
26         NTB = data_size;
27     else
28         NTB = 1024;
29     dim3 threadsPerBlockS1(NTB);//Number of threads in a block
30     dim3 numBlocksS1(data_size/NTB);//Number of blocks in a grid
31
32     vector_add<<<numBlocksS1,threadsPerBlockS1>>>(A_GPU,B_GPU,C_GPU);
33
34     if(data_size <= 512)//Scenario 2 - Increase NTB to its double
35         NTB = 32;
36     else if(data_size == 1024)
37         NTB = 64;
38     else if(data_size == 2048)
39         NTB = 128;
40     else if(data_size == 4096)
41         NTB = 256;
42     else if(data_size == 8192)
43         NTB = 512;
44     else
45         NTB = 1024;
46     dim3 threadsPerBlockS2(NTB);//Number of threads in a block
47     dim3 numBlocksS2(data_size/NTB);//Number of blocks in a grid
48
49     vector_add<<<numBlocksS2,threadsPerBlockS2>>>(A_GPU,B_GPU,C_GPU);
50
51 }
```

- We set the number of iterations as 1000000 so that ‘if-elseif’ structure dominates the execution time
- No warp divergence is occurred
- We use **cudaEventRecord** in order to measure the kernel execution times

In scenario 1, we set the number of threads as data_size until data_size is 2048

In scenario 2, we set the number of threads as 32 until data_size is 1024. Then we double the number of threads until data_size is 16384

Occupancy

data_size	Scenario 1			Scenario 2		
	# of threads	# of blocks	T. Occup.	# of threads	# of blocks	T. Occup.
32	32	1	%25	32	1	%25
64	64	1	%50	32	2	%25
128	128	1	%100	32	4	%25
256	256	1	%100	32	8	%25
512	512	1	%100	32	16	%25
1024	1024	1	%100	64	16	%50
2048	1024	2	%100	128	16	%100
4096	1024	4	%100	256	16	%100
8192	1024	8	%100	512	16	%100
16384	1024	16	%100	1024	16	%100
32768	1024	32	%100	1024	32	%100
65536	1024	64	%100	1024	64	%100

- Increasing theoretical occupancy does not always mean better time performance
- Utilization of streaming multiprocessors efficiently is also an important issue for the better time performance
- In the second scenario, we try to distribute the blocks to the SMs evenly

SM Efficiency

- **sm_efficiency** metric: The percentage of time at least one warp is active on a multiprocessor averaged over all multiprocessors on the GPU
  - The ratios of the running time of each SM to the total running time of the GPU is calculated. The average of these ratios is the result of the metric
- **achieved_occupancy** metric: the ratio of the average active warps per active cycle to the maximum number of warps supported on a multiprocessor
  - achieved_occupancy can not exceed the theoretical occupancy

Data Size	S1	S2	S1	S2	S1	S2	S1	S2	S1	S2	S1	S2
	# of threads		# of blocks		SM Efficiency		T. Occup.		Achieved Occ.		Exec. Time	
32	32	32	1	1	6.54%	6.55%	25%	25%	1.5%	1.5%	0.35	0.35
64	64	32	1	2	6.54%	12.46%	50%	25%	2.9%	1.5%	0.41	0.41
128	128	32	1	4	6.54%	18.83%	100%	25%	4.3%	1.5%	0.69	0.69
256	256	32	1	8	6.54%	37.17%	100%	25%	8.8%	1.5%	0.70	0.69
512	512	32	1	16	6.54%	68.48%	100%	25%	17%	1.6%	0.71	0.69
1024	1024	64	1	16	6.54%	78.32%	100%	50%	35%	2.8%	0.74	0.70
2048	1024	128	2	16	13.15%	96.32%	100%	100%	35%	4.7%	0.75	0.70
4096	1024	256	4	16	26.26%	95.91%	100%	100%	35%	9.4%	0.75	0.72
8192	1024	512	8	16	52.41%	95.10%	100%	100%	35%	18%	0.75	0.74
16384	1024	1024	16	16	83.29%	83.28%	100%	100%	39%	39%	0.91	0.91
32768	1024	1024	32	32	62.07%	62.11%	100%	100%	72%	72%	1.6	1.6
65536	1024	1024	64	64	72.80%	72.80%	100%	100%	77%	75%	2.49	2.49

- Values with green background mean the scenario is better than the other scenario for the corresponding output
- Values with yellow background mean both scenarios have the same values of parameters. Hence the values of the outputs are almost same
- Having better theoretical and achieved occupancy does not always mean better time performance
  - In this example, SM efficiency is more effective on the execution time of the kernel
  - Even though S1 has better occupancy, the execution times of S2 are better than those in S1 in some cases

Causes of Low Achieved Occupancy

1. Unbalanced workload within blocks
   - the warps in a block have unbalanced workload
2. Unbalanced workload across blocks
   - the blocks in a grid have unbalanced workload
3. Too few blocks launched
   - running few blocks in an SM than the maximum active blocks per SM
4. Partial last wave
   - maximum number of warps that can be active at once in an SM

- https://docs.nvidia.com/gameworks/content/developertools/desktop/analysis/report/cudaexperiments/kernellevel/achievedoccupancy.htm

Optimizing Parallel Reduction in CUDA

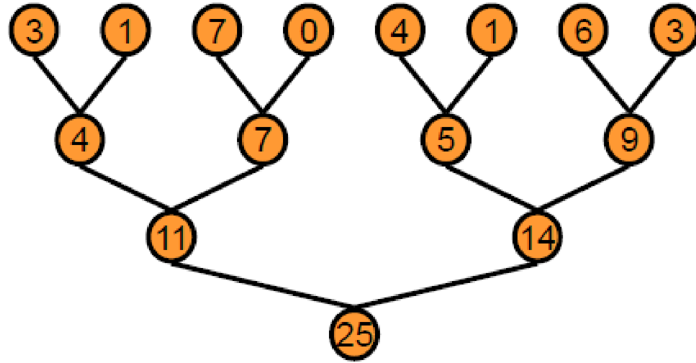
Mark Harris

NVIDIA Developer Technology

https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf

Parallel Reduction

- Tree based approach is used for each thread block



- We need too many blocks when:
  - the array size is very large
  - we want to keep all SMs on the GPU busy
- Each block does reduction over each portion of the array and produces a single output
- How do we combine the output of each block?

Ref: Mark Harris - https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf

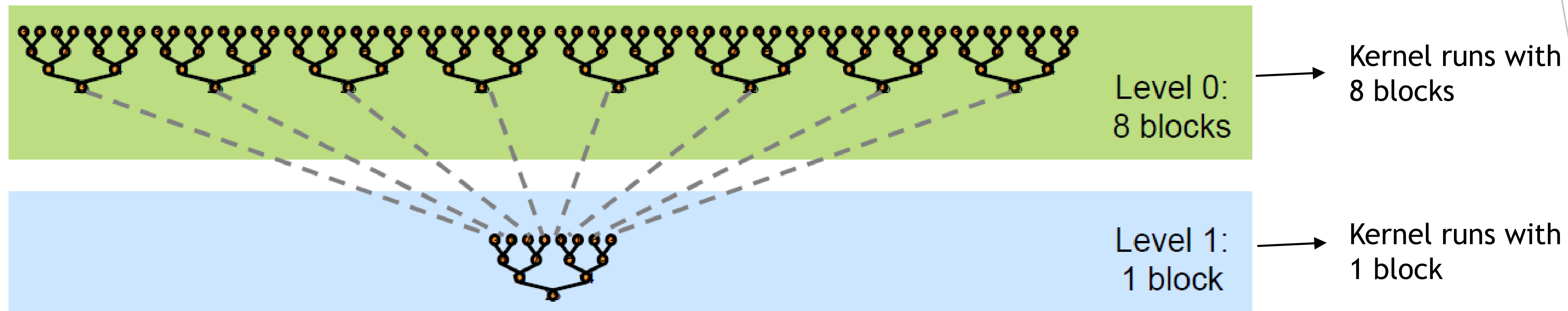
Global Synchronization

- Once the partial outputs of the blocks are produced, a synchronization mechanism between the blocks is needed in order to combine them.
- The blocks must wait in a synchronization point until all the blocks complete producing their outputs. Then recursive processes should follow to obtain overall output.

- In CUDA, the synchronization between the threads in the same block is possible.
- However, the threads in different blocks can not be synchronized each other.
  - Because the synchronization of the blocks is expensive for the GPUs whose processor counts is large
  - In fact, CUDA forces the programmers to create fewer blocks which may reduce the overall efficiency

- Using multiple kernels is a good solution. There is an implicit barrier between the kernel launches. The next kernel can not be executed before the current kernel finishes its execution.
- By the way, kernel launch has low overhead

Multiple Kernels and The Goal



Kernel runs with 8 blocks

Kernel runs with 1 block

- The code of both kernels is same. So we can launch them recursively.

- Since the reduction has low arithmetic cost, we should try to achieve high bandwidth.
- NVIDIA G80 is used in this experiment.
  - 86GB/s memory bandwidth

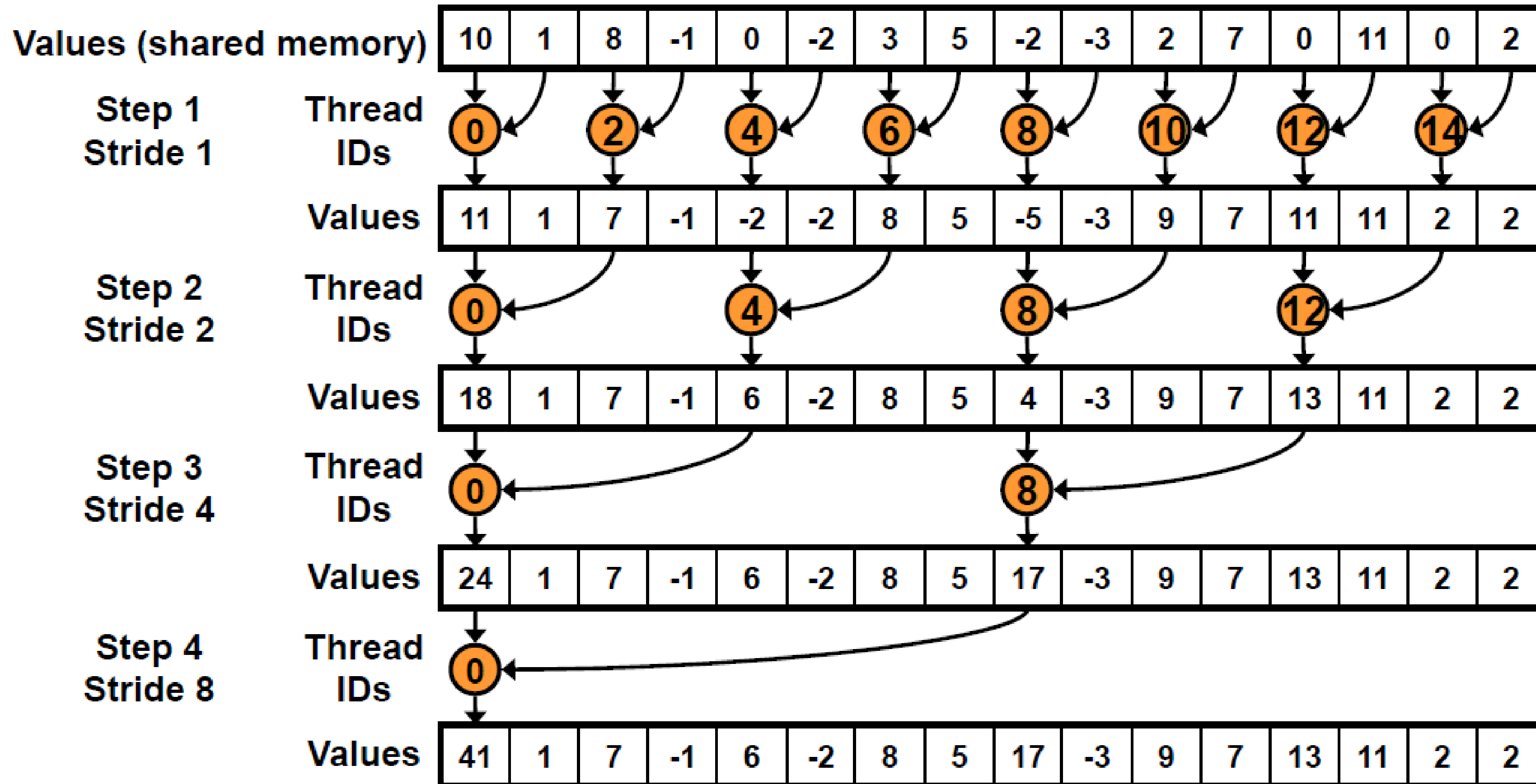
Ref: Mark Harris - https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf

Reduction 1: Interleaved Addressing

```
__global__ void reduce0(int *g_idata, int *g_odata) {
    extern __shared__ int sdata[];

    // each thread loads one element from global to shared mem
    unsigned int tid = threadIdx.x;                          → Local thread id
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;    → Global thread id
    sdata[tid] = g_idata[i];
    __syncthreads();                                          → Threads in the same block synchronize here

    // do reduction in shared mem
    for(unsigned int s=1; s < blockDim.x; s *= 2) {
        if (tid % (2*s) == 0) {                               → Threads whose local id is 0 or multiple of 2*s
            sdata[tid] += sdata[tid + s];
        }
        __syncthreads();
    }

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

Ref: Mark Harris - https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf

Interleaved Addressing 1



Ref: Mark Harris - https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf

Reduction 1: Interleaved Addressing

```
__global__ void reduce1(int *g_idata, int *g_odata) {
   extern __shared__ int sdata[];

   // each thread loads one element from global to shared mem
   unsigned int tid = threadIdx.x;
   unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
   sdata[tid] = g_idata[i];
   __syncthreads();

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

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

Problem: highly divergent warps are very inefficient, and % operator is very slow

Ref: Mark Harris - https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf

Performance for 4M element reduction

	Time (2^{22} ints)	Bandwidth
Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s

- The block size is 128 threads for all kernel launches

Ref: Mark Harris - https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf

Reduction 2: Interleaved Addressing

Just replace divergent branch in inner loop:

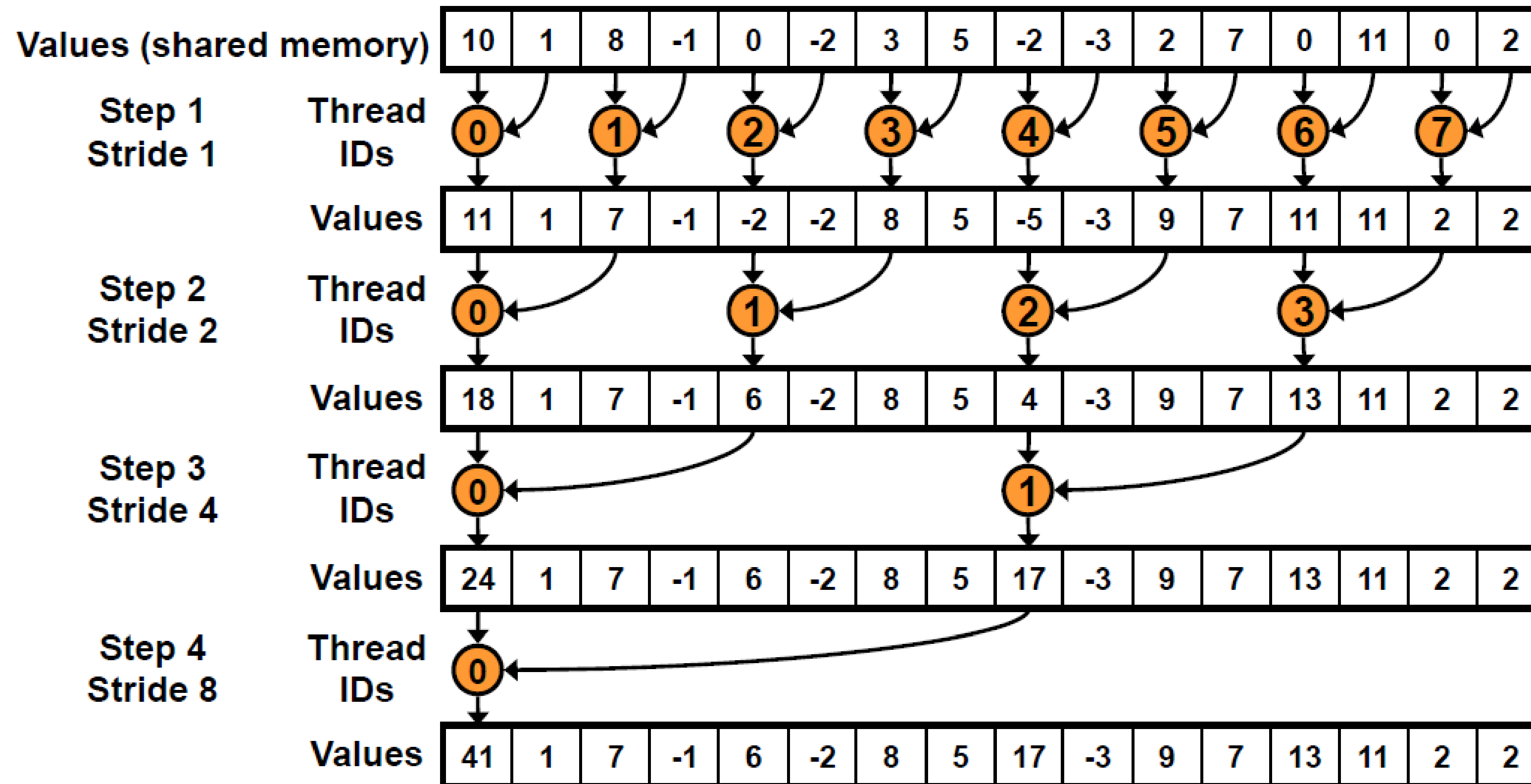
```
for (unsigned int s=1; s < blockDim.x; s *= 2)  {
    if (tid % (2*s) == 0) {
        sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
}
```

With strided index and non-divergent branch:

```
for (unsigned int s=1; s < blockDim.x; s *= 2)  {
    int index = 2 * s * tid;

    if (index < blockDim.x) {
        sdata[index] += sdata[index + s];
    }
    __syncthreads();
}
```

Ref: Mark Harris - https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf

Interleaved Addressing 2



Valid for first generation hardware

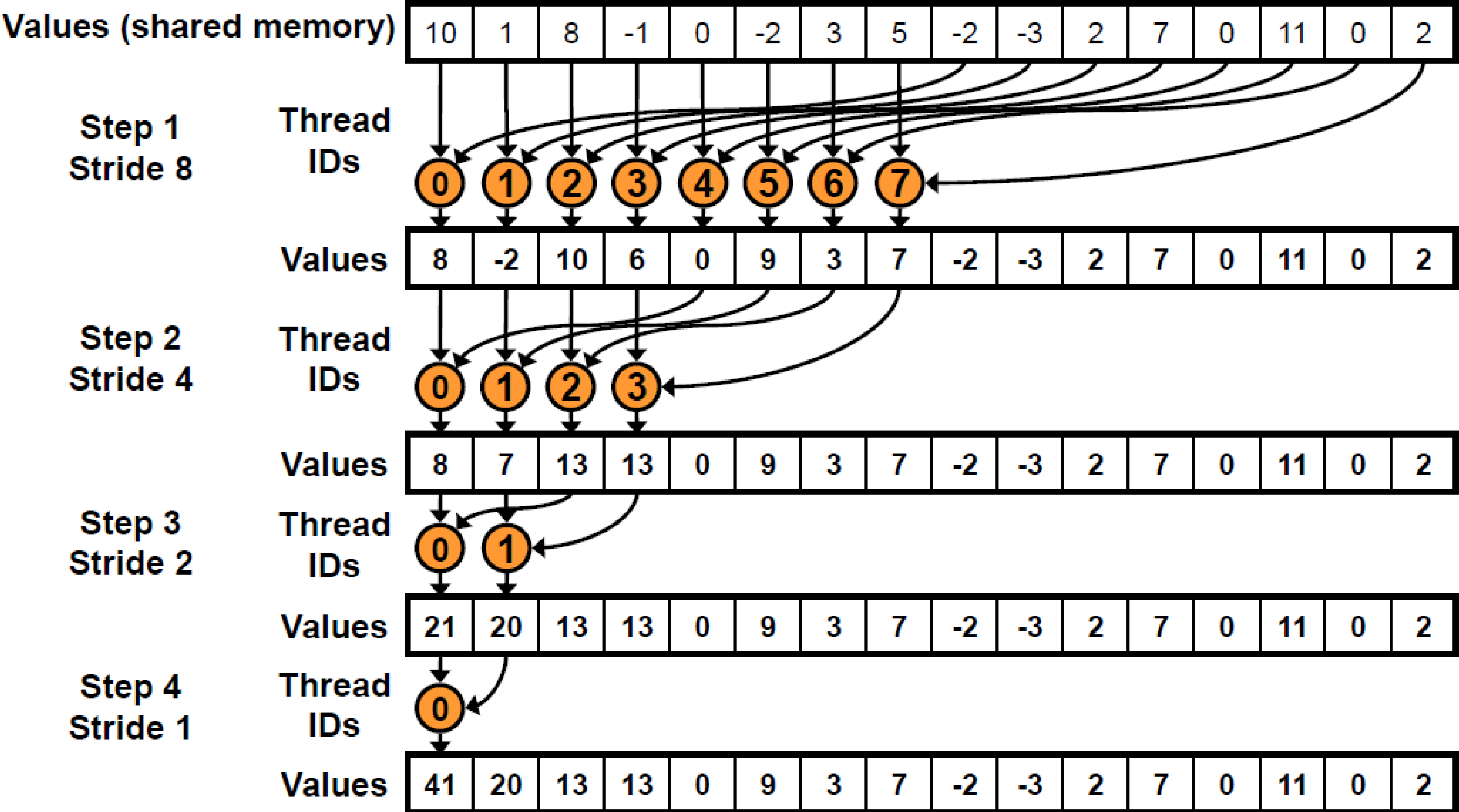
Ref: Mark Harris - https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf

Performance for 4M element reduction

	Time (2^{22} ints)	Bandwidth	Step Speedup	Cumulative Speedup
Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s		
Kernel 2: interleaved addressing with bank conflicts	3.456 ms	4.854 GB/s	2.33x	2.33x

Ref: Mark Harris - https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf

Sequential Addressing



Conflict Free

Ref: Mark Harris - https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf

Reduction 3: Sequential Addressing

Just replace strided indexing in inner loop:

```
for (unsigned int s=1; s < blockDim.x; s *= 2)  {
    int index = 2 * s * tid;

    if (index < blockDim.x) {
        sdata[index] += sdata[index + s];
    }
    __syncthreads();
}
```

With reversed loop and threadID-based indexing:

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

Ref: Mark Harris - https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf

Performance for 4M element reduction

	Time (2^{22} ints)	Bandwidth	Step Speedup	Cumulative Speedup
Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s		
Kernel 2: interleaved addressing with bank conflicts	3.456 ms	4.854 GB/s	2.33x	2.33x
Kernel 3: sequential addressing	1.722 ms	9.741 GB/s	2.01x	4.68x

Ref: Mark Harris - https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf

Idle Threads

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

- Half of the threads are idle in the first iteration of the loop. Wasteful!

Ref: Mark Harris - https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf

Reduction 4: First Add During Load

Halve the number of blocks, and replace single load:

```
// each thread loads one element from global to shared mem
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
sdata[tid] = g_idata[i];
__syncthreads();
```

With two loads and first add of the reduction:

```
// perform first level of reduction,
// reading from global memory, writing to shared memory
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*(blockDim.x*2) + threadIdx.x;
sdata[tid] = g_idata[i] + g_idata[i+blockDim.x];
__syncthreads();
```

Ref: Mark Harris - https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf

Performance for 4M element reduction

	Time (2^{22} ints)	Bandwidth	Step Speedup	Cumulative Speedup
Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s		
Kernel 2: interleaved addressing with bank conflicts	3.456 ms	4.854 GB/s	2.33x	2.33x
Kernel 3: sequential addressing	1.722 ms	9.741 GB/s	2.01x	4.68x
Kernel 4: first add during global load	0.965 ms	17.377 GB/s	1.78x	8.34x

Ref: Mark Harris - https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf

Unrolling the Last Warp

- 17 GB/s is very low
- Loop overhead occurs!
- The number of the active threads reduces at each iteration of the loop
- When s < 32, only one warp is active!
  - Since the threads in a warp execute the same instruction at a time, we do not need to synchronize them and do not need __syncthreads() function
  - No need to evaluate last 6 iterations of the loop for the remaining warps
  - So one can unroll the last 6 iterations of the loop

Reduction 5: Unroll the Last Warp

```
__device__ void warpReduce(volatile int* sdata, int tid) {
      sdata[tid] += sdata[tid + 32];
      sdata[tid] += sdata[tid + 16];
      sdata[tid] += sdata[tid +  8];
      sdata[tid] += sdata[tid +  4];
      sdata[tid] += sdata[tid +  2];
      sdata[tid] += sdata[tid +  1];
 }
```

IMPORTANT:
For this to be correct,
we must use the
"volatile" keyword!

Disabling optimizations (such as caching)

```
// later…
 for (unsigned int s=blockDim.x/2; s>32; s>>=1) {
      if (tid < s)
         sdata[tid] += sdata[tid + s];
      __syncthreads();
 }

 if (tid < 32) warpReduce(sdata, tid);
```

Ref: Mark Harris - https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf

Performance for 4M element reduction

	Time (2^{22} ints)	Bandwidth	Step Speedup	Cumulative Speedup
Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s		
Kernel 2: interleaved addressing with bank conflicts	3.456 ms	4.854 GB/s	2.33x	2.33x
Kernel 3: sequential addressing	1.722 ms	9.741 GB/s	2.01x	4.68x
Kernel 4: first add during global load	0.965 ms	17.377 GB/s	1.78x	8.34x
Kernel 5: unroll last warp	0.536 ms	31.289 GB/s	1.8x	15.01x

Ref: Mark Harris - https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf

Complete Unrolling

- Can we unroll all of the iterations of the loop?
- Since the possible values of the block size are fixed and known at compile time and there is an upper limit on it, one can completely unroll the iterations of the loop.
  - Using C++ templates enables us to do it

```
template <unsigned int blockSize>
__global__ void reduce5(int *g_idata, int *g_odata)
```

Function template parameter

Ref: Mark Harris - https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf

Completely Unrolled

```
Template <unsigned int blockSize>
__device__ void warpReduce(volatile int* sdata, int tid) {
        if (blockSize >= 64) sdata[tid] += sdata[tid + 32];
        if (blockSize >= 32) sdata[tid] += sdata[tid + 16];
        if (blockSize >= 16) sdata[tid] += sdata[tid +  8];
        if (blockSize >=  8) sdata[tid] += sdata[tid +  4];
        if (blockSize >=  4) sdata[tid] += sdata[tid +  2];
        if (blockSize >=  2) sdata[tid] += sdata[tid +  1];
 }
```

```
    if (blockSize >= 512) {
        if (tid < 256) { sdata[tid] += sdata[tid + 256]; } __syncthreads(); }
    if (blockSize >= 256) {
        if (tid < 128) { sdata[tid] += sdata[tid + 128]; } __syncthreads(); }
    if (blockSize >= 128) {
        if (tid <  64)  { sdata[tid] += sdata[tid +   64]; } __syncthreads(); }

    if (tid < 32) warpReduce<blockSize>(sdata, tid);
```

- The maximum block size is 1024 in newer GPUs !
- Reds are evaluated at compile time

Ref: Mark Harris - https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf

Invoking Template Kernels

```
switch (threads)
    {
    case 512:
      reduce5<512><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case 256:
      reduce5<256><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case 128:
      reduce5<128><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case 64:
      reduce5< 64><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case 32:
      reduce5< 32><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case 16:
      reduce5< 16><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case  8:
      reduce5<  8><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case  4:
      reduce5<  4><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case  2:
      reduce5<  2><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case  1:
      reduce5<  1><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    }
```

- 10 + 1 (1024) = 11 possible block sizes

Ref: Mark Harris - https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf

Performance for 4M element reduction

	Time (2^{22} ints)	Bandwidth	Step Speedup	Cumulative Speedup
Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s		
Kernel 2: interleaved addressing with bank conflicts	3.456 ms	4.854 GB/s	2.33x	2.33x
Kernel 3: sequential addressing	1.722 ms	9.741 GB/s	2.01x	4.68x
Kernel 4: first add during global load	0.965 ms	17.377 GB/s	1.78x	8.34x
Kernel 5: unroll last warp	0.536 ms	31.289 GB/s	1.8x	15.01x
Kernel 6: completely unrolled	0.381 ms	43.996 GB/s	1.41x	21.16x

Ref: Mark Harris - https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf

Cost and Algorithm Cascading

- Step Complexity: O(log N)
- Work Complexity: O(N)
- Time Complexity: O(log N)
- Processor Time Complexity: O(Nlog N)

- Algorithm Cascading: Combining sequential and parallel reduction
  - Each thread performs loading and summing multiple elements sequentially and store the result to the shared memory
  - Tree based reduction performs through shared memory in parallel

Reduction 7: Multiple Adds / Thread

Replace load and add of two elements:

```
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*(blockDim.x*2) + threadIdx.x;
sdata[tid] = g_idata[i] + g_idata[i+blockDim.x];
__syncthreads();
```

With a while loop to add as many as necessary:

```
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*(blockSize*2) + threadIdx.x;
unsigned int gridSize = blockSize*2*gridDim.x;
sdata[tid] = 0;

while (i < n) {
    sdata[tid] += g_idata[i] + g_idata[i+blockSize];
    i += gridSize;
}
__syncthreads();
```

Decreasing the number of blocks means increasing the number of iterations of 'while' loop

A balance between the level of the tree and the iteration of the loop should be maintained

Maintain coalescing

Ref: Mark Harris - https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf

Performance for 4M element reduction

	Time (2^{22} ints)	Bandwidth	Step Speedup	Cumulative Speedup
Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s		
Kernel 2: interleaved addressing with bank conflicts	3.456 ms	4.854 GB/s	2.33x	2.33x
Kernel 3: sequential addressing	1.722 ms	9.741 GB/s	2.01x	4.68x
Kernel 4: first add during global load	0.965 ms	17.377 GB/s	1.78x	8.34x
Kernel 5: unroll last warp	0.536 ms	31.289 GB/s	1.8x	15.01x
Kernel 6: completely unrolled	0.381 ms	43.996 GB/s	1.41x	21.16x
Kernel 7: multiple elements per thread	0.268 ms	62.671 GB/s	1.42x	30.04x

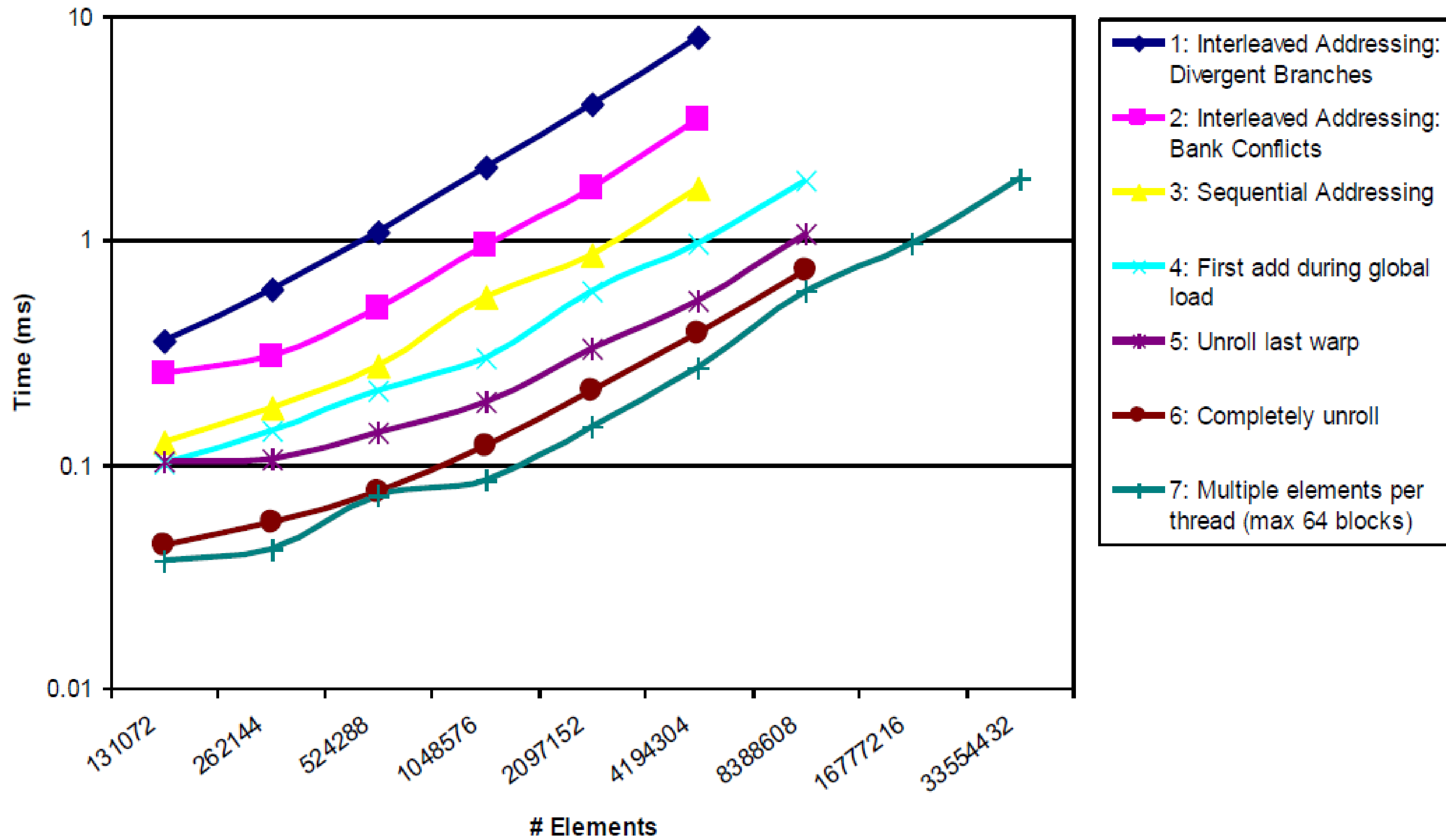
Ref: Mark Harris - https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf

Final Optimized Kernel

```
template <unsigned int blockSize>
__device__ void warpReduce(volatile int *sdata, unsigned int tid) {
   if (blockSize >=  64) sdata[tid] += sdata[tid + 32];
   if (blockSize >=  32) sdata[tid] += sdata[tid + 16];
   if (blockSize >=  16) sdata[tid] += sdata[tid +  8];
   if (blockSize >=   8) sdata[tid] += sdata[tid +  4];
   if (blockSize >=   4) sdata[tid] += sdata[tid +  2];
   if (blockSize >=   2) sdata[tid] += sdata[tid +  1];
}
template <unsigned int blockSize>
__global__ void reduce6(int *g_idata, int *g_odata, unsigned int n) {
   extern __shared__ int sdata[];
   unsigned int tid = threadIdx.x;
   unsigned int i = blockIdx.x*(blockSize*2) + tid;
   unsigned int gridSize = blockSize*2*gridDim.x;
   sdata[tid] = 0;

   while (i < n) { sdata[tid] += g_idata[i] + g_idata[i+blockSize];  i += gridSize;  }
   __syncthreads();

   if (blockSize >= 512) { if (tid < 256) { sdata[tid] += sdata[tid + 256]; } __syncthreads(); }
   if (blockSize >= 256) { if (tid < 128) { sdata[tid] += sdata[tid + 128]; } __syncthreads(); }
   if (blockSize >= 128) { if (tid <  64) { sdata[tid] += sdata[tid +  64]; } __syncthreads(); }

   if (tid < 32) warpReduce(sdata, tid);
   if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

Ref: Mark Harris - https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf

Performance Comparison



Ref: Mark Harris - https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf

References

- https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-product-literature/NVIDIA-Kepler-GK110-GK210-Architecture-Whitepaper.pdf
- https://docs.nvidia.com/cuda/pdf/CURAND_Library.pdf
- https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__EVENT.html
- https://docs.nvidia.com/cuda/profiler-users-guide
- Dülger, Ö., Oğuztüzün, H. & Demirekler, M. Memory Coalescing Implementation of Metropolis Resampling on Graphics Processing Unit. J Sign Process Syst 90, 433–447 (2018) https://rdcu.be/cLz8N
- https://docs.nvidia.com/cuda/cuda-occupancy-calculator/index.html
- https://docs.nvidia.com/gameworks/content/developertools/desktop/analysis/report/cudaexperiments/kernellevel/achievedoccupancy.htm
- https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf