```
/* Program to compute swaption portfolio using NVIDIA CUDA */

#include <stdio.h>
#include <cutil.h>

// parameters for nVidia device execution

#define BLOCK_SIZE 64
#define GRID_SIZE 1500

// parameters for LIBOR calculation

#define NN 80
#define NMAT 40
#define L2_SIZE 3280 //NN*(NMAT+1)
#define NOPT 15
#define NPATH 96000

// constant data for swaption portfolio: stored in device memory,
// initialised by host and read by device threads

__constant__ int    N, Nmat, Nopt, maturities[NOPT];
__constant__ float  delta, swaprates[NOPT], lambda[NN];


/* Monte Carlo LIBOR path calculation */

__device__ void path_calc(float *L, float *z)
{
  int   i, n;
  float sqez, lam, con1, v, vrat;

  for(n=0; n<Nmat; n++) {
    sqez = sqrtf(delta)*z[n];
    v = 0.0;

    for (i=n+1; i<N; i++) {
      lam  = lambda[i-n-1];
      con1 = delta*lam;
      v   += __fdividef(con1*L[i],1.0+delta*L[i]);
      vrat = __expf(con1*v + lam*(sqez-0.5*con1));
      L[i] = L[i]*vrat;
    }
  }
}


/* forward path calculation storing data
   for subsequent reverse path calculation */

__device__ void path_calc_b1(float *L, float *z, float *L2)
{
  int   i, n;
  float sqez, lam, con1, v, vrat;

  for (i=0; i<N; i++) L2[i] = L[i];

  for(n=0; n<Nmat; n++) {
    sqez = sqrt(delta)*z[n];
    v = 0.0;

    for (i=n+1; i<N; i++) {
      lam  = lambda[i-n-1];
      con1 = delta*lam;
      v   += __fdividef(con1*L[i],1.0+delta*L[i]);
      vrat = __expf(con1*v + lam*(sqez-0.5*con1));
      L[i] = L[i]*vrat;

      // store these values for reverse path //
      L2[i+(n+1)*N] = L[i];
    }
  }
}


/* reverse path calculation of deltas using stored data */

__device__ void path_calc_b2(float *L_b, float *z, float *L2)
{
  int   i, n;
  float faci, v1;

  for (n=Nmat-1; n>=0; n--) {
    v1 = 0.0;
    for (i=N-1; i>n; i--) {
      v1     += lambda[i-n-1]*L2[i+(n+1)*N]*L_b[i];
      faci    = __fdividef(delta,1.0+delta*L2[i+n*N]);
      L_b[i] = L_b[i]*__fdividef(L2[i+(n+1)*N],L2[i+n*N])
               + v1*lambda[i-n-1]*faci*faci;

    }
  }
}

/* calculate the portfolio value v, and its sensitivity to L */
/* hand-coded reverse mode sensitivity */

__device__ float portfolio_b(float *L, float *L_b)
{
  int   m, n;
  float b, s, swapval,v;
  float B[NMAT], S[NMAT], B_b[NMAT], S_b[NMAT];

  b = 1.0;
  s = 0.0;
  for (m=0; m<N-Nmat; m++) {
    n    = m + Nmat;
    b    = __fdividef(b,1.0+delta*L[n]);
    s    = s + delta*b;
```

```
    B[m] = b;
    S[m] = s;
  }

  v = 0.0;

  for (m=0; m<N-Nmat; m++) {
    B_b[m] = 0;
    S_b[m] = 0;
  }

  for (n=0; n<Nopt; n++){
    m = maturities[n] - 1;
    swapval = B[m] + swaprates[n]*S[m] - 1.0;
    if (swapval<0) {
      v     += -100*swapval;
      S_b[m] += -100*swaprates[n];
      B_b[m] += -100;
    }
  }

  for (m=N-Nmat-1; m>=0; m--) {
    n = m + Nmat;
    B_b[m] += delta*S_b[m];
    L_b[n]  = -B_b[m]*B[m]*__fdividef(delta,1.0+delta*L[n]);
    if (m>0) {
      S_b[m-1] += S_b[m];
      B_b[m-1] += __fdividef(B_b[m],1.+delta*L[n]);
    }
  }

  // apply discount //

  b = 1.0;
  for (n=0; n<Nmat; n++) b = b/(1.0+delta*L[n]);

  v = b*v;

  for (n=0; n<Nmat; n++){
    L_b[n] = -v*delta/(1.0+delta*L[n]);
  }

  for (n=Nmat; n<N; n++){
    L_b[n] = b*L_b[n];
  }

  return v;
}


/* calculate the portfolio value v */

__device__ float portfolio(float *L)
{
  int   n, m, i;
  float v, b, s, swapval, B[40], S[40];

  b = 1.0;
  s = 0.0;

  for(n=Nmat; n<N; n++) {
    b = b/(1.0+delta*L[n]);
    s = s + delta*b;
    B[n-Nmat] = b;
    S[n-Nmat] = s;
  }

  v = 0.0;

  for(i=0; i<Nopt; i++){
    m = maturities[i] -1;
    swapval = B[m] + swaprates[i]*S[m] - 1.0;
    if(swapval<0)
      v += -100.0*swapval;
  }

  // apply discount //

  b = 1.0;
  for (n=0; n<Nmat; n++) b = b/(1.0+delta*L[n]);

  v = b*v;

  return v;
}


__global__ void Pathcalc_Portfolio_KernelGPU(float *d_v, float *d_Lb)
{
  const int     tid = blockDim.x * blockIdx.x + threadIdx.x;
  const int threadN = blockDim.x * gridDim.x;

  int   i,path;
  float L[NN], L2[L2_SIZE], z[NN];
  float *L_b = L;

  /* Monte Carlo LIBOR path calculation*/

  for(path = tid; path < NPATH; path += threadN){
    // initialise the data for current thread
    for (i=0; i<N; i++) {
      // for real application, z should be randomly generated
      z[i] = 0.3;
      L[i] = 0.05;
    }
    path_calc_b1(L, z, L2);
    d_v[path] = portfolio_b(L,L_b);
    path_calc_b2(L_b, z, L2);
    d_Lb[path] = L_b[NN-1];
```

```
  }
}


__global__ void Pathcalc_Portfolio_KernelGPU2(float *d_v)
{
  const int     tid = blockDim.x * blockIdx.x + threadIdx.x;
  const int threadN = blockDim.x * gridDim.x;

  int   i, path;
  float L[NN], z[NN];

  /* Monte Carlo LIBOR path calculation*/

  for(path = tid; path < NPATH; path += threadN){
    // initialise the data for current thread
    for (i=0; i<N; i++) {
      // for real application, z should be randomly generated
      z[i] = 0.3;
      L[i] = 0.05;
    }
    path_calc(L, z);
    d_v[path] = portfolio(L);
  }
}


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

int main(int argc, char **argv){

  // 'h_' prefix - CPU (host) memory space

  float  *h_v, *h_Lb, h_lambda[NN], h_delta=0.25;
  int     h_N=NN, h_Nmat=NMAT, h_Nopt=NOPT, i;
  int     h_maturities[] = {4,4,4,8,8,8,20,20,20,28,28,28,40,40,40};
  float   h_swaprates[]  = {.045,.05,.055,.045,.05,.055,.045,.05,
                            .055,.045,.05,.055,.045,.05,.055 };
  double  v, Lb;

  unsigned int hTimer;
  double  gpuTime;

  // 'd_' prefix - GPU (device) memory space

  float  *d_v,*d_Lb;

  CUT_DEVICE_INIT();
  CUT_SAFE_CALL( cutCreateTimer(&hTimer) );

  for (i=0; i<NN; i++) h_lambda[i] = 0.2;

  // Copy all constants into constant memory

  cudaMemcpyToSymbol(N, &h_N, sizeof(h_N));
  cudaMemcpyToSymbol(Nmat, &h_Nmat, sizeof(h_Nmat));
  cudaMemcpyToSymbol(Nopt, &h_Nopt, sizeof(h_Nopt));
  cudaMemcpyToSymbol(delta, &h_delta, sizeof(h_delta));
  cudaMemcpyToSymbol(maturities, &h_maturities, sizeof(h_maturities));
  cudaMemcpyToSymbol(swaprates, &h_swaprates, sizeof(h_swaprates));
  cudaMemcpyToSymbol(lambda, &h_lambda, sizeof(h_lambda));

  // Allocate memory on host and device

  h_v      = (float *)malloc(sizeof(float)*NPATH);
  CUDA_SAFE_CALL( cudaMalloc((void **)&d_v, sizeof(float)*NPATH) );
  h_Lb     = (float *)malloc(sizeof(float)*NPATH);
  CUDA_SAFE_CALL( cudaMalloc((void **)&d_Lb, sizeof(float)*NPATH) );

  // Execute GPU kernel -- no Greeks

  CUDA_SAFE_CALL( cudaThreadSynchronize() );
  CUT_SAFE_CALL( cutResetTimer(hTimer) );
  CUT_SAFE_CALL( cutStartTimer(hTimer) );

  // Set up the execution configuration

  dim3 dimBlock(BLOCK_SIZE);
  dim3 dimGrid(GRID_SIZE);

  // Launch the device computation threads

  Pathcalc_Portfolio_KernelGPU2<<<dimGrid, dimBlock>>>(d_v);
  CUT_CHECK_ERROR("Pathcalc_Portfolio_kernelGPU2() execution failed\n");
  CUDA_SAFE_CALL( cudaThreadSynchronize() );

  // Read back GPU results and compute average

  CUDA_SAFE_CALL( cudaMemcpy(h_v, d_v, sizeof(float)*NPATH,
                  cudaMemcpyDeviceToHost) );
  CUT_SAFE_CALL( cutStopTimer(hTimer) );
  gpuTime = cutGetTimerValue(hTimer);

  v = 0.0;
  for (i=0; i<NPATH; i++) v += h_v[i];
  v = v / NPATH;

  printf("v  = %15.8f\n", v);
  printf("Time(No Greeks) : %f msec\n", gpuTime);

  // Execute GPU kernel -- Greeks

  CUDA_SAFE_CALL( cudaThreadSynchronize() );
  CUT_SAFE_CALL( cutResetTimer(hTimer) );
  CUT_SAFE_CALL( cutStartTimer(hTimer) );
```

```
  // Launch the device computation threads

  Pathcalc_Portfolio_KernelGPU<<<dimGrid, dimBlock>>>(d_v,d_Lb);
  CUT_CHECK_ERROR("Pathcalc_Portfolio_kernelGPU() execution failed\n");
  CUDA_SAFE_CALL( cudaThreadSynchronize() );

  // Read back GPU results and compute average

  CUDA_SAFE_CALL( cudaMemcpy(h_v, d_v, sizeof(float)*NPATH,
                  cudaMemcpyDeviceToHost) );
  CUDA_SAFE_CALL( cudaMemcpy(h_Lb, d_Lb, sizeof(float)*NPATH,
                  cudaMemcpyDeviceToHost) );
  CUT_SAFE_CALL( cutStopTimer(hTimer) );
  gpuTime = cutGetTimerValue(hTimer);

  v = 0.0;
  for (i=0; i<NPATH; i++) v += h_v[i];
  v = v / NPATH;

  Lb = 0.0;
  for (i=0; i<NPATH; i++) Lb += h_Lb[i];
  Lb = Lb / NPATH;

  printf("v  = %15.8f\n", v);
  printf("Lb = %15.8f\n", Lb);
  printf("Time (Greeks)   : %f msec\n", gpuTime);

  // Release GPU memory

  CUDA_SAFE_CALL( cudaFree(d_v));
  CUDA_SAFE_CALL( cudaFree(d_Lb));

  // Release CPU memory

  free(h_v);
  free(h_Lb);

  CUT_SAFE_CALL( cutDeleteTimer(hTimer) );
  CUT_EXIT(argc, argv);
}
```