CUDA implementation of MLMC on NVIDIA GPUs

Mike Giles

Mathematical Institute, University of Oxford

MCM 2025

July 28, 2025

Outline

- motivation
- MLMC algorithm
- key considerations
- implementation
- performance results
- current work

Motivation

NVIDIA GPUs have become dominant in HPC because of their performance, particularly for AI/ML

- top-of-the-line B200 GPU has 18,432 CUDA cores, capable of 80 TFlops (single-precision)
- my little 70W desktop RTX 4000 SFF Ada GPU has 6,144 cores, capable of 19 TFlops (single-precision)
- in general, achieving good parallel performance on GPUs is no harder than good parallel/vector performance on CPUs

Performance

Intel Xeon Gold 5418Y

- 24 cores with 2 AVX vector units and 80KB L1 cache per core
- \$1500, 185W
- MT19337 uniform RNG, with inverse CDF conversion to Normals
- 2 CPUs generate 1.9×10^{10} Normals/s

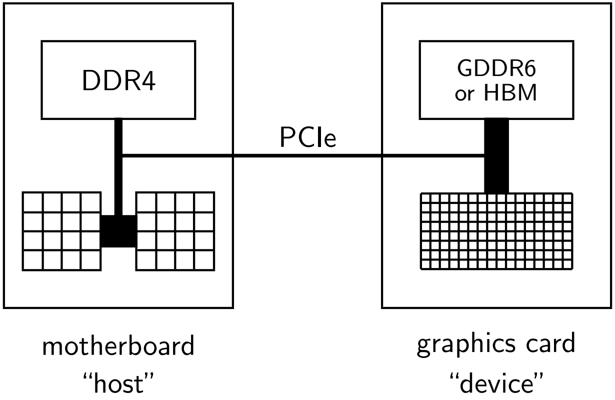
NVIDIA RTX 4000 SFF Ada GPU

- 6144 cores, 20GB memory
- \$1250, 70W
- XORWOW uniform RNG, with inverse CDF conversion to Normals
- 1 GPU generates 2.7×10^{11} Normals/s

Code available at: `people.maths.ox.ac.uk/gilesm/codes/RNG_test/`

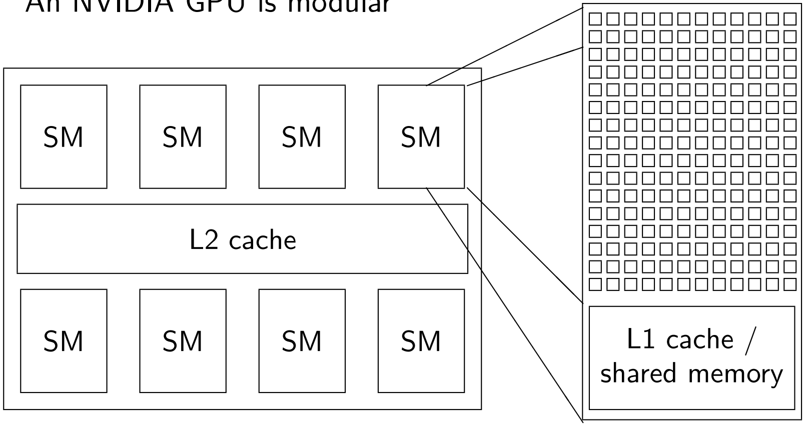
Hardware view

At the top-level, a PCIe graphics card with a many-core GPU and high-speed graphics “device” memory sits inside a standard PC/server with one or two multicore CPUs:



Hardware view

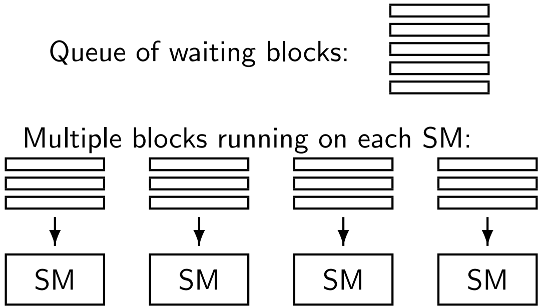
An NVIDIA GPU is modular



SM = Streaming Multiprocessor – many more than can be shown here!

Software view

- Host code:
  - runs on CPU, typically single-threaded
  - transfers data to/from GPU memory,
  - launches multiple copies of CUDA kernel code on GPU
- Kernel code:
  - each copy runs within one SM, independent of all other copies
  - typically, each has 128-512 threads, in groups of 32 (a "warp")



MLMC algorithm

start with $L=2$, and initial number of samples N_ℓ on levels $\ell = 0, 1, 2$

```
while extra samples need to be evaluated do
   evaluate extra samples on each level
   compute/update estimates for $V_\ell,\ C_\ell,\ \ell = 0, \ldots, L$
   define optimal $N_\ell,\ \ell = 0, \ldots, L$
   if no new samples needed then
      test for weak convergence
      if not converged then
         if $L = L_{max}$ then
            print warning message – failed to converge
         else
            set $L := L+1$, and initialise target $N_L$
         end if
      end if
   end if
end while
```

Software design

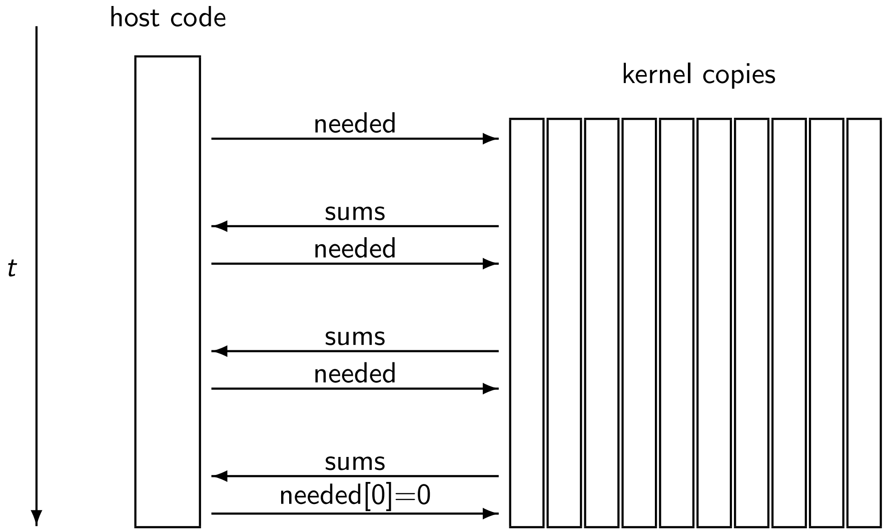
Key considerations:

- to maximise parallelism, compute additional paths for all levels at same time
- generate random numbers on-the-fly within each thread, but they must use different random number sub-streams
- (instead of GPU idling while waiting for new instructions from CPU, let it keep calculating more samples – future work)

Software design

- host launches the maximum number of kernel copies which can run without queueing
- each thread initialises random number generator using skip-ahead feature to ensure independent random number sequences
- host sets/updates number of samples needed on each level
- each warp operates independently computing additional samples as needed (continuing even when need is 100% satisfied – future work)
- kernels update sample sums on host ($\sum \Delta P_\ell$, $\sum \Delta P_\ell^2$,$\sum$ cost, etc.) when needs satisfied
- host tells kernels when to stop

Software design



Software design

Host/kernel handshaking:

- `needed` (in GPU memory)
  - host sets/updates the array of required samples N_ℓ
  - first kernel to reach N_ℓ sends sums to host, and negates N_ℓ (so others know not to do anything)
- `sums` (in host memory)
  - host initialises the array elements to NaN
  - waits for them to be set by kernels
  - resets to NaN before updating N_ℓ

Kernel coordination:

- `started`: number of samples on each level which have been started
- `device_sums`: local array of sums updated by kernels
- `lock`: atomic lock to coordinate updating of `device_sums`

Software design

Minor bits and pieces:

- each warp acts independently, looping until the final termination, deciding on each pass which level to work on
- at the end of each pass, the warp has to add together the partial sums from the 32 threads in the warp – doing this efficiently for multiple sums required some careful coding (I'm happy with this bit)
- atomic lock is used when updating the host (would prefer to use simple atomic adds – future work)
- all calculations are performed in single precision, except for sums in double precision to avoid accumulation of rounding errors

Software design

Less common CUDA features:

- `cudaOccupancyMaxActiveBlocksPerMultiprocessor` function used to determine maximum number of kernel copies which can run simultaneously in one SM, and hence the whole GPU
- two CUDA streams, one for computation and one for data transfer
- pinned host memory required for both `needed` and `sums`:
  - `needed` data transferred by asynchronous `cudaMemcpy`
  - `sums` in host memory directly updated by CUDA kernel
- atomic locks for coordination between warps

Observation: debugging massively parallel codes with asynchronous communication is tough

Results

It works! (but I'd prefer not to be using atomic locks – future work)

The current testcase is a European call option based on scalar geometric Brownian motion.

In practice, it runs so fast that I think the timing is limited by the main C/C++ code printing out the results to a text file.

The CUDA software is available here:

`https://people.maths.ox.ac.uk/gilesm/mlmc/`

and includes

- mlmc.cpp – main MLMC driver routine
- mlmc_test.cpp – routine for MLMC tests
- mcqmc06.cu – top-level application code
- mcqmc06_device.cu – low-level application code with CUDA kernels
- Makefile – uses NVIDIA's nvcc compiler

Results

```
 ---- European call ----

**********************************************************
*** MLMC file version 1.0     produced by              ***
*** C++/CUDA mlmc_test on Wed Jul 23 10:00:12 2025     ***
**********************************************************

**********************************************************
*** Convergence tests, kurtosis, telescoping sum check ***
*** using N =     32 samples                           ***
**********************************************************

 l   ave(Pf-Pc)    ave(Pf)   var(Pf-Pc)    var(Pf)    kurtosis     check        cost
---------------------------------------------------------------------------------------
 0   9.9892e+00  9.9892e+00  1.8213e+02  1.8213e+02  6.0958e+00  0.0000e+00  1.0000e+00
 1   1.7378e-01  9.9936e+00  1.1858e-01  1.9078e+02  2.9612e+01  1.1556e-02  2.0000e+00
 2   1.0289e-01  1.0627e+01  4.2051e-02  2.2255e+02  2.9782e+01  3.4558e-02  4.0000e+00
 3   5.5127e-02  1.0237e+01  1.1711e-02  2.1820e+02  2.4802e+01  2.8176e-02  8.0000e+00
 4   2.9023e-02  1.0560e+01  4.0310e-03  2.4338e+02  1.4449e+01  1.8238e-02  1.6000e+01
 5   1.2859e-02  8.1586e+00  6.2777e-04  1.6427e+02  6.5303e+00  1.6005e-01  3.2000e+01
 6   7.5855e-03  9.4988e+00  2.6258e-04  2.2050e+02  6.2783e+00  9.0772e-02  6.4000e+01
 7   3.3581e-03  1.1780e+01  3.6911e-05  2.5661e+02  2.8254e+00  1.3909e-01  1.2800e+02
 8   2.8687e-03  1.3247e+01  2.0576e-05  3.2954e+02  5.0006e+00  8.0794e-02  2.5600e+02
 9   6.1800e-04  1.0237e+01  1.3021e-06  1.4798e+02  3.0654e+00  1.8721e-01  5.1200e+02
10   7.0577e-04  1.3135e+01  1.3426e-06  3.5231e+02  5.4735e+00  1.7659e-01  1.0240e+03
```

Results

```
*******************************************************
*** Linear regression estimates of MLMC parameters ***
*******************************************************

 alpha = 0.936768  (exponent for MLMC weak convergence)
 beta  = 1.939754  (exponent for MLMC variance)
 gamma = 1.000000  (exponent for MLMC cost)



*****************************
*** MLMC complexity tests ***
*****************************

 eps     value    mlmc_cost  std_cost savings     N_l
-----------------------------------------------------
0.001 1.0451e+01 3.410e+08 4.810e+11 1410.64 298508288 5834752 2215936 825344 299264 108288 38592 13888
                                                                                   4992   1792   640
0.002 1.0451e+01 8.495e+07 2.525e+10  297.30  74465280 1458176  551936 206848  74496  26752  9600  3648
                                                                                          1344   448
0.005 1.0455e+01 1.355e+07 4.499e+09  332.17  11894784  233472   88064  32768  12032   4480  1600   576
                                                                                                 224
0.010 1.0460e+01 3.387e+06 4.379e+08  129.29   2965504   59392   23552   8704   3072   1280   448   160
0.020 1.0452e+01 8.643e+05 4.704e+07   54.43    741376   16384    7168   3072   1024    384   128
```

Current work

- CUDA half-precision calculations:
  - `half2` datatype with two FP16 variables in a 32-bit register
  - uniform $\rightarrow$ approximate Normal mapping using a lookup table
  - nested MLMC to correct accuracy to single precision
- AVX-512 half-precision calculations on latest Intel Xeon CPUs:
  - `__m512h` datatype with 32 FP16 variables in a 512-bit vector register
  - uniform $\rightarrow$ approximate Normal mapping using piecewise linear approximation on dyadic intervals
  - nested MLMC to correct accuracy to single precision
- overall objective is to get fair comparison of FPGA, GPU and CPU architectures, looking at both price/performance and energy efficiency

Is anyone else interested in collaborating on CUDA code?