Option Pricing with Cuda Programming

Jian Wang

wangjian790@gmail.com

Financial math Ph.D.
Florida State University

Table of contents

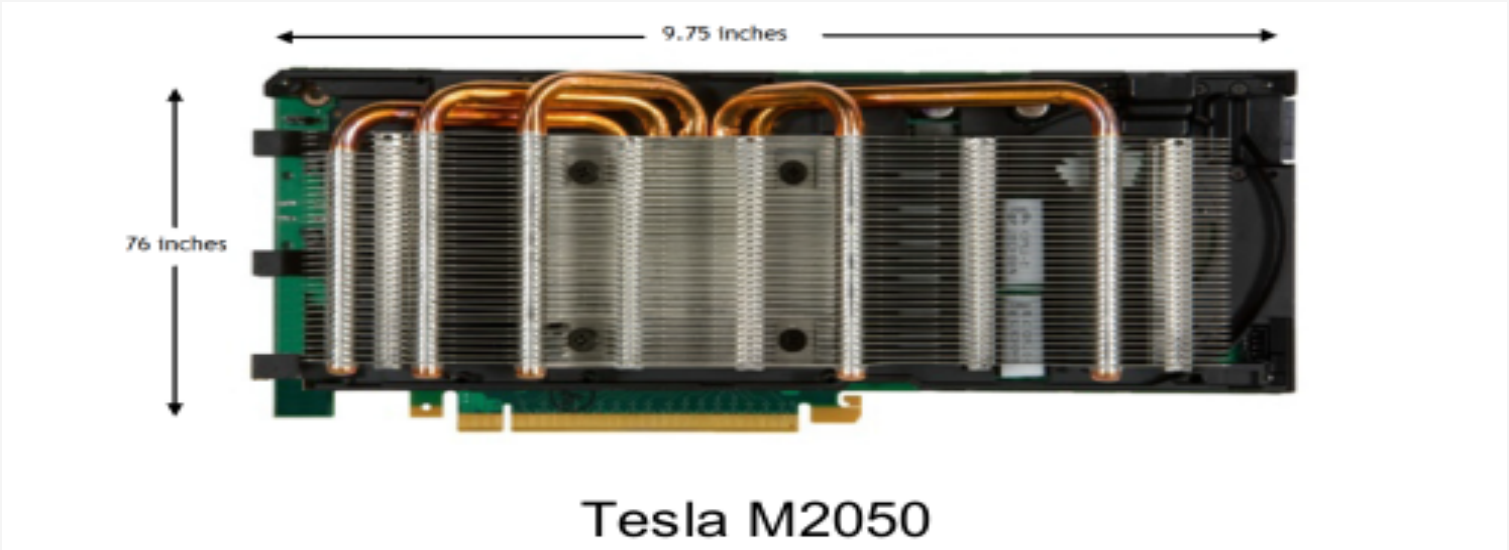
Contents

GPU-Computing

- **GPU:** Graphicsx Processing Unit
- **GPGPU**: General Purpose Graphics Processing Unit
- **GPU-accelerated computing:** is the use of a GPU together with a CPU to accelerate scientific and engineering applications
- **Remark**. GPU does NOT work by itself. It is used as a device of a CPU, and is often called “accelerator”
- **Remark**. Intel call their Xeon Phi product “coprocessor”

GPU-Computing

- **Tegra:** mobile and embedded devices (e.g., smart phones)
- **GeForce:** consumer graphics (e.g., PC Gaming)
- **Quadro:** professional visualization (e.g., CAD)
- **Tesla:** parallel computing (what we have at RCC)



Tesla M2050

Tesla => Fermi => Kepler => Maxwell => Pascal

- Tesla product family is classified using Compute Capability
- Kepler class architecture has major version number 3
- Fermi class architecture has major version number 2
- Tesla class architecture has major version number 1

GPU	COMPUTE CAPABILITY
Tesla K40	3.5
Telsa K20	3.5
Telsa K10	3.0
Telsa C2070	2.0
Telsa C1060	1.3

Telsa Family Compute Capability

Compute Capability

FERMI vs KEPLER

	FERMI (TESLA C2050)	KEPLER (TESLA k10)
CUDA Cores	448	2 x 1536
Memory	6 GB	8 GB
Peak Performance*	1.03 Tflops	4.58 Tflops
Memory Bandwidth	144 GB/s	320 GB/s

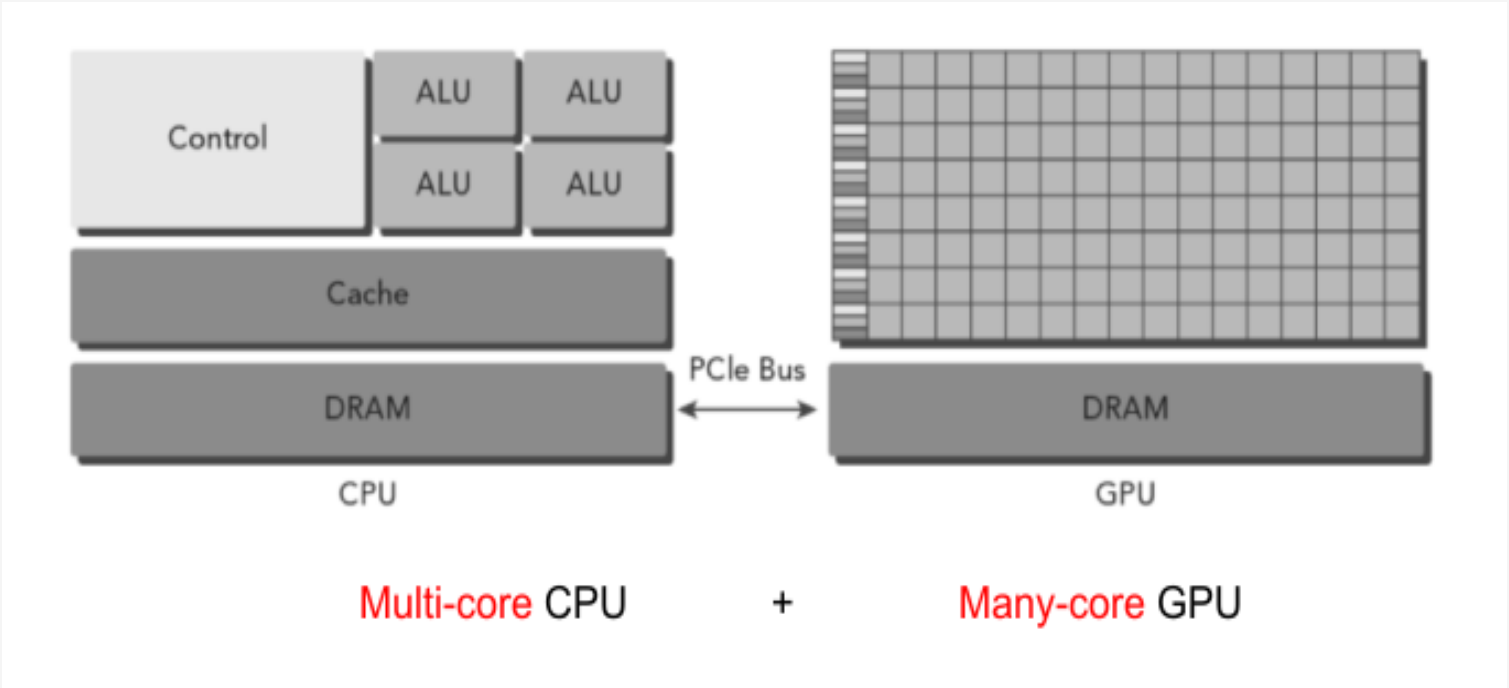
* Peak single- precision floating point performance

- Question: are these many GPU cores equivalent to same number of CPU cores?
- Answer: no. (that would cost you a million dollars)

GPU core VS CPU core

- **CPU core:** relatively heavy-weight, designed for complex control logic, optimized for sequential programs.
- **GPU core:** relatively light-weight, designed with simple control logic, optimized for data-parallel tasks, focusing on throughput of parallel programs.
- **CPU+GPU:** heterogeneous architecture

Heterogenneous Architecture



- **Remark:** GPU has its own memory, connect to CPU via PCI-express bus
- **Remark:** Differentiate Multi-Core from Many-Core (e.g., Intel Xeon Phi co-processor is also many-core).

CUDA Programming Model

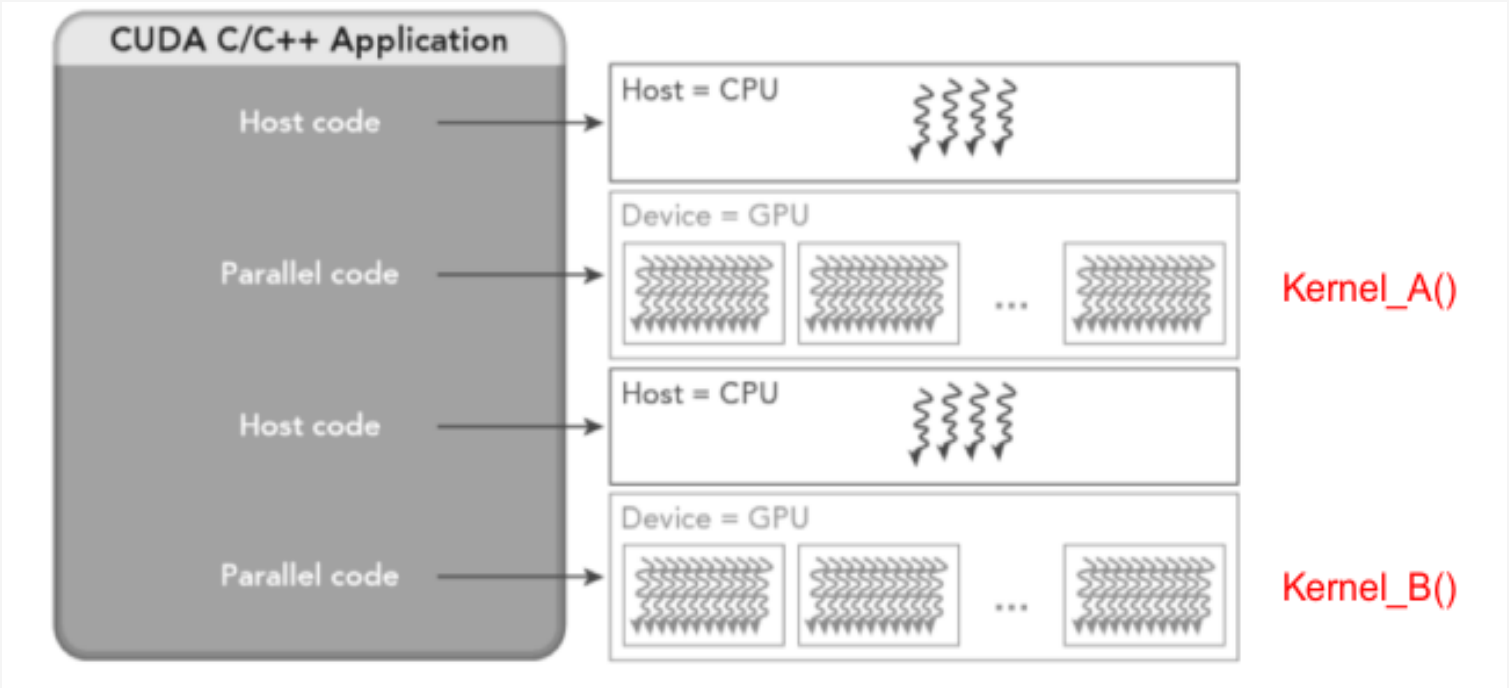
Question: how to use GPU to accelerate your code?

- A. Figure out which part of your code need speedup.
- B. Rewrite that part of your program for GPU using CUDA.
- C. Re-compile your program for GPU (nvcc -o a.out a.cu).
- D. Send that piece of Code to GPU memory (automatic).
- E Send the data needed to GPU memory (by you).
- F. Copy the resulting data back to CPU memory (by you).
- G. Continue your CPU calculation.

Contents

CUDA Programming Model

- Divide your code into Host (CPU) and Device (GPU) Code
- Processing flow of a CUDA program:
  - a. Copy data from CPU memory to GPU memory
  - b. Invoke kernel to run on the GPU.
  - c. Copy data back from GPU to CPU memory
  - d. Release the GPU memory and reset the GPU.
- CUDA code file name extension .cu
- CUDA compiler: nvcc (it compiles .c, .cpp too!)
- $ nvcc -lm -o a.out a.cu

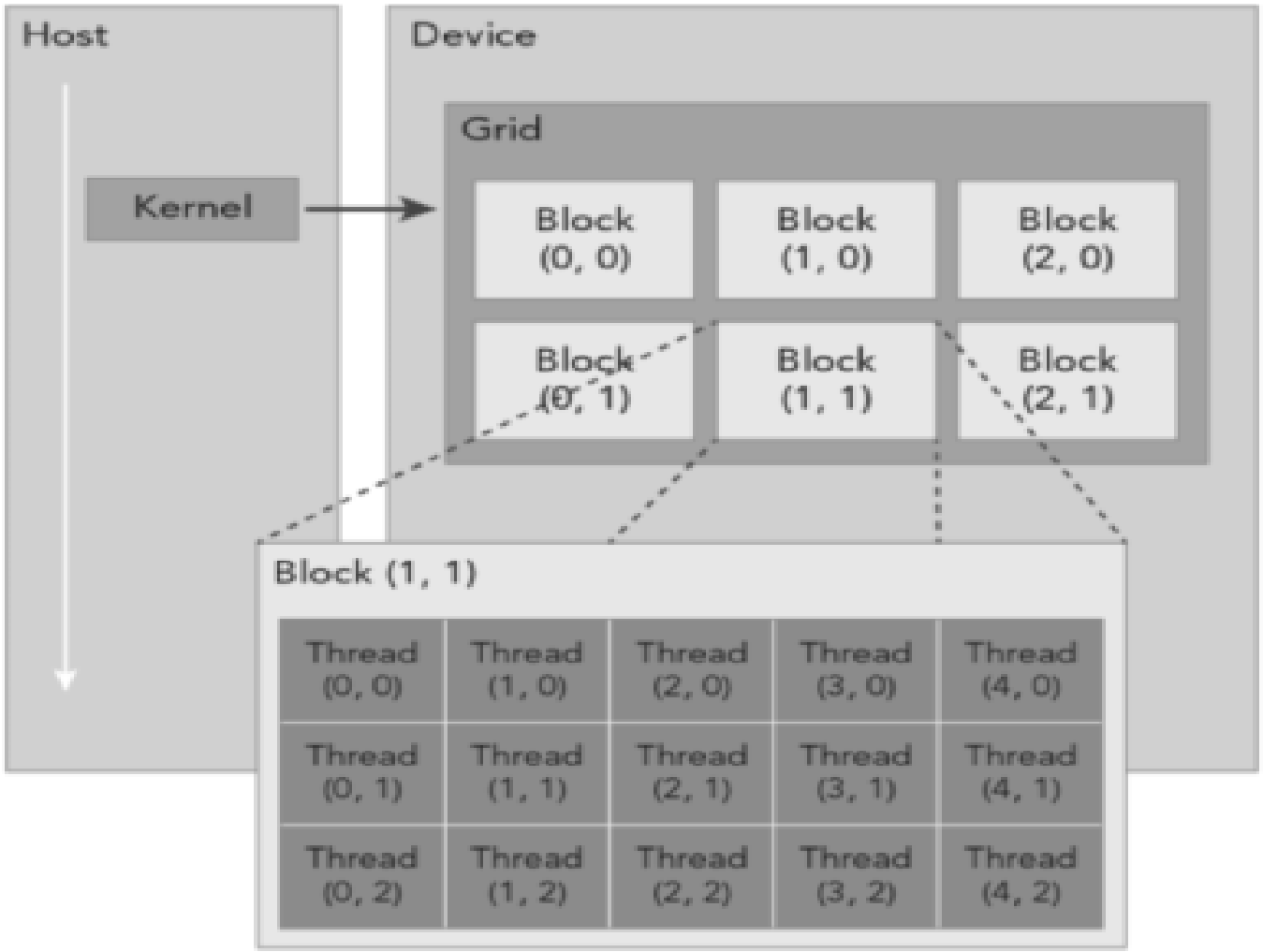


- The kernel function run concurrently by many threads on the GPU.
- CPU might or might not wait for GPU depending on synchronization. (Q: when GPU is busy, what is CPU doing?)
- You can have more than one kernel functions in your CUDA APP

2-Level Thread Hierarchy (p1)

- There are many many threads, so they need be managed.
- Grid: All threads spawned by a single kernel.
- Grid is made up of many thread blocks.
- A thread block is a group of threads which can cooperate
  Intra-block synchronization
  Sharing memory within a block
  NO memory sharing or synchronization across blocks
- A thread finds its own unique id using two coordinates: blockIdx and threadIdx, for example (1D case):
  id = threadIdx.x + blockIdx.x*blockDim.x
  Summary: Level 1 is a grid of blocks; level 2 is block of threads

2-Level Thread Hierarchy



Example: 2D grid + 2D block

- Declaration Syntax:
  ```
  __global__ void kernel(arg1, arg2, ...){
        function body;
  }
  ```
- __global__ is a function type qualifier.
- Kernel function is invoked by CPU, but run on GPU with many copies (one thread per copy).
- Kernel invoking Syntax:
  kernel«<grid, block»>(arg1, arg2, ...)
  both grid, block are of type dim3, e.g.,
  dim3 gridDim(256,256,1); dim3 blockDim(16,16,1);

CUDA Function Type Qualifiers

Declaration Syntax:

```
__global__ void name1(arg1, arg2, ...){
    name2(arg1,arg2); // invoke device function
}
__device__ double name2(arg1, arg2, ...){
    function body;
}
__host__ float name3(arg1, arg2, ...){
    function body;
}
```

Host and device routines only run on CPU, and GPU respectively. Global declares kernel function, run on GPU, which can call device functions.

CUDA Kernel Function

Question: What should be put in the Kernel function?

```
for (i = 0; i < 1000; i++)
    C[i] = A[i] + B[i];
}
__global__ void kernel(int* A, int* B, int* C) {
    id = threadIdx.x + blockIdx.x*blockDim.x;
    C[id] = A[id] +B[id];
}
```

In essence, your for loop with for peeled off, but keep the things inside.

The key part is to map your data to threads (array indices).

- Question: How to move data between CPU and GPU?
  cudaMalloc((void**) A_d, size_t n_bytes);
  cudaMemcpy(ptr_dest, ptr_src, n_bytes, direction);
- Where
  ptr_dest, ptr_src are destination/source pointers;
  direction can be
    cudaMemcpyHostToDevice
    cudaMemcpyDeviceToHost
    cudaMemcpyDeviceToDevice
- How to free Cuda Memory?
  cudaFree(A_d);

CUDA Memory Operations

```
int     nElem  = 1024;
size_t nbytes = nElem*sizeof(float);
float *A_h, *B_h, *C_h;
float *A_d, *B_d, *C_d;
int    i;

A_h  = (float*) malloc(nbytes);
B_h  = (float*) malloc(nbytes);
C_h  = (float*) malloc(nbytes);
init_data(A_h, nElem);
init_data(B_h, nElem);

cudaMalloc( (void **) &A_d, nbytes);
cudaMalloc( (void **) &B_d, nbytes);
cudaMalloc( (void **) &C_d, nbytes);

cudaMemcpy(A_d, A_h, nbytes, cudaMemcpyHostToDevice);
cudaMemcpy(B_d, B_h, nbytes, cudaMemcpyHostToDevice);

sum_1D<<<2, 512>>>(A_d, B_d, C_d, nElem);
cudaMemcpy(C_h, C_d, nbytes, cudaMemcpyDeviceToHost);
cudaFree(A_d);
cudaFree(B_d);
cudaFree(C_d);
```

Example: Sum two 1D arrays assuming 1D block and 1D grid: Here is the kernel routine sum_1D:

```
__global__ void sum_1D(float* A_d, float* B_d, float* C_d,
                       int size) {

   int tx = threadIdx.x;
   int bx = blockIdx.x;
   int id = tx + bx*blockDim.x;
   if (id < size) {
      C_d[id] = A_d[id] + B_d[id];
   }
   return;
}
```

The main function was already shown in the previous page

How to Compile CUDA Code?

Cuda nvcc compiler (cuda-7.0 at the RCC):

a. Pure C code: a.c
   $ nvcc -o a.out a.c

b. Single Cuda code: a.cu
   $ nvcc -arch sm_20 -O3 -o a.out a.cu

c.C and Cuda Mixed: a.cu and b.c
   $ gcc -o b.o -c b.c (or icc if you use intel compiler)
   $ nvcc -o a.out b.o a.cu

Use GPU ON the HPC Cluster

Step 1: Load the cuda module

$ module load cuda

Step 2: Compile your cuda code

$ nvcc -o a.out a.cu

Step 3: Create a SLURM job script

$ vi slurm.sub

Step 4: Submit your job.

$sbatch slurm.sub

```
#!/bin/bash
#SBATCH -n 1
#SBATCH -J "sum1d"
#SBATCH -p gpu_q
#SBATCH -t 48:00:00
#SBATCH --mail-type all

cat $SLURM_JOB_NODELIST
pwd
srun  -n1 ./sum1d

~
~
```

Example of job submission script

Use GPU ON the HPC Cluster

Now let's submit something to the GPU node of the cluster.

Sadly, we have only 1 node (16 CPUs) armed with 2 GPUs

Your job has to be submitted to the queue called gpu_q

SBATCH -p gpu_q

The clock limit is 2 days.

SBATCH -t 48:00:00

Goal: 8 compute nodes with 1 GPU card on each node.

Example: Black-Scholes Option Pricing

Black-Scholes formula for European options:

$$
\begin{aligned}
V_{call} &= S\Phi(d_1) - Xe^{-rT}\Phi(d_2) \\
V_{put} &= Xe^{-rT}\Phi(-d_2) - S\Phi(d_1) \\
d_1 &= \frac{log(s/x) + (r + \sigma^2/2)T}{\sigma T} \\
d_2 &= d_1 - \sigma\sqrt{(T)}
\end{aligned}
$$

Note: $\Phi(x)$ is the cumulative standard normal distribution

Contents

Example: Black-Scholes Option Pricing

$\Phi(x)$ can be evaluated via numerical approximation

$$
\begin{aligned}
K &= \frac{1}{1 + 0.2316419|d|} \\
\Phi(d) &= \frac{1}{\sqrt{2\pi}} e^{-d^2/2}[a_1 K + a_2 K^2 + a_3 K^3 + a_4 K^4 + a_5 K^5] \\
a_1 &= 0.31938153 \\
a_2 &= -0.356563782 \\
a_3 &= 1.781477937 \\
a_4 &= -1.821255978 \\
a_5 &= 1.330274429
\end{aligned}
$$

Black-Scholes Option Pricing

The option price $V(S_0, X, T, r, \sigma)$ depends on 5 parameters
Set up of the numerical problem.
Input parameters:

- So[4000000] = rand(5, 60)
- X[4000000] = rand(1, 100)
- T[4000000] = rand(1/12, 5)
- r = 0.03
- σ = 0.3

Output:

- C[4000000]
- P[4000000]

Input data So[],X[], T[] generated on CPU, copied to GPU.
Output data C[],P[] calculated by GPU, copied back to CPU.

Black-Scholes Option Pricing

- Output of computing the 4 million options prices:
- Summary: GPU speed up by 267 times is achieved!

```
c557-002.stampede(12)$ make blackscholes
nvcc -arch=compute_35 -code=sm_35 -O3 -o blackscholes blackscholes.cu
c557-002.stampede(13)$ ./blackscholes
serial CPU code takes  0.834643 seconds
Parallel GPU code takes  0.003126 seconds
GPU speed up =     267.0084 times.
max_err_Call =   0.000000000000444, max_err_Put =   0.000000000000533
The first 100 results from CPU, and GPU are ...
i=  0, C_c =   1.360605 C_g =   1.360605 P_c =  13.511108 P_g =  13.511108
i=  1, C_c =   2.367699 C_g =   2.367699 P_c =  37.758286 P_g =  37.758286
i=  2, C_c =   0.000000 C_g =   0.000000 P_c =  26.572678 P_g =  26.572678
i=  3, C_c =  36.304579 C_g =  36.304579 P_c =   0.000003 P_g =   0.000003
i=  4, C_c =   2.168827 C_g =   2.168827 P_c =  15.454739 P_g =  15.454739
```

GPU VS CPU for Black-Scholes Model

Monte-Carlo Option Pricing

Assume stock price $S(t)$ follows a geometrical Brownian motion.

$$dS_t = \mu S_t dt + \sigma S_t dW_t$$

$$dW_t \sim N(0, dt)$$

$$S(t) = S_0 e^{\mu t + \sigma \sqrt{t} N(0,1)}$$

$$\mu = r - 0.5\sigma^2$$

$$C(T) = \max(S(T) - X, 0)$$

- μ is the constant drift.
- $W(t)$: Wiener process
- σ: the volatility
- r: risk free rate
- X: strike price
- $C(T)$: the option price

Goal: Estimate $C(T)$ by Monte-Carlo simulation of $S(T)$

Monte-Carlo Option Pricing

What I am going to do is

- Compute option price using the analytical BK formula.
- Estimate option price via Monte-Carlo on CPU (serial).
- Estimate option price via Monte-Carlo on GPU (parallel).
- Compare the performance

About the random number generators

- CPU version: gsl_ran_gaussian(). (gnu-science library)
- GPU version: curand_normal().(cuda random library)

Monte-Carlo Option Pricing

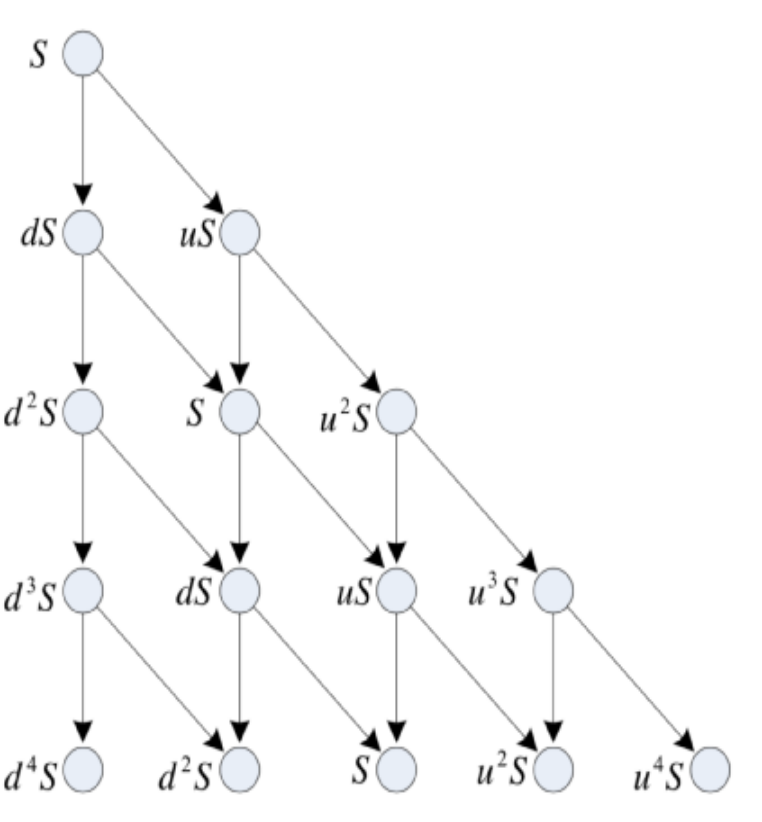
Output of computing the 8192 option prices (131072 paths):
Summary: GPU speed up by around 1,000 times!

```
c557-002.stampede(25)$ ./montecarlo
NPATHS =     131072
OptN   =       8192
First, do the analytical Black-Scholes models..
finish Black-Scholes in     0.001323 seconds.
Next, CPU version of the MonteCarlo Simulation...
finish CPU version of Monte Carlo in   140.457312 seconds.
GPU version starts...
Initialize the random seeds for the Monte Carlo simulation...
Seed Setups finished in     0.029165 seconds
GPU Monte Carlo Simulation Starts now...
GPU Monte Carlo finished with     0.091413 seconds
GPU Monte Carlo speed up is     1164.8663 times
the first 100 simulations produce the following outputs:
S0 =    42.81, X =    30.18, BK =   19.2555, C_C =   19.2271, C_G =   19.2618,
S0 =    40.93, X =    68.46, BK =    0.0022, C_C =    0.0024, C_G =    0.0018,
S0 =    20.09, X =    57.85, BK =    0.0000, C_C =    0.0000, C_G =    0.0000,
S0 =    29.93, X =    36.33, BK =    2.4804, C_C =    2.4932, C_G =    2.4632,
```

GPU VS CPU for Monte-Carlo Option Pricing

Binomial tree Option Model

Goal: Price European options using the binomial tree option model.



Binomial tree for 4 time steps

$$\mu = e^{\sigma\sqrt{dt}}$$

$$d = 1/u$$

$$P_u = \frac{e^{rd} - d}{u - d}$$

$$[P_u u + (1 - P_u)d]S_k = S_k s^{rdt}$$

P_u is the probability of pricing going up

Binomial tree Option Model

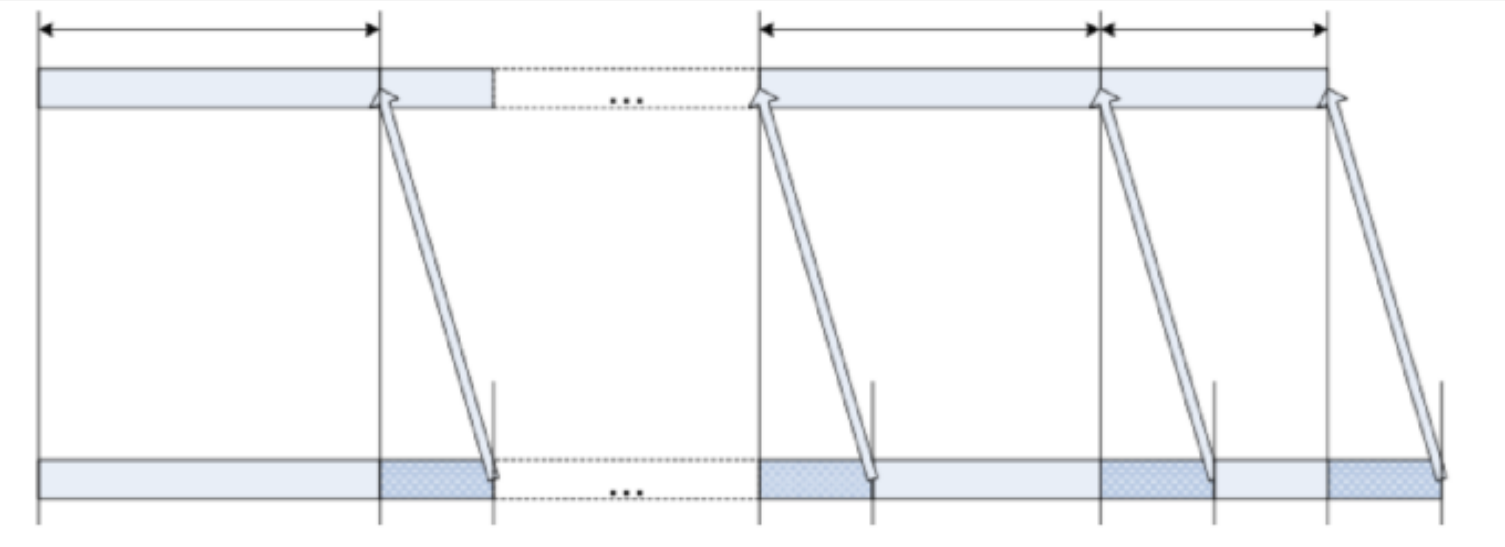
- Idea: Backward iteration. Let V(t) be price of the option at t.
- Each node on the tree has exactly two child nodes.
- The option price on a node can be evaluated using the prices of its two child nodes.

$$V_t = [P_u V_{u,t+dt} + (1 - P_u) V_{d,t+dt}] e^{-rdt}$$

- The option price on the expiry day, i.e., the leaf node can be directly evaluated, V = max(S-X, o).
- List of parameters S, X, T, N, r, .
- Amount of computation of the order O(N*N).
- In contrast, Black-Scholes work of the order O(1).

Binomial tree Option Model

- Question: How to realize the iterative algorithm on GPU?
- At each instant, GPU threads must be working on nodes of the same level (i.e., same time step).
- Suppose each thread work on some nodes of the same level, how to deal with the boundary condition?
- Each thread has its own number of iteration



Split work among threads of the block

Binomial tree Option Model

Numerical set up:

evaluate the price of 1024 options with N = 2048.

GRIDSIZE = 1024, BLOCKSIZE=128

```
c557-001.stampede(20)$ ./latticev4
Depth of the tree NUM_STEPS = 2048
grid structure: <<<1024, 128>>>
Starting the GPU code...
GPU code finished within     0.018006 seconds
CPU code finished within     2.354731 seconds
Speed up you got   130.77
Compare the GPU and CPU binary model now...
passed comparison between GPU and CPU binomial model
Compare the binary model with Black-Scholes model now...
passed comparison between binomial and Black-Scholes model
Here are outputs for the first 10 lines
   26.00,    16.38,    10.0870,    10.0870,    10.0870
   24.58,    32.14,     0.9494,     0.9494,     0.9494
   27.79,     8.70,    19.2591,    19.2591,    19.2591
```

Results: 130 times speed up on GPU for binomial tree option model

Contents

Summary

- GPU hardware architecture (why should I care?)
- CUDA/C/C++ programming model
- Black-Scholes formula (200x speedup)
- Monte-Carlo option pricing (1000x speedup)
- Binomial option model (130x speedup)