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

EURO

TRUBA
Turkish Science e-Infrastructure

TÜBİTAK
ULAKBİM

Sabancı Üniversitesi

ORTA DOĞU TEKNİK ÜNİVERSİTESİ
MIDDLE EAST TECHNICAL UNIVERSITY

Contents

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

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

Some Libraries and Remaining Issues

- Thrust Library
- CUB Library
- Comparison of three parallel reduction algorithms
- cuBlas Library
- Remaining Issues
  - Usage of cudaMemset
  - Initializing cost of the states of RNGs
  - L1 cache in Tesla K40

Thrust Library

- is a CUDA C++ template library
- performs data parallel operations such as:
  - scan
  - sort
  - Reduce

- https://docs.nvidia.com/cuda/thrust/index.html
- https://thrust.github.io/doc/index.html

Thrust Library

- Two kind of vectors in Thrust:
  - **host_vector**: stored in the memory of host
  - **device_vector**: stored in the device memory of GPU
- Host memory is allocated as below:
  - thrust::host_vector<int> H(4);
  - has 4 elements
  - we can initialize it directly by copying a value as in the regular arrays (e.g. H[0] = 4)
- Device memory is allocated as below:
  - thrust::device_vector<int> D;
  - We can initialize it in when it is being created
    - thrust::device_vector<int> D = H;
- Copying host vector to host vector or device vector to device vector is also possible

Thrust Library

```
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>

#include <thrust/copy.h>
#include <thrust/fill.h>
#include <thrust/sequence.h>

int main(void)
{
    // Creating a device array with ten elements and setting all the elements to 9
    thrust::device_vector<int> D(10, 9);

    // Setting the first five elements to 10
    thrust::fill(D.begin(), D.begin() + 5, 10);

    // Creating a host array and initializing its elements with the values of first 3 elements of D
    thrust::host_vector<int> H(D.begin(), D.begin() + 3);//The size of H is 3

    // Setting the elements of H to 0, 1, 2
    thrust::sequence(H.begin(), H.end());

    // Copying all of the elements of H back to the beginning of D consecutively
    thrust::copy(H.begin(), H.end(), D.begin());
}
```

→ Corresponding thrust libraries

```
D[0]    D[1]    D[2]    D[3]    D[4]    D[5]    D[6]    D[7]    D[8]    D[9]
0        1       2       10      10      9       9       9       9       9
H[0]    H[1]    H[2]
0        1       2
```

Thrust Library

- We can use a raw pointer in a thrust function
  - int * raw_ptr;

    cudaMalloc((void **) &raw_ptr, N * sizeof(int));

    thrust::device_ptr<int> dev_ptr(raw_ptr); -> it is ready to be used in thrust function

    thrust::fill(dev_ptr, dev_ptr + N, (int) 0);
- Reverse is also possible. We can extract raw pointer from thrust device pointer
  - thrust::device_ptr<int> dev_ptr = thrust::device_malloc<int>(N);

    int * raw_ptr = thrust::raw_pointer_cast(dev_ptr); -> it is ready to be used as regular array

Algorithms in Thrust Library

- **1) Transformations:** applies an operation to the each element in a set of input range and store the result in a destination range

```
#include <thrust/device_vector.h>
#include <thrust/transform.h>
#include <thrust/sequence.h>
#include <thrust/copy.h>
#include <thrust/fill.h>
#include <thrust/replace.h>
#include <thrust/functional.h>
#include <iostream>

int main(void)
{
    // Creating three device_vectors with 10 elements
    thrust::device_vector<int> X(10);
    thrust::device_vector<int> Y(10);
    thrust::device_vector<int> Z(10);

    // Setting the elements of X to 0,1,2,3,...,9 consecutively
    thrust::sequence(X.begin(), X.end());

    // Setting the elements of Y to the negation of each element of X
    thrust::transform(X.begin(), X.end(), Y.begin(), thrust::negate<int>());

    // Setting all the elements of Z to three
    thrust::fill(Z.begin(), Z.end(), 3);

    // Modulus operation Y = X mod Z
    thrust::transform(X.begin(), X.end(), Z.begin(), Y.begin(), thrust::modulus<int>());

    // Replacing all the twos in Y with fives
    thrust::replace(Y.begin(), Y.end(), 2, 5);

    // Printing Y
    thrust::copy(Y.begin(), Y.end(), std::ostream_iterator<int>(std::cout, "\n"));

    return 0;
}
```

```
0
1
5
0
1
5
0
1
5
0
```

Algorithms in Thrust Library

- 2) **Reductions**: uses a binary operation to reduce an input sequence to a single value. Some of the operations are:
  - sum of an array of numbers
  - maximum of an array of numbers
  - minimum of an array of numbers
- **thrust::reduce**: is the function of reduction operations
- int sum = thrust::reduce(D.begin(), D.end(), (int) 0, thrust::plus<int>());
  - reduce function for the summation of the numbers in an array
  - fourth parameter determines the type of operation
  - third parameter is the initial value of the output
- Below also are the implementations of the sum of an array
  - int sum = thrust::reduce(D.begin(), D.end(), (int) 0);
  - int sum = thrust::reduce(D.begin(), D.end());

Algorithms in Thrust Library

- Other reduction operations are:
  - *thrust*::*min_element*: Finds the smallest element in the range
  - *thrust*::*max_element*: Finds the biggest element in the range
  - *thrust*:: *is_sorted*: Checks the numbers in the range are in ascending order or not
  - *thrust*::*count_if*: Finds the number of elements in the range which satisfy the given predicate
  - *thrust*::*inner_product*: Computes the inner product of two vectors

Algorithms in Thrust Library

- 3) **Prefix-Sums**: Performs scan operation:
  - inclusive scan operation
  - exclusive scan operation
- **thrust::inclusive_scan**: Performs in-place scan. The default operator is plus op.
  - int data[6] = {1, 0, 2, 2, 1, 3};

    thrust::inclusive_scan(data, data + 6, data);

    // data is now {1, 1, 3, 5, 6, 9}
- **thrust::exclusive_scan**: Similar to the inclusive_scan but shifted by one place to the right
  - int data[6] = {1, 0, 2, 2, 1, 3};

    thrust::exclusive_scan(data, data + 6, data);

    // data is now {0, 1, 1, 3, 5, 6}

Algorithms in Thrust Library

- 4) **Reordering**: Performs partitioning and stream compaction operations:
  - *copy_if*: Copy the elements that satisfy the given predicate in the first array to the second array
  - *partition*: Reorder the elements so that the numbers that satisfy the given predicate precede the numbers that do not satisfy the given predicate
  - *remove and remove_if*: Remove the elements from the array which are equal to given value or which do not satisfy the given predicate
  - *unique*: Remove the consecutive duplicates of the numbers in the array

Algorithms in Thrust Library

- 5) **Sorting**: Sorts data or reorders data according to given criterion
  - *thrust::sort*: Sort the elements of the array in ascending order
    - int A[6] = {1, 4, 2, 8, 5, 7};

      thrust::sort(A, A + 6);

      // A is now {1, 2, 4, 5, 7, 8}
  - *thrust::sort_by_key*: Sort the key-value pairs stored in different arrays
    - int keys[6] = { 1, 4, 2, 8, 5, 7};

      char values[6] = {'a', 'b', 'c', 'd', 'e', 'f'};

      thrust::sort_by_key(keys, keys + 6, values);

      // keys is now { 1, 2, 4, 5, 7, 8}

      // values is now {'a', 'c', 'b', 'e', 'f', 'd'}
  - *thrust::stable_sort*: Sort the elements of the array according to given comparison operator
    - int A[6] = {1, 4, 2, 8, 5, 7};

      thrust::stable_sort(A, A + 6, thrust::greater<int>());

      // A is now {8, 7, 5, 4, 2, 1}

CUB (CUDA UnBound) Library

- is a generic, configurable C++ template library for high performance CUDA primitives
- targets the each layer of the CUDA programming model
- reusable software components are:
  - Parallel primitives:
    - Warp-wide "collective" primitives
    - Block-wide "collective" primitives
    - Device-wide primitives
  - Utilities:
    - Fancy iterators
    - Thread and thread block I/O
    - PTX intrinsics
    - Device, kernel, and storage management

- https://nvlabs.github.io/cub/index.html

CUB Library

- **Parallel primitives:**
  - ***Warp-wide "collective" primitives***
    - Cooperative warp-wide prefix scan, reduction, etc.
    - Safely specialized for each underlying CUDA architecture
  - ***Block-wide "collective" primitives***
    - Cooperative I/O, sort, scan, reduction, histogram, etc.
    - Compatible with arbitrary thread block sizes and types
  - ***Device-wide primitives***
    - Parallel sort, prefix scan, reduction, histogram, etc.
    - Compatible with CUDA dynamic parallelism

- Ref: https://nvlabs.github.io/cub/index.html

CUB's collective primitives

- Collectives allow complex parallel code:
  - to be re-used rather than re-implemented
  - to be re-compiled rather than hand-ported
- are not bound to any particular width of parallelism or data type. So they can be:
  - Adaptable to fit the needs of the enclosing kernel computation
  - Trivially tunable to different grain sizes (threads per block, items per thread, etc.)
- Thus CUB is CUDA Unbound

- Ref: https://nvlabs.github.io/cub/index.html

CUB

- When writing a kernel code considering the issues such as deadlock, bottleneck, throughput, shared memory layout, synchronization, granularity, latency is very challenging and time consuming process
  - CUB's collectives consider these issues in an efficient way
- Also CUB's primitives allow the programmers to determine the proper value of parameters of the kernel easily in order to use the processor resources of the GPU architecture efficiently
- In addition, the programmers do not need to write a kernel code for the different cases of a particular operation such as reduction, scan. Just using the template of CUB's primitive is enough

CUB

- A code snippet of device parallel reduction:

```
#include <cub/cub.cuh>

// Declare, allocate, and initialize device-accessible pointers for input and output
int  num_items;      // e.g., 7
int  *d_in;          // e.g., [8, 6, 7, 5, 3, 0, 9]
int  *d_out;         // e.g., [-]

...

// Determine temporary device storage requirements
void     *d_temp_storage = NULL;
size_t   temp_storage_bytes = 0;
cub::DeviceReduce::Sum(d_temp_storage, temp_storage_bytes, d_in, d_out, num_items);

// Allocate temporary storage
cudaMalloc(&d_temp_storage, temp_storage_bytes);

// Run sum-reduction
cub::DeviceReduce::Sum(d_temp_storage, temp_storage_bytes, d_in, d_out, num_items);
// d_out <-- [38]
```

Ref: https://nvlabs.github.io/cub/index.html

CUB vs Thrust

- Both provide device primitives for programmers
  - Moreover, CUB provides SIMT block wide and warp wide primitives
- Thrust is not CUDA specific
  - Does not allow the programmers to interfere CUDA specific details (the number of threads, cudaStream_t parameters etc.)
  - Because of that, the primitives can not be configured for a specific GPU architecture
- CUB is CUDA C++ specific
  - Allows the programmers to interfere CUDA specific details (the number of threads, cudaStream_t parameters etc.)
  - So the primitives can be configured for a specific GPU architecture

Comparison of Parallel Reduction Algorithms

- Three different approaches:
  - Efficient parallel reduction by Mark Harris
  - Thrust reduce primitive
  - CUB DeviceReduce primitive
- Tesla K40 Board
- 4M elements (2^{22})
- We measure the execution times of reduce kernels or reduce functions
- The kernel execution times of the approaches:
  - Parallel reduction by Harris: 0.147 milliseconds
  - Thrust library: 0.490 milliseconds
  - CUB library: 0.125 milliseconds

Comparison of Parallel Reduction Algorithms

- Thrust has the worst execution time since it does not consider the resources of the specific GPU architecture and CUDA specific details
- Further, we can not modify the implementation as we needed, so it is hard to improve the performance of a Thrust function

- On the other hand, parallel reduction by Harris and CUB have similar execution time since they consider the specific GPU hardware and CUDA specific details
- With the parallel reduction algorithm of Harris, we can implement our parallel reduction method by using CUDA built-in functions and APIs. We can modify the implementation easily when we need to:
  - Replace the operation in reduce process
  - Add new parameters to the algorithm
  - Improve the performance
  - Adapt our implementation to new architectures in the future

- Although CUB is an open source library, it is hard to modify the functions in the library. Because understanding and examining the codes of someone else are a challenging process. Furthermore, CUB collectives do not suggest re-implementing the parallel code.

Parallel Reduction by Harris

```
T mySum = 0;
while (i < n)
{
    mySum += g_idata[i];

    if (nIsPow2 || i + blockSize < n)
        mySum += g_idata[i+blockSize];

    i += gridSize;
}
```

→ Summing the elements of an array

```
T myMax = 0;
while (i < n)
{
    if(myMax < g_idata[i])
        myMax = g_idata[i];

    if (nIsPow2 || i + blockSize < n)
        if(myMax < g_idata[i+blockSize])
            myMax = g_idata[i+blockSize];

    i += gridSize;
}
```

→ Finding maximum element of an array (all elements > 0)

Parallel Reduction by Harris

```
T myMax = 0;
T myMax_Related1;
T myMax_Related2;
T myMax_Related3;
while (i < n)
{
    if(myMax < g_idata[i])
    {
        myMax = g_idata[i];

        myMax_Related1 = g_idata_Related1[i];
        myMax_Related2 = g_idata_Related2[i];
        myMax_Related3 = g_idata_Related3[i];
    }

    if (nIsPow2 || i + blockSize < n)
        if(myMax < g_idata[i+blockSize])
        {
            myMax = g_idata[i+blockSize];

            myMax_Related1 = g_idata_Related1[i+blockSize];
            myMax_Related2 = g_idata_Related2[i+blockSize];
            myMax_Related3 = g_idata_Related3[i+blockSize];
        }

    i += gridSize;
}
if (tid == 0)
{
    g_odata[blockIdx.x] = sdata[0];
    g_odata_Related1[blockIdx.x] = sdata_Related1[0];
    g_odata_Related2[blockIdx.x] = sdata_Related2[0];
    g_odata_Related3[blockIdx.x] = sdata_Related3[0];
}
```

- Arrays that are related to main array (g_idata)
- We carry them along with main array

CUDA Libraries

- **cuBLAS:** implementation of BLAS (Basic Linear Algebra Subprograms) on CUDA
- **cuSPARSE:** a set of BLA subroutines used for handling sparse matrices
- **cuRAND:** API for generation of high-quality pseudorandom and quasirandom numbers
- **cuSOLVER:** provides LAPACK-like features such as common matrix factorization
- **cuFFT:** a CUDA Fast Fourier Transform library
- **nvJPEG:** high-performance, GPU-accelerated JPEG encoding and decoding functionality
- **NPP:** functions for performing CUDA-accelerated 2D image and signal processing
- **cuTENSOR:** a first-of-its-kind GPU-accelerated tensor linear algebra library
- **cuSPARSELt:** provides high-performance structured matrix-dense matrix multiplication
- **nvJPEG2000:** provides high-performance GPU-accelerated JPEG2000 decoding

- https://docs.nvidia.com/cuda-libraries/index.html

Remaining Issues

- Usage of cudaMemset function:
  - _host__cudaError_t cudaMemset (void* devPtr, int value, size_t count)

```
__global__ void print_array(int *A)
{
    printf("%d\n",A[0]);
    printf("%d\n",A[1]);
    printf("%d\n",A[2]);
}

int main()
{
    int *A_GPU;
    cudaMalloc((void**)&A_GPU,sizeof(int)*3);
    cudaMemset(A_GPU, 0, 12);
    print_array<<<1,1>>>(A_GPU);
    cudaFree(A_GPU);
}
```

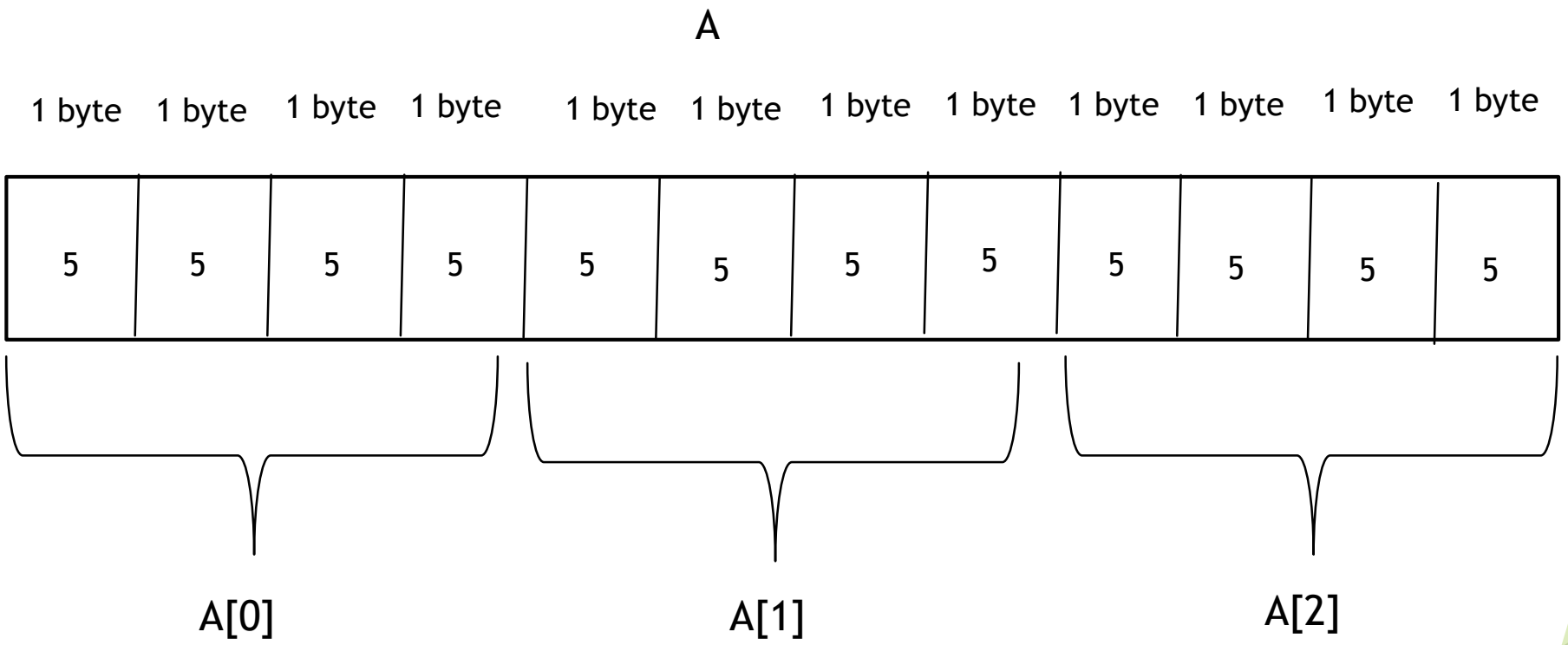
Output:
0
0
0

Question: How can we set all the elements of the array to 5?

Remaining Issues

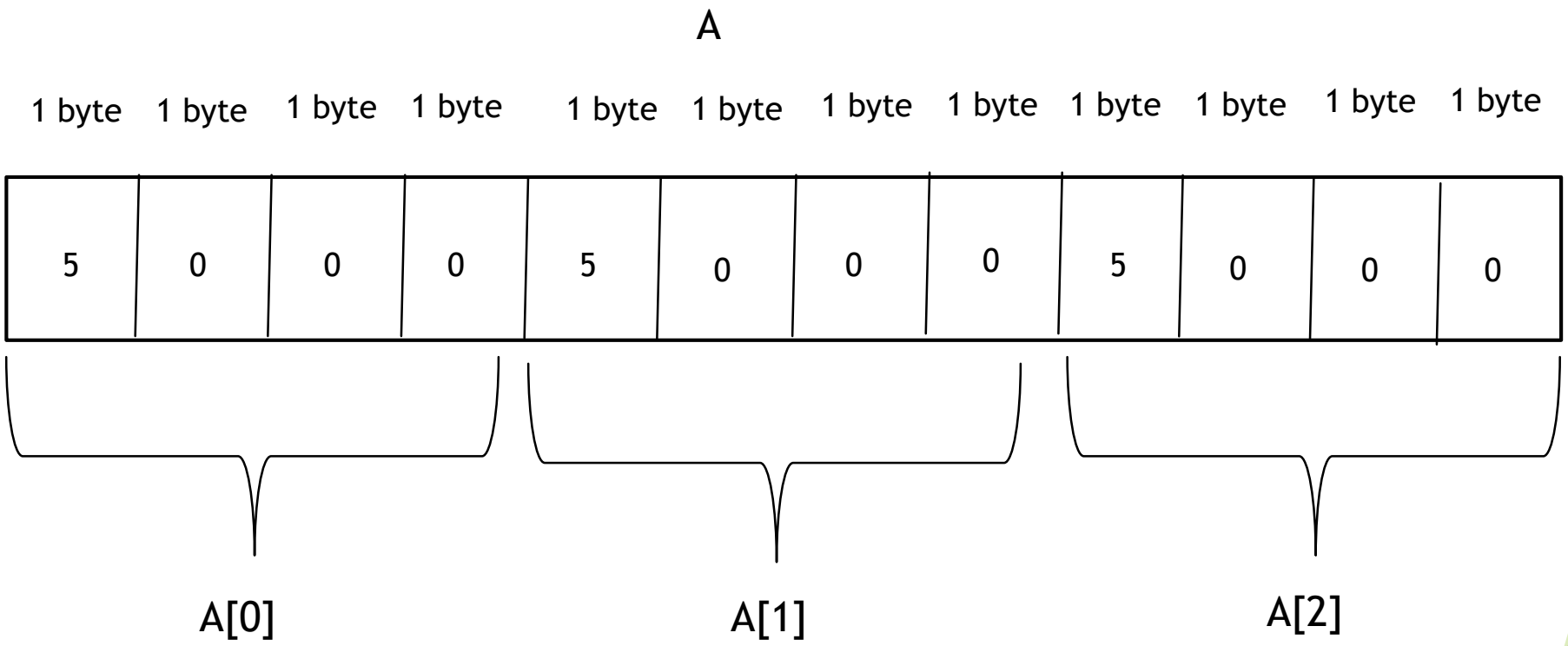
- cudaMemset(A_GPU, 5, 12); is this correct? Answer is no?

Output:
84215045
84215045
84215045



Remaining Issues

- Below are correct values! How can we set each byte?



Remaining Issues

- How can we set each elements to 5?

```
cudaMemset(A_GPU, 0, 12);
cudaMemset(A_GPU+0, 5, 1)
cudaMemset(A_GPU+1, 5, 1)
cudaMemset(A_GPU+2, 5, 1)
```

First way

Second way

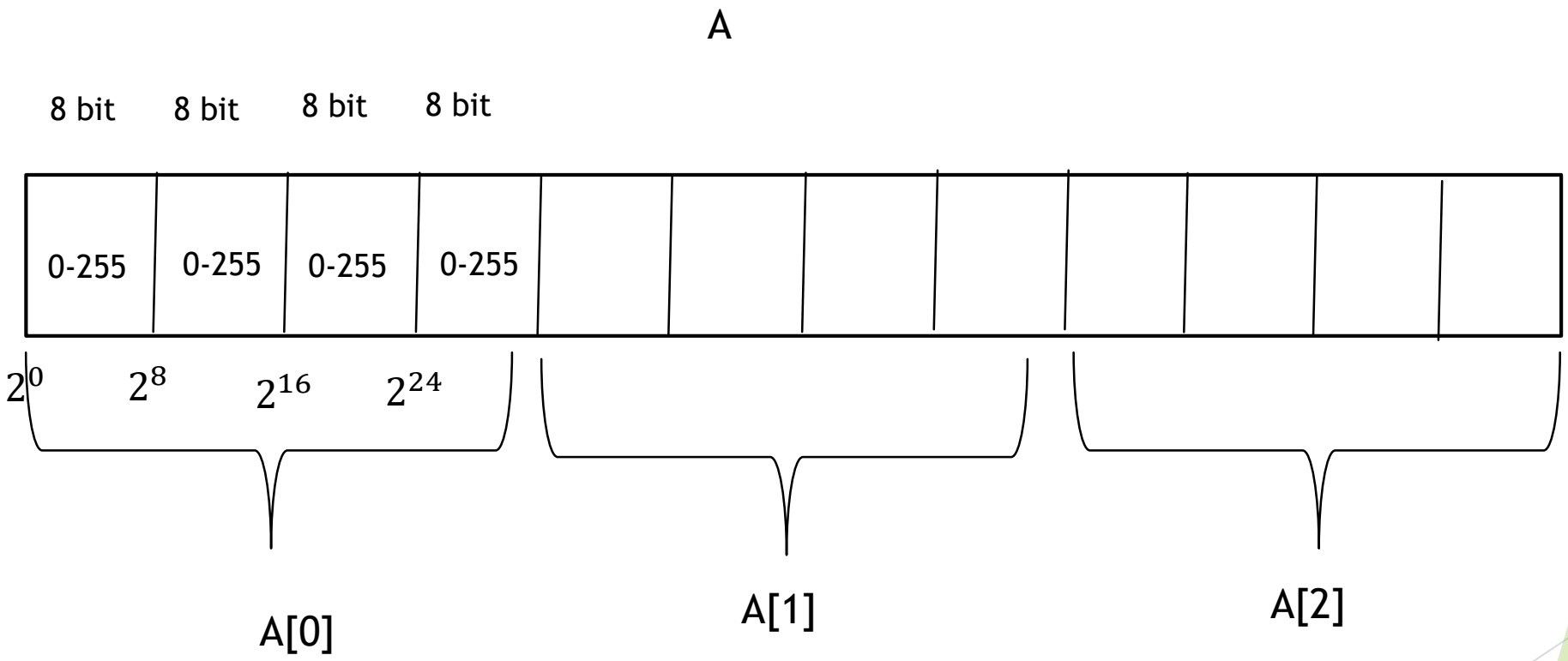
```
cudaMemset(A_GPU, 0, 12);
cudaMemset((int*) ((char*)&A_GPU[0] + 0), 5, 1);
cudaMemset((int*) ((char*)&A_GPU[1] + 0), 5, 1);
cudaMemset((int*) ((char*)&A_GPU[2] + 0), 5, 1);
```

- How can we set first element to 300?

Remaining Issues

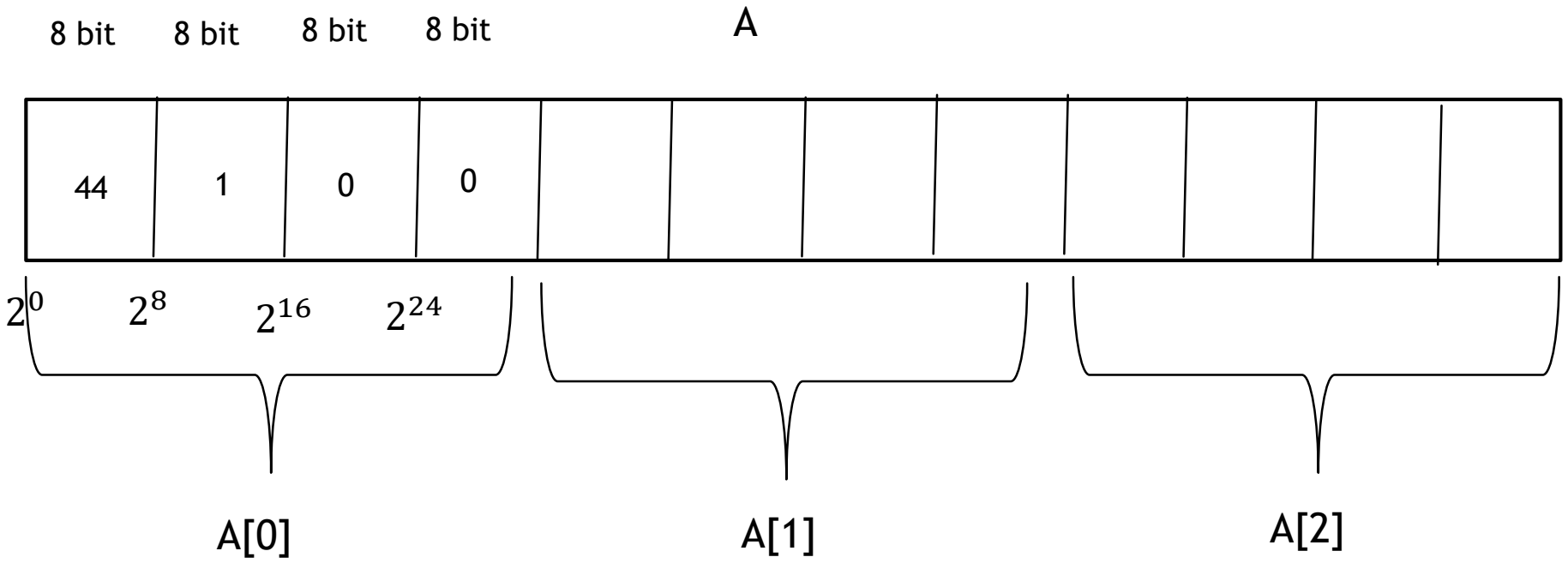
- cudaMemset(A_GPU, 300, 1); is this correct? Answer is no?

Output:
44



Remaining Issues

- How can we set first element to 300?

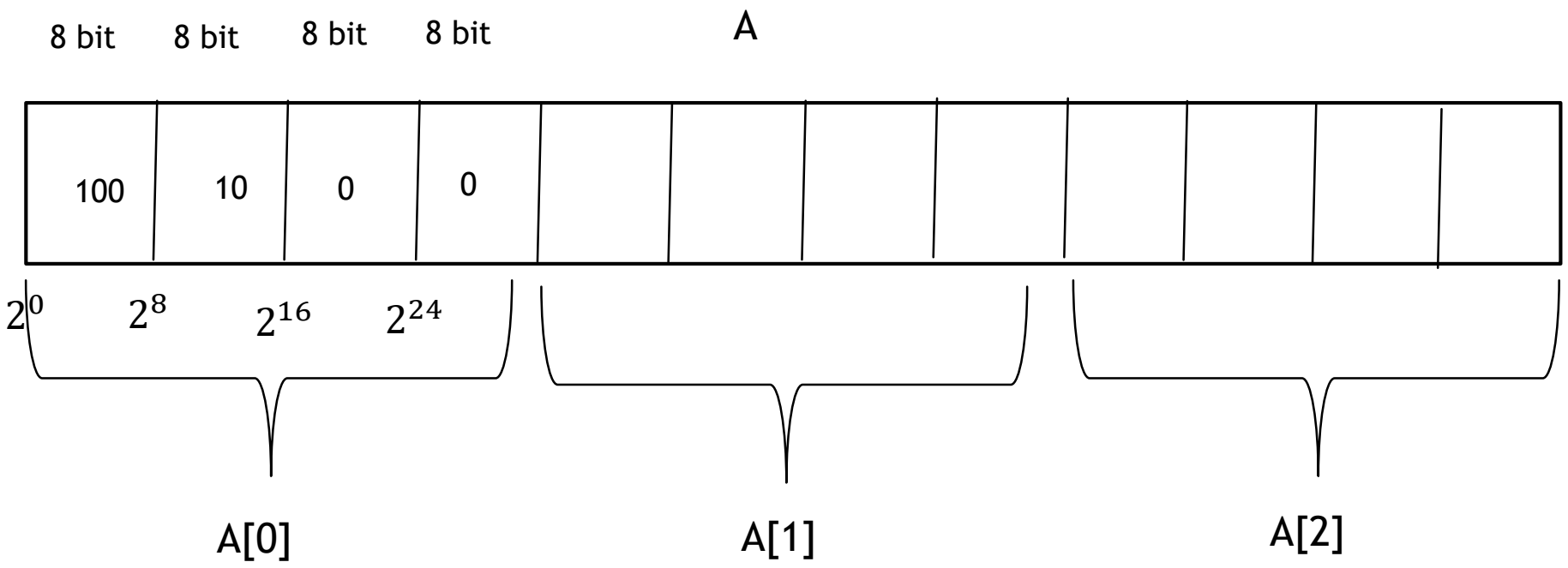



```
cudaMemset(A_GPU, 0, 12);
cudaMemset((int*) ((char*)&A_GPU[0] + 0), 44, 1);
cudaMemset((int*) ((char*)&A_GPU[0] + 1), 1, 1);
```

$\longrightarrow$ $2^0 \times 44 + 2^8 \times 1 = 300$

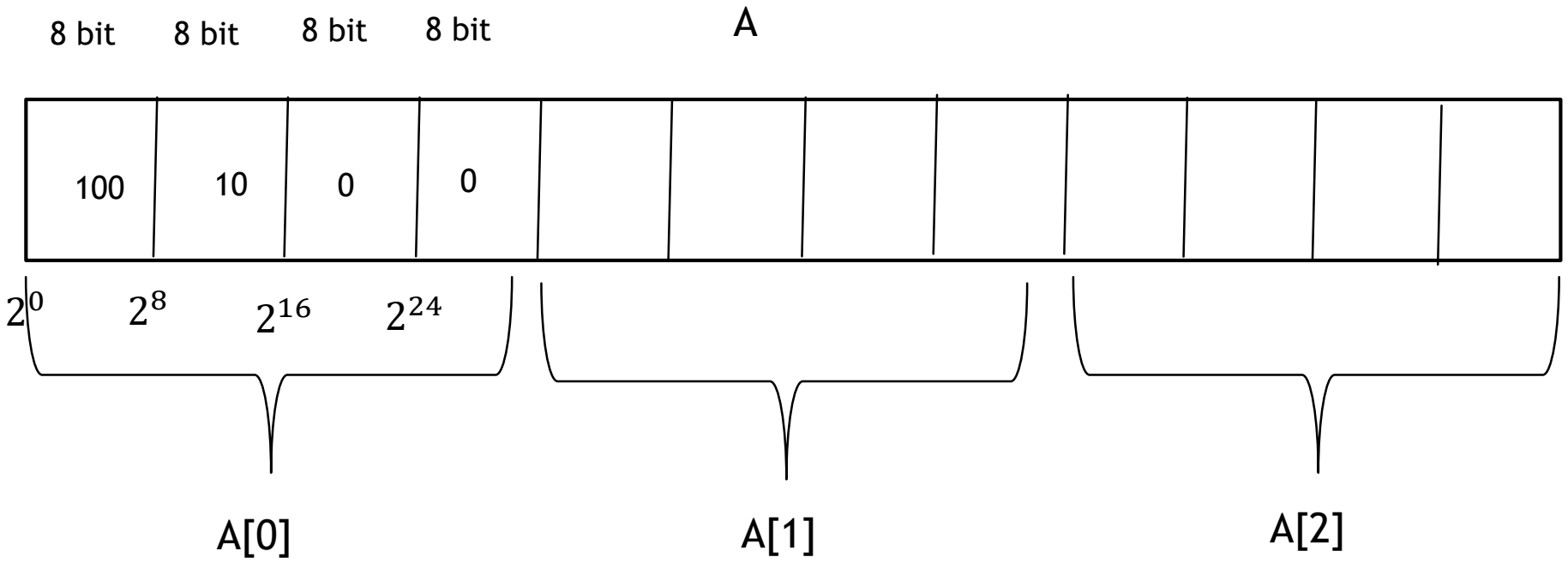
Remaining Issues

- What is the value of A[0]?



Remaining Issues

- What is the value of A[0]?




```
cudaMemset(A_GPU, 0, 12);
cudaMemset((int*) ((char*)&A_GPU[0] + 0), 100, 1)
cudaMemset((int*) ((char*)&A_GPU[0] + 1), 10, 1);
```

$2^0 \times 100 + 2^8 \times 10 = 2660$

Remaining Issues

- Setting up initial state takes too much time!

```
__global__ void curand_init(float *A,float *B,float *C,unsigned int N,unsigned int clck)
{
    int tid = blockDim.x*blockIdx.x+threadIdx.x;//Global thread id
    curandState_t state;// State of the generator
    curand_init(clck,tid,0,&state);//Initialize state

    int index = curand(&state)/N;// Generate random number
    C[tid] = A[index] + B[index];//Vector addition
}

int main(int argc, char **argv)
{
    int data_size = 4194304;//Data size
    float *A_host,*B_host,*C_host;//Host Arrays
    float *A_GPU,*B_GPU,*C_GPU;//Device Arrays

    dim3 threadsPerBlock(1024);//Number of threads in a block
    dim3 numBlocks(size/1024);//Number of blocks in a grid

    unsigned int clck = clock();

    cudaEventRecord(start);//Start timer
    for(int i=0;i<10;i++)// Call kernel ten times
    {
        curand_init<<<numBlocks,threadsPerBlock>>>(A_GPU,B_GPU,C_GPU,(unsigned long int)pow(2,32)/data_size,clck);
        cudaMemcpy(C_host,C_GPU,sizeof(float)*data_size,cudaMemcpyDeviceToHost);
    }
    cudaEventRecord(stop);//Stop timer
}
```

curand_init is called ten times for each thread

execution time is 285.431 seconds. Very huge !

curand_init is called at each running of the kernel. It is not efficient !

Remaining Issues

```
__global__ void curand_init(float *A,float *B,float *C,unsigned int N,unsigned int clck,curandInitializer RNGs)
{
    int tid = blockDim.x*blockIdx.x+threadIdx.x;//Global thread id
    curandState_t state;// State of the generator
    RNGs.load(state,tid);// Loading the state

    int index = curand(&state)/N;// Generate random number
    C[tid] = A[index] + B[index];//Vector addition

    RNGs.store(state,tid);// Storing the state
}

// Initialize function for random number generators
__global__ void initialize_RNGs(curandInitializer initRNG, unsigned int clck)
{
    unsigned int i = blockDim.x*blockIdx.x+threadIdx.x;
    curandState_t state;

    initRNG.initialize(state,i,clck);
    initRNG.store(state,i);
}

int main(int argc, char **argv)
{
    int data_size = 4194304;//Data size
    float *A_host,*B_host,*C_host;//Host Arrays
    float *A_GPU,*B_GPU,*C_GPU;//Device Arrays

    dim3 threadsPerBlock(1024);//Number of threads in a block
    dim3 numBlocks(size/1024);//Number of blocks in a grid

    unsigned int clck = clock();
    curandInitializer RNGs(data_size);//Creating a generator for 'curand_init' kernel
    initialize_RNGs<<<numBlocks,threadsPerBlock>>>(RNGs,clck);//Initialize states only once and save the states

    cudaEventRecord(start);//Start timer
    for(int i=0;i<10;i++)// Call kernel ten times
    {
        curand_init<<<numBlocks,threadsPerBlock>>>(A_GPU,B_GPU,C_GPU,(unsigned long int)pow(2,32)/data_size,clck,RNGs);
        cudaMemcpy(C_host,C_GPU,sizeof(float)*data_size,cudaMemcpyDeviceToHost);
    }
    cudaEventRecord(stop);//Stop timer
}
```

The states are loaded from global memory as a coalesced way

the actual values of the states are stored into global memory as a coalesced way

execution time is 28.652 seconds. 10x speed up !

If the # of iterations is set to 1, the execution time of both approaches will be similar and will be around 28 seconds

curand_init is called once for each thread and the states are stored into global memory

Remaining Issues

```
// Loading the state of the random number generator
__device__ void load(curandState_t &state,const unsigned  int i) const
{
    state.d = d[i];
    state.v[0] = v0[i];
    state.v[1] = v1[i];
    state.v[2] = v2[i];
    state.v[3] = v3[i];
    state.v[4] = v4[i];
    state.boxmuller_flag = boxmuller_flag[i];
    state.boxmuller_flag_double = boxmuller_flag_double[i];
    state.boxmuller_extra = boxmuller_extra[i];
    state.boxmuller_extra_double = boxmuller_extra_double[i];
}

// Storing the state of the random number generator
__device__ void store(const curandState_t &state, const unsigned int i)
{
    d[i] = state.d;
    v0[i] = state.v[0];
    v1[i] = state.v[1];
    v2[i] = state.v[2];
    v3[i] = state.v[3];
    v4[i] = state.v[4];
    boxmuller_flag[i] = state.boxmuller_flag;
    boxmuller_flag_double[i] = state.boxmuller_flag_double;
    boxmuller_extra[i] = state.boxmuller_extra;
    boxmuller_extra_double[i] = state.boxmuller_extra_double;
}
```

We load the latest values of the parameters of the state from global memory with load function

We create memory locations in global memory for the parameters of the state and save them with store function

Remaining Issues

- L1 cache usage in Tesla K40
  - Disable in default
  - ***-arch=sm_35 -Xptxas=-dlcm=ca*** flags must be used to enable
  - 16KB in default
- Experiment:
  - Remember ***Memory Coalesced Access*** example
  - Configurations:
    - Non-coalesced access (global load at most 32 transaction)
    - Semi-coalesced access (16 segments in each group. Global load at most 16 transactions)
    - Semi-coalesced access (1 segment in each group. Global load at most 1 transaction)

```
unsigned int GN = curand(&state2)/NP_group_count;//Generate a random index between 0 and group_count-1
unsigned int index,i;

for(i=0;i<10000;i++)
{
    index = (curand(&state1)/NP_group_size) + (GN*GS);//Generate a random index within selected group
    C[tid] = A[index] + B[index];//Vector addition
}
```

Remaining Issues

- Metric:
  - l1_cache_global_hit_rate: Hit rate in L1 cache for global loads

Coalesced Type	Number of Particles								
	32768			1048576			8388608		
	Cache Off	Cache On	Hit Rate	Cache Off	Cache On	Hit Rate	Cache Off	Cache On	Hit Rate
Semi-1	11.94 ms	10.25 ms	89.25%	261.3 ms	237.3 ms	89.81%	2069 ms	1887 ms	89.57%
Semi-16	36.88 ms	36.91 ms	11.32%	861.1 ms	932.2 ms	6.58%	6839 ms	7243 ms	9.06%
None	220.5 ms	405.3 ms	6.38%	11617 ms	12918 ms	0.22%	105133 ms	124417 ms	0.027%

- Only semi-1 benefits from L1 cache:
  - The group consists of one segment, so 128 bytes are stored in L1 cache for a warp
  - Since at most 64 warps can be active in an SM, at most 8192 bytes can be stored in L1 cache
  - L1 cache is enough for any number of particles
  - 9 of 10 hits are successful, in other words 9 of 10 data retrievals are successful in L1 cache
- For the other approaches:
  - L1 cache is not enough to store all the requested data of all active warps
  - The execution times become worse because of the cache miss ratio (extra cost)

References

- https://docs.nvidia.com/cuda/thrust/index.html
- https://thrust.github.io/doc/index.html
- https://nvlabs.github.io/cub/index.html
- https://docs.nvidia.com/cuda-libraries/index.html
- https://docs.nvidia.com/cuda/curand/index.html
- https://docs.nvidia.com/cuda/profiler-users-guide