

NAG Numerical Functions for GPUs Function Document

nag_gpu_mrg32k3a_uniform

1 Purpose

nag_gpu_mrg32k3a_uniform generates n values x_i from a uniform distribution over the half closed interval $(0, 1]$.

The initialization function **nag_gpu_mrg32k3a_init** must be called prior to the first call to **nag_gpu_mrg32k3a_uniform**.

2 Specification

```
#include <nag_gpu.h>
```

```
void nag_gpu_mrg32k3a_uniform (int nb, int nt, int np, float *d_P)
```

```
void nag_gpu_mrg32k3a_uniform (int nb, int nt, int np, double *d_P)
```

3 References

None.

4 Arguments

1: **nb** – int *Input*

On entry: the number of GPU blocks.

Constraint: $1 \leq \text{nb}$.

2: **nt** – int *Input*

On entry: the number of GPU threads per block. When $\text{nt} = 1$, the ordering of the output values x_i is identical to the standard sequential storage, and a default value of nt is chosen internally for efficient execution. For other choices of nt the ordering is chosen for efficient use of bandwidth and storage and it is recommended that nt is set to a multiple of 64.

Constraint: $1 \leq \text{nt}$.

3: **np** – int *Input*

On entry: the number of random values to be generated by each thread.

Constraint: $1 \leq \text{np}$.

4: **d_p**[**np** × **nb** × **nt**] – float * *Output*

5: **d_p**[**np** × **nb** × **nt**] – double * *Output*

This overloaded parameter can take type double or float.

The array must be allocated in the *device memory space*.

When $\text{nt} = 1$, the ordering of the output values, $x_{n,\tau,\beta}$, is identical to the standard sequential storage, $n + \text{np} \times \tau + \text{nt} \times \text{np} \times \beta$, where the indices, $n = 0, 1, \dots, \text{np} - 1$, $\tau = 0, 1, \dots, \text{nt} - 1$ and $\beta = 0, 1, \dots, \text{nb} - 1$ denote the element, thread and block respectively.

Otherwise $x_{n,\tau,\beta}$, the n th element generated by thread τ in block β , is stored at location $\tau + \text{nt} \times n + \text{nt} \times \text{np} \times \beta$.

On exit: the $\text{np} \times \text{nb} \times \text{nt}$ pseudorandom numbers from a uniform distribution over the half closed interval $(0, 1]$.

5 Error Indicators and Warnings

No argument constraint checking is carried out by this function. You can insert a call to function `nag_gpu_utilCheckMsg` (contained in the header file `nag_gpu.h`) following the call to `nag_gpu_mrg32k3a_uniform` to check for the last CUDA error message, if any, generated during execution.

6 Example

This example prints 240 pseudorandom numbers from a uniform distribution on $(0, 1]$ generated by `nag_gpu_mrg32k3a_uniform` after initialization by `nag_gpu_mrg32k3a_init`.

6.1 Program Text

```

/* nag_gpu_mrg32k3a_uniform
 * single-precision example program to compute random numbers
 *
 * Copyright 2009, Numerical Algorithms Group Ltd, Oxford, UK.
 *
 * Version 0.3, 2009.
 *
 */

/////////////////////////////////////////////////////////////////
// pre-processor includes
/////////////////////////////////////////////////////////////////
#include <stdio.h>

/////////////////////////////////////////////////////////////////
// include libraries
/////////////////////////////////////////////////////////////////
#include <nag_gpu.h>

/////////////////////////////////////////////////////////////////
// define precision
/////////////////////////////////////////////////////////////////
#define PRS float

/////////////////////////////////////////////////////////////////
// Main program
/////////////////////////////////////////////////////////////////

int main(int argc, char **argv)
{
    FILE *fpout=stdout;

    // Generate batchSize numbers at a time on the GPU. For large sample
    // sizes this can avoid out-of-memory errors
    const int batchSize = 2621440;

    // Total number of batches to generate
    const int nBatches = 10;

    PRS
        // Host (PC) storage for batch-generated GPU random numbers
        *h_GpuBatchNums = 0,
        // Device (GPU) storage for batch-generated random numbers
        *d_GpuBatchNums = 0;

    // Number of columns to print
    const int printCols = 6;
    // Number of random numbers to print
    int printRows = 30;
    // Make sure we have enough numbers

```

```

printRows =
    (printRows*printCols > batchSize*nBatches
     ? batchSize*nBatches/printCols + 1
     : printRows);
// Storage for numbers we'll print
PRS *h_print = 0;
int
    // Current offset into h_print
    printIdx = 0,
    // Final print counter
    idx = 0,
    r = 0,
    c = 0;

unsigned int
    // Seed vector 1
    v1[] = {1, 2, 3},
    // Seed vector 2
    v2[] = {1, 2, 3};
unsigned long long
    // Offset into random number sequence
    offset = 0;

// Greeting
fprintf(fpout, "\n\n\tNAG GPU Mrg32k3a Example Program: "
        "\n\tuniform numbers, %s precision\n\n",
        (sizeof(PRS)==sizeof(float) ? "SINGLE" : "DOUBLE"));

// Set device with highest Gflops/s
cudaSetDevice( nag_gpu_GetMaxGflopsDeviceId() );

// Allocate memory
if (!(h_GpuBatchNums = (PRS*)malloc(sizeof(PRS)*batchSize)) ) {
    fprintf(stderr, "h_GpuBatchNums allocation failure\n");
    goto END;
}
if (!(cudaSuccess == cudaMalloc((void **)&d_GpuBatchNums,
    sizeof(PRS)*batchSize))) {
    fprintf(stderr, "d_GpuBatchNums allocation failure\n");
    goto END;
}
if (!(h_print = (PRS*)malloc(sizeof(PRS)*printCols*printRows))) {
    fprintf(stderr, "h_print allocation failure\n");
    goto END;
}

for(int batchCount = 0; batchCount < nBatches; batchCount++) {
    // Due to the way mrg works, we have to initialise each time
    // in order to update the offset.
    nag_gpu_mrg32k3a_init(v1, v2, offset);
    // Launch GPU computation asynchronously. Feel free to experiment
    // with nb=#thread blocks, nt=#threads/block & np=#points/thread
    const int np = 1024;
    const int nt = 64;
    const int nb = batchSize/(np*nt);
    const int size = nb*nt*np;
    nag_gpu_mrg32k3a_uniform(nb, nt, np, d_GpuBatchNums);
    // Update the offset
    offset += size;

/*
 * One can now launch other kernels to operate on the random
 * numbers, or copy the numbers to the host and operate on
 * them there. Here we simply copy them to the host in order
 * to print them
 */
}

```

```

        cudaMemcpy(h_GpuBatchNums, d_GpuBatchNums, sizeof(PRS)*size,
                  cudaMemcpyDeviceToHost);

        // Copy GPU values to print storage
        for (int i = 0; i < size; i++) {
            if (printIdx < printRows*printCols)
                h_print[printIdx++] = h_GpuBatchNums[i];
            else break;
        } // END print storage loop

    } // END batching for loop

    // Print random numbers
    fprintf(fpout, "\nThe first %d GPU random numbers: "
            "read from left to right\n", printIdx);
    for(r = 0; r < printRows; r++) {
        for(c = 0; c < printCols; c++) {
            if (idx < printIdx) fprintf(fpout, "%.6f    ", h_print[idx++]);
            else goto END;
        }
        fprintf(fpout, "\n");
    }

END:
    fprintf(fpout, "\n\n");

    // Release GPU and CPU memory
    if (d_GpuBatchNums)
        nag_gpu_utilSafeCall( cudaFree(d_GpuBatchNums) );

    if (h_GpuBatchNums)
        free(h_GpuBatchNums);

    if (h_print)
        free(h_print);

    fflush(stdout);
    fflush(stderr);

    if (fpout!=stdout)
        fclose(fpout);

    cudaThreadExit();
    return 0;
}

```

6.2 Program Data

None.

6.3 Program Results

NAG GPU Mrg32k3a Example Program:
uniform numbers, SINGLE precision

```

The first 180 GPU random numbers: read from left to right
0.000415    0.043554    0.676632    0.953780    0.538364    0.122517
0.233872    0.330953    0.225701    0.664424    0.775400    0.970859
0.792876    0.984592    0.373765    0.439304    0.418242    0.575960
0.966857    0.389915    0.571876    0.384966    0.031386    0.909178
0.162638    0.306882    0.120568    0.249783    0.037150    0.280915
0.246188    0.620578    0.979356    0.551458    0.017794    0.452655
0.729985    0.878693    0.715318    0.833430    0.186105    0.394484
0.848503    0.669989    0.073921    0.130401    0.770848    0.520256
0.642063    0.744381    0.993680    0.701829    0.675583    0.385901
0.024892    0.727786    0.555035    0.523705    0.971990    0.812888

```

0.685879	0.674934	0.623475	0.359713	0.927781	0.566790
0.438031	0.322223	0.877097	0.963485	0.550975	0.387357
0.790583	0.137520	0.353310	0.024610	0.176005	0.931508
0.883461	0.582002	0.411362	0.976011	0.588420	0.772725
0.887605	0.461451	0.057652	0.127207	0.965246	0.362521
0.481971	0.581671	0.445595	0.857446	0.582982	0.881819
0.160718	0.809630	0.919738	0.933184	0.443287	0.198569
0.563220	0.265901	0.367080	0.290648	0.491975	0.337688
0.468207	0.246256	0.998063	0.287744	0.808686	0.176092
0.798688	0.711342	0.371650	0.193506	0.376407	0.092000
0.899071	0.216387	0.553035	0.488227	0.863395	0.777854
0.829833	0.829498	0.540458	0.704450	0.044545	0.248650
0.849092	0.522692	0.061332	0.976543	0.564674	0.420892
0.982199	0.311279	0.188874	0.350320	0.307417	0.598216
0.407396	0.726512	0.935612	0.279929	0.161260	0.326840
0.964348	0.233104	0.831688	0.700945	0.942495	0.282367
0.593335	0.072147	0.451031	0.817683	0.665705	0.502804
0.996688	0.765628	0.033733	0.123260	0.211957	0.741097
0.792567	0.389487	0.377073	0.643856	0.660084	0.995133
0.342968	0.198364	0.500442	0.044740	0.514454	0.055501
