Vector addition

```
// vecadd.cu

// add to PATH: /usr/local/cuda/bin/
// add to LD_LIBRARY_PATH: /usr/lib64/nvidia:/usr/local/cuda/lib64
// nvcc -o vecadd -O3 vecadd.cu

#include <stdio.h>

// CUDA kernel code
__global__ void VectorAdd (float *c, float *a, float *b, int vLen)
{
  int n;
  n = blockDim.x * blockIdx.x + threadIdx.x;
  if (n < vLen) c[n] = a[n] + b[n];
}

// host main routine
int main (void)
{
  float *h_a, *h_b, *h_c, *d_a , *d_b, *d_c;
  int vLen, nThread, nBlock, n;

  vLen = 50000;
// allocate host vectors
  h_a = (float *) malloc (vLen * sizeof (float));
  h_b = (float *) malloc (vLen * sizeof (float));
  h_c = (float *) malloc (vLen * sizeof (float));
// initialize
  for (n = 0; n < vLen; n ++) {
    h_a[n] = rand () / (float) RAND_MAX;
    h_b[n] = rand () / (float) RAND_MAX;
  }
// allocate device vectors
  cudaMalloc (&d_a, vLen * sizeof (float));
  cudaMalloc (&d_b, vLen * sizeof (float));
  cudaMalloc (&d_c, vLen * sizeof (float));
// copy host input vectors to device memory
  cudaMemcpy (d_a, h_a, vLen * sizeof (float), cudaMemcpyHostToDevice);
  cudaMemcpy (d_b, h_b, vLen * sizeof (float), cudaMemcpyHostToDevice);
// launch CUDA kernel
  nThread = 256;
  nBlock = (vLen + nThread - 1) / nThread;
  printf ("Kernel launch: grid with %d blocks of %d threads\n", nBlock, nThread);
  VectorAdd <<< nBlock, nThread >>> (d_c, d_a, d_b, vLen);
  cudaDeviceSynchronize ();
  cudaGetLastError ();
// copy device result vector from device memory to host
  cudaMemcpy (h_c, d_c, vLen * sizeof (float), cudaMemcpyDeviceToHost);
// check results
  for (n = 0; n < vLen; n ++) {
    if (fabs (h_a[n] + h_b[n] - h_c[n]) > 1e-5) {
      printf ("Error at element %d!\n", n);
      exit (0);
    }
  }
  printf ("Test OK\n");
// free device memory
  cudaFree (d_a);
  cudaFree (d_b);
  cudaFree (d_c);
// free host memory
  free (h_a);
  free (h_b);
  free (h_c);
  exit (0);
}
```

Matrix multiplication

```
// matmul.cu

#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/time.h>

void GetCurSecMusec (int *time);

void MatrixMulCPU (float *c, float *a, float *b, int aW, int aH, int bW)
{
  float ct;
  int i, j, k;
  for (i = 0; i < aH; i ++) {
    for (j = 0; j < bW; j ++) {
      ct = 0.f;
      for (k = 0; k < aW; k ++)
         ct += a[aW * i + k] * b[bW * k + j];
      c[bW * i + j] = ct;
    }
  }
}

// GPU kernel: each thread computes one element of C
__global__ void MatrixMul (float *c, float *a, float *b, int aW, int bW)
{
  float ct;
  int i, j, k;
  ct = 0.f;
  i = blockDim.y * blockIdx.y + threadIdx.y;
  j = blockDim.x * blockIdx.x + threadIdx.x;
  for (k = 0; k < aW; k ++)
     ct += a[aW * i + k] * b[bW * k + j];
  c[bW * i + j] = ct;
}

#define BLK_SIZE 32

// GPU kernel with SM: each thread computes one element of C, but in blocks
__global__ void MatrixMulSM (float *c, float *a, float *b, int aW, int bW)
{
  float ct;
  int ib, jb, nb, k, kt;
  // shared memory arrays used to store submatrices
  __shared__ float sh_a[BLK_SIZE * BLK_SIZE], sh_b[BLK_SIZE * BLK_SIZE];

  ib = BLK_SIZE * blockIdx.y + threadIdx.y;
  jb = BLK_SIZE * blockIdx.x + threadIdx.x;
  ct = 0.f;

  // loop over all sub-matrices of A and B required for block sub-matrix
  for (nb = 0; nb < aW / BLK_SIZE; nb ++) {
    kt = BLK_SIZE * threadIdx.y + threadIdx.x;
    sh_a[kt] = a[aW * ib + BLK_SIZE * nb + threadIdx.x];
    sh_b[kt] = b[bW * (BLK_SIZE + threadIdx.y) * nb + jb];
    // wait till all data in SM
    __syncthreads ();

    for (k = 0; k < BLK_SIZE; k ++) {
      ct += sh_a[BLK_SIZE * threadIdx.y + k] * sh_b[BLK_SIZE * k + threadIdx.x];
    }

    // synchronize betwen iterations
    __syncthreads ();
  }

  // write block sub-matrix to memory; each thread writes one element
  c[bW * ib + jb] = ct;
}

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

```
{
  float *h_a, *h_b, *h_c, *hh_c, *d_a, *d_b, *d_c;
  float mSec, gFlops;
  int timeB[2], timeF[2];
  int nRun, j, n;
  bool ok;
  cudaEvent_t tBgn, tEnd;
  int devID = 0;

  cudaDeviceProp deviceProp;
  cudaGetDevice (&devID);

  cudaGetDeviceProperties (&deviceProp, devID);

  printf ("GPU device %d: %s, compute capability %d.%d\n", devID,
     deviceProp.name, deviceProp.major, deviceProp.minor);

  dim3 aSize (12 * 32, 10 * 32);
  dim3 bSize (20 * 32, 12 * 32);

  printf ("sizes: A(%d,%d), B(%d,%d)\n", aSize.x, aSize.y, bSize.x, bSize.y);

  if (aSize.x != bSize.y) {
    printf ("Error: outer matrix dimensions must be equal. (%d != %d)\n",
       aSize.x, bSize.y);
    exit (EXIT_FAILURE);
  }

  h_a = (float *) malloc (aSize.x * aSize.y * sizeof (float));
  h_b = (float *) malloc (bSize.x * bSize.y * sizeof (float));
  h_c = (float *) malloc (aSize.y * bSize.x * sizeof (float));
  hh_c = (float *) malloc (aSize.y * bSize.x * sizeof (float));

  for (n = 0; n < aSize.x * aSize.y; n ++)
     h_a[n] = rand () / (float) RAND_MAX;
  for (n = 0; n < bSize.x * bSize.y; n ++)
     h_b[n] = rand () / (float) RAND_MAX;
  GetCurSecMusec (timeB);
  MatrixMulCPU (hh_c, h_a, h_b, aSize.x, aSize.y, bSize.x);
  GetCurSecMusec (timeF);
  mSec = 1e-3 * (1e6 * (timeF[1] - timeB[1]) + timeF[0] - timeB[0]);
  gFlops = (2. * (double) (aSize.x * aSize.y) * (double) bSize.x * 1.e-9) /
     (mSec / (1.e3f));
  printf ("CPU Time: %.1f msec  %.1f GFlop/s\n", mSec, gFlops);

  cudaMalloc (&d_a, aSize.x * aSize.y * sizeof (float));
  cudaMalloc (&d_b, bSize.x * bSize.y * sizeof (float));
  cudaMalloc (&d_c, aSize.y * bSize.x * sizeof (float));

  cudaMemcpy (d_a, h_a, aSize.x * aSize.y * sizeof (float), cudaMemcpyHostToDevice);
  cudaMemcpy (d_b, h_b, bSize.x * bSize.y * sizeof (float), cudaMemcpyHostToDevice);

  // kernel parameters
  dim3 nThread (BLK_SIZE, BLK_SIZE);
  dim3 nBlock (bSize.x / BLK_SIZE, aSize.y / BLK_SIZE);

  cudaEventCreate (&tBgn);
  cudaEventCreate (&tEnd);

  nRun = 10;

  printf ("w/o SM \n");
  cudaEventRecord (tBgn, 0);
  for (j = 0; j < nRun; j ++) {
    MatrixMul <<< nBlock, nThread >>> (d_c, d_a, d_b, aSize.x, bSize.x);
  }
  cudaDeviceSynchronize ();
  cudaEventRecord (tEnd, 0);
  cudaEventSynchronize (tEnd);
  cudaEventElapsedTime (&mSec, tBgn, tEnd);

  cudaMemcpy (h_c, d_c, aSize.y * bSize.x * sizeof (float), cudaMemcpyDeviceToHost);
  ok = true;
```

```
  for (n = 0; n < aSize.y * bSize.x; n ++) {
    if (fabs (h_c[n] - hh_c[n]) / fabs (h_c[n]) > 1e-6) ok = false;
  }
  mSec /= nRun;
  gFlops = (2. * (double) (aSize.x * aSize.y) * (double) bSize.x * 1e-9) /
     (1e-3 * mSec);
  printf ("   Time: %.1f msec  %.1f GFlop/s %s\n", mSec, gFlops, (ok ? "OK" : "Err"));

  printf ("with SM\n");
  cudaEventRecord (tBgn, 0);
  for (j = 0; j < nRun; j ++) {
    MatrixMulSM <<< nBlock, nThread >>> (d_c, d_a, d_b, aSize.x, bSize.x);
  }
  cudaDeviceSynchronize ();
  cudaEventRecord (tEnd, 0);
  cudaEventSynchronize (tEnd);
  cudaEventElapsedTime (&mSec, tBgn, tEnd);

  cudaMemcpy (h_c, d_c, aSize.y * bSize.x * sizeof (float), cudaMemcpyDeviceToHost);
  ok = true;
  for (n = 0; n < aSize.y * bSize.x; n ++) {
    if (fabs (h_c[n] - hh_c[n]) / fabs (h_c[n]) > 1e-6) ok = false;
  }
  mSec /= nRun;
  gFlops = (2. * (double) (aSize.x * aSize.y) * (double) bSize.x * 1e-9) /
     (1e-3 * mSec);
  printf ("   Time: %.1f msec  %.1f GFlop/s %s\n", mSec, gFlops, (ok ? "OK" : "Err"));

  cudaFree (d_a);
  cudaFree (d_b);
  cudaFree (d_c);
  free (h_a);
  free (h_b);
  free (h_c);
  free (hh_c);
  cudaEventDestroy (tBgn);
  cudaEventDestroy (tEnd);
  exit (0);
}

void GetCurSecMusec (int *time)
{
  struct timeval tv;

  gettimeofday (&tv, NULL);
  time[1] = tv.tv_sec;
  time[0] = tv.tv_usec;
}
```

Reduction – sum elements of vector

```
// reduction.cu

#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/time.h>

void GetCurSecMusec (int *time);

void ReduceCPU (float *res, float *data, int dSize)
{
  int n;
  res[0] = 0.;
  for (n = 0; n < dSize; n ++) res[0] += data[n];
}

__global__ void Reduce (float *res, float *data, int dSize)
{
  int ib, m;
  extern __shared__ float r_sh[];

  ib = blockIdx.x * blockDim.x + threadIdx.x;
  r_sh[threadIdx.x] = (ib < dSize) ? data[ib] : 0.f;
  __syncthreads ();
  for (m = blockDim.x / 2; m > 0; m /= 2) {
    if (threadIdx.x < m) r_sh[threadIdx.x] += r_sh[threadIdx.x + m];
    __syncthreads ();
  }
  if (threadIdx.x == 0) res[blockIdx.x] = r_sh[0];
}

__global__ void ReduceAt (float *res, float *data, int dSize)
{
  int ib, m;
  extern __shared__ float r_sh[];

  ib = blockIdx.x * blockDim.x + threadIdx.x;
  r_sh[threadIdx.x] = (ib < dSize) ? data[ib] : 0.f;
  __syncthreads ();
  for (m = blockDim.x / 2; m > 0; m /= 2) {
    if (threadIdx.x < m) r_sh[threadIdx.x] += r_sh[threadIdx.x + m];
    __syncthreads ();
  }
  if (threadIdx.x == 0) atomicAdd (res, r_sh[0]);
}

__device__ unsigned int bCount = 0;

__global__ void ReduceFence (volatile float *res, float *data, int dSize)
{
  int ib, m;
  extern __shared__ float r_sh[];
  int *cnt = (int *) (r_sh + blockDim.x);

  ib = blockIdx.x * blockDim.x + threadIdx.x;
  r_sh[threadIdx.x] = (ib < dSize) ? data[ib] : 0.f;
  __syncthreads ();
  for (m = blockDim.x / 2; m > 0; m /= 2) {
    if (threadIdx.x < m) r_sh[threadIdx.x] += r_sh[threadIdx.x + m];
    __syncthreads ();
  }
  if (threadIdx.x == 0) {
    res[blockIdx.x] = r_sh[0];
    __threadfence ();
    cnt[0] = atomicInc (&bCount, gridDim.x);
  }
  __syncthreads();
  if (cnt[0] == gridDim.x - 1) {
    r_sh[threadIdx.x] = 0.f;
    for (m = 0; m < gridDim.x; m += blockDim.x) {
       r_sh[threadIdx.x] += res[threadIdx.x + m];
```

```
    }
    __syncthreads ();
    for (m = blockDim.x / 2; m > 0; m /= 2) {
      if (threadIdx.x < m) r_sh[threadIdx.x] += r_sh[threadIdx.x + m];
      __syncthreads ();
    }
    if (threadIdx.x == 0) {
      res[0] = r_sh[0];
      bCount = 0;
    }
  }
}

void CheckErrGpu (const char *fn, const char *txt)
{
  cudaError_t err;

  err = cudaGetLastError ();
  if (err != cudaSuccess) {
    printf ("CUDA error (%s): %s: %s\n", fn, txt, cudaGetErrorString (err));
    exit (-1);
  }
}

int main (int argc, char **argv)
{
  float *h_data, *h_res, *d_data, *d_res, hh_res[1];
  float mSec;
  int timeB[2], timeF[2];
  int dSize, nThread, nBlock, j, n, nRun;
  cudaEvent_t tBgn, tEnd;
  int devID = 0;

  cudaDeviceProp deviceProp;
  cudaGetDevice (&devID);

  cudaGetDeviceProperties (&deviceProp, devID);

  printf ("GPU device %d: %s, compute capability %d.%d\n", devID,
     deviceProp.name, deviceProp.major, deviceProp.minor);

  dSize = 1 << 20;
  nThread = 256;
  nBlock = (dSize + nThread - 1) / nThread;
  h_data = (float *) malloc (dSize * sizeof (float));
  h_res = (float *) malloc (nBlock * sizeof (float));

  cudaMalloc (&d_data, dSize * sizeof (float));
  cudaMalloc (&d_res, nBlock * sizeof (float));

  for (n = 0; n < dSize; n ++) h_data[n] = rand () / (float) RAND_MAX;
  GetCurSecMusec (timeB);
  ReduceCPU (hh_res, h_data, dSize);
  GetCurSecMusec (timeF);
  mSec = 1e-3 * (1e6 * (timeF[1] - timeB[1]) + timeF[0] - timeB[0]);
  printf ("CPU:       %.1f msec\n", mSec);

  nRun = 20;
  cudaEventCreate (&tBgn);
  cudaEventCreate (&tEnd);

  cudaMemcpy (d_data, h_data, dSize * sizeof (float),
     cudaMemcpyHostToDevice);
  cudaEventRecord (tBgn, 0);
  for (j = 0; j < nRun; j ++) {
    Reduce <<< nBlock, nThread, nThread * sizeof (float) >>> (d_res, d_data, dSize);
  }
  cudaDeviceSynchronize ();
  CheckErrGpu ("", "Kernel exec failed");
  cudaEventRecord (tEnd, 0);
  cudaEventSynchronize (tEnd);
  cudaEventElapsedTime (&mSec, tBgn, tEnd);
```

```
  cudaMemcpy (h_res, d_res, nBlock * sizeof (float),
     cudaMemcpyDeviceToHost);
  for (n = 1; n < nBlock; n ++) h_res[0] += h_res[n];
  printf ("GPU+host:  %.1f msec %.2e\n", mSec / nRun,
     fabs (h_res[0] / hh_res[0] - 1.));

  cudaMemcpy (d_data, h_data, dSize * sizeof (float), cudaMemcpyHostToDevice);
  cudaEventRecord (tBgn, 0);
  for (j = 0; j < nRun; j ++) {
    cudaMemset (d_res, 0, sizeof (float));
    ReduceAt <<< nBlock, nThread, nThread * sizeof (float) >>> (d_res, d_data, dSize);
  }
  cudaDeviceSynchronize ();
  cudaEventRecord (tEnd, 0);
  cudaEventSynchronize (tEnd);
  cudaEventElapsedTime (&mSec, tBgn, tEnd);
  cudaMemcpy (h_res, d_res, sizeof (float), cudaMemcpyDeviceToHost);
  printf ("GPU+AT:    %.1f msec %.2e\n", mSec / nRun,
     fabs (h_res[0] / hh_res[0] - 1.));

  cudaMemcpy (d_data, h_data, dSize * sizeof (float), cudaMemcpyHostToDevice);
  cudaEventRecord (tBgn, 0);
  for (j = 0; j < nRun; j ++) {
    ReduceFence <<< nBlock, nThread, nThread * sizeof (float) + sizeof (int) >>>
       (d_res, d_data, dSize);
  }
  cudaDeviceSynchronize ();
  cudaEventRecord (tEnd, 0);
  cudaEventSynchronize (tEnd);
  cudaEventElapsedTime (&mSec, tBgn, tEnd);
  CheckErrGpu ("", "Kernel exec failed");
  cudaMemcpy (h_res, d_res, sizeof (float), cudaMemcpyDeviceToHost);
  printf ("GPU+fence: %.1f msec %.2e\n", mSec / nRun,
     fabs (h_res[0] / hh_res[0] - 1.));
  exit (0);
}

void GetCurSecMusec (int *time)
{
  struct timeval tv;

  gettimeofday (&tv, NULL);
  time[1] = tv.tv_sec;
  time[0] = tv.tv_usec;
}
```

Histogram evaluation

```
// histogram.cu

#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/time.h>

#define BL_SIZE 64

__global__ void Hist (int *hist, int hSize, float *data, int dSize)
{
  int ib, j, k;

  ib = blockIdx.x * blockDim.x + threadIdx.x;
  for (j = 0; j < BL_SIZE; j ++) {
    k = (int) (data[ib] * (hSize - 1));
    atomicAdd (&hist[k], 1);
    ib += dSize / BL_SIZE;
  }
}

__global__ void HistSM (int *hist, int hSize, float *data, int dSize)
{
  extern __shared__ int sh_hist[];
  int ib, j, k, nb;

  for (j = threadIdx.x; j < hSize; j += blockDim.x) sh_hist[j] = 0;
  __syncthreads ();
  ib = blockIdx.x * blockDim.x + threadIdx.x;
  for (nb = 0; nb < BL_SIZE; nb ++) {
    k = (int) (data[ib] * (hSize - 1));
    atomicAdd (&sh_hist[k], 1);
    ib += dSize / BL_SIZE;
  }
  __syncthreads ();
  for (j = threadIdx.x; j < hSize; j += blockDim.x) atomicAdd (&hist[j], sh_hist[j]);
}

void HistCPU (int *hist, int hSize, float *data, int dSize)
{
  int k, n;
  for (n = 0; n < dSize; n ++) {
    k = (int) (data[n] * (hSize - 1));
    ++ hist[k];
  }
}

void GetCurSecMusec (int *time);

int main (int argc, char **argv)
{
  float *h_data, *d_data, mSec;
  int *h_hist, *hh_hist, *d_hist, dSize, hSize, nThread, nBlock;

  int timeB[2], timeF[2], n, nErr;
  cudaEvent_t tBgn, tEnd;
  cudaError_t err;

  err = cudaSuccess;
  dSize = 1<<25;
  hSize = ((500 + 15) / 16) * 16;
  h_data = (float *) malloc (dSize * sizeof (float));
  h_hist = (int *) malloc (hSize * sizeof (int));
  hh_hist = (int *) malloc (hSize * sizeof (int));

  srandom (1001);
  for (n = 0; n < dSize; n ++) h_data[n] = rand () / (float) RAND_MAX;

  for (n = 0; n < hSize; n ++) h_hist[n] = 0;
  GetCurSecMusec (timeB);
  HistCPU (h_hist, hSize, h_data, dSize);
```

```
  GetCurSecMusec (timeF);
  mSec = (1e6 * (timeF[1] - timeB[1]) + timeF[0] - timeB[0]);
  printf ("CPU data/musec %.1f\n", dSize / mSec);

  cudaMalloc (&d_data, dSize * sizeof (float));
  cudaMalloc (&d_hist, hSize * sizeof (int));
  cudaMemcpy (d_data, h_data, dSize * sizeof (float), cudaMemcpyHostToDevice);
  nThread = 256;
  nBlock = (dSize / BL_SIZE) / nThread;
  cudaEventCreate (&tBgn);
  cudaEventCreate (&tEnd);

  for (n = 0; n < hSize; n ++) hh_hist[n] = 0;
  cudaMemcpy (d_hist, hh_hist, hSize * sizeof (int), cudaMemcpyHostToDevice);
  cudaEventRecord (tBgn, 0);
  Hist <<< nBlock, nThread >>> (d_hist, hSize, d_data, dSize);
  cudaDeviceSynchronize ();
  err = cudaGetLastError ();
  if (err != cudaSuccess) {
    printf ("CUDA error: Hist failed: %s\n", cudaGetErrorString (err));
    exit (0);
  }
  cudaEventRecord (tEnd, 0);
  cudaEventSynchronize (tEnd);
  cudaEventElapsedTime (&mSec, tBgn, tEnd);
  printf ("GPU data/musec %.1f  ", dSize / (mSec * 1e3));
  cudaMemcpy (hh_hist, d_hist, hSize * sizeof (int), cudaMemcpyDeviceToHost);
  nErr = 0;
  for (n = 0; n < hSize; n ++) {
    if (hh_hist[n] != h_hist[n]) ++ nErr;
  }
  if (nErr > 0) printf ("err\n");
  else printf ("OK\n");

  for (n = 0; n < hSize; n ++) hh_hist[n] = 0;
  cudaMemcpy (d_hist, hh_hist, hSize * sizeof (int), cudaMemcpyHostToDevice);
  cudaEventRecord (tBgn, 0);
  HistSM <<< nBlock, nThread, hSize * sizeof (int) >>>
     (d_hist, hSize, d_data, dSize);
  cudaDeviceSynchronize ();
  err = cudaGetLastError ();
  if (err != cudaSuccess) {
    printf ("CUDA error: HistSM failed: %s\n", cudaGetErrorString (err));
    exit (0);
  }
  cudaEventRecord (tEnd, 0);
  cudaEventSynchronize (tEnd);
  cudaEventElapsedTime (&mSec, tBgn, tEnd);
  printf ("GPU-SM data/musec %.1f  ", dSize / (mSec * 1e3));
  cudaMemcpy (hh_hist, d_hist, hSize * sizeof (int), cudaMemcpyDeviceToHost);
  nErr = 0;
  for (n = 0; n < hSize; n ++) {
    if (hh_hist[n] != h_hist[n]) ++ nErr;
  }
  if (nErr > 0) printf ("err\n");
  else printf ("OK\n");

  cudaFree (d_data);
  cudaFree (d_hist);
  cudaEventDestroy (tBgn);
  cudaEventDestroy (tEnd);
}

void GetCurSecMusec (int *time)
{
  struct timeval tv;

  gettimeofday (&tv, NULL);
  time[1] = tv.tv_sec;
  time[0] = tv.tv_usec;
}
```

Molecular dynamics (all pairs)

```
// mdallpairs.cu

#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/time.h>

#define USE_SM          1

// macros

#define AllocMem(a, n, t)  a = (t *) malloc ((n) * sizeof (t))
#define FreeMem(a) free (a)

#define CMalloc(a, n, t) \
  err = cudaMalloc ((void**) &a, n * sizeof (t))
#define CFree(a) cudaFree (a)

#define VOp(v1, v2, op, v3) \
   (v1).x = (v2).x op (v3).x, (v1).y = (v2).y op (v3).y, (v1).z = (v2).z op (v3).z
#define VSCopy(v1, s2, v2) \
   (v1).x = (s2) * (v2).x, (v1).y = (s2) * (v2).y, (v1).z = (s2) * (v2).z
#define VSOp(v1, v2, op, s3, v3) \
   (v1).x = (v2).x op (s3) * (v3).x, \
   (v1).y = (v2).y op (s3) * (v3).y, \
   (v1).z = (v2).z op (s3) * (v3).z
#define VSet(v, sx, sy, sz) (v).x = sx, (v).y = sy, (v).z = sz
#define VSetAll(v, s) VSet (v, s, s, s)
#define VZero(v) VSetAll (v, 0.f)
#define VDot(v1, v2) ((v1).x * (v2).x + (v1).y * (v2).y + (v1).z * (v2).z)
#define VProd(v) ((v).x * (v).y * (v).z)

#define VAdd(v1, v2, v3) VOp (v1, v2, +, v3)
#define VDiv(v1, v2, v3) VOp (v1, v2, /, v3)
#define VLenSq(v) VDot (v, v)
#define VMul(v1, v2, v3) VOp (v1, v2, *, v3)
#define VSAdd(v1, v2, s3, v3) VSOp (v1, v2, +, s3, v3)
#define VScale(v1, s1) VSCopy (v1, s1, v1)
#define VSub(v1, v2, v3) VOp (v1, v2, -, v3)
#define VVAdd(v1, v2) VAdd (v1, v1, v2)
#define VVSAdd(v1, s2, v2) VSAdd (v1, v1, s2, v2)

typedef struct {
  float4 *rMol, *rvMol, *raMol;
  float3 region;
  float *prop, rrCut, deltaT;
  int nMol, szPropArray;
} Conf;

Conf confH;
Conf confD;
float3 region;
int3 initUcell;
float velMag, temperature, density, rCut, vvSum, uSum, kinEnergy, totEnergy,
   kinEnergySum, totEnergySum, kinEnInitSum;
int edgeCell, stepEquil, stepInitTemp, stepAvg, stepLimit, stepCount, nThread, nBlock;

__constant__ Conf conf;

// prototypes

void SetParams ();
void AllocArraysHost ();
void FreeArraysHost ();
void SingleStep ();
void InitState ();
void InitRand (int randSeed);
void VRandR (float3 *w);
void GetCurSecMusec (int *time);
void PerfTime (int code);
void AllocArraysGpu ();
```

```
void FreeArraysGpu ();
void CheckErrGpu (const char *fn, const char *txt);

void ComputeForcesGpu ();
void LeapfrogStep1Gpu ();
void LeapfrogStep2Gpu ();
void ApplyBoundaryCondGpu ();
void EvalPropsGpu ();
void AdjustInitTempGpu (float velFac);

int main (int argc, char **argv)
{
  float tVal, tValTot, devMemSize;
  int timeB[2], timeF[2];

  SetParams ();
  confD = confH;
  AllocArraysHost ();
  AllocArraysGpu ();
  cudaMemcpyToSymbol (conf, &confD, sizeof (Conf), 0, cudaMemcpyHostToDevice);
  InitState ();
  printf ("\n Input: edgeCell = %d  (nMol: %d)  stepLimit = %d  ",
     edgeCell, confH.nMol, stepLimit);
  printf ("GPU threads: %d\n", nThread);
#if USE_SM
  printf (" Use SM\n");
#else
  printf (" No SM\n");
#endif
  printf (" dens, temp, cut: %.3f %.3f %.3f\n\n", density, temperature, rCut);
  printf ("  Step     Tot energy   Kin energy   Time/m-step\n");
  fflush (stdout);
  tValTot = 0.;
  GetCurSecMusec (timeB);
  PerfTime (-1);
  while (stepCount < stepLimit) {
    SingleStep ();
    if (stepCount % stepAvg == 0) {
      GetCurSecMusec (timeF);
      tVal = (1e6 * (timeF[1] - timeB[1]) + timeF[0] - timeB[0]);
      tValTot += tVal;
      printf ("%6d  %12.7f %12.7f %8.4f\n", stepCount,
         totEnergySum / stepAvg, kinEnergySum / stepAvg,
         tVal / ((double) stepAvg * confH.nMol));
      fflush (stdout);
      totEnergySum = 0.f;
      kinEnergySum = 0.f;
      GetCurSecMusec (timeB);
    }
  }
  devMemSize = confH.nMol * (3 * sizeof (float4));
  printf ("\n Run time: %.1f sec (%.4f musec/mol)"
     "   Dev mem: %.2f MB  (%.0f byte/mol)\n",
     tValTot * 1e-6, tValTot / ((float) stepLimit * confH.nMol),
     devMemSize  / 1e6, devMemSize / confH.nMol);
  PerfTime (-2);
  printf ("\n============================================================\n\n");
  FreeArraysHost ();
  FreeArraysGpu ();
  return (0);
}

// set parameters
void SetParams ()
{
  density = 0.8f;
  rCut = 1.12246f;
  edgeCell = 16;
  nThread = 256;
  stepAvg = 1000;
  stepEquil = 500;
  stepInitTemp = 20;
  stepLimit = 5000;
```

```
  temperature = 1.f;
  confH.deltaT = 0.005f;
  InitRand (17);
  VSet (initUcell, edgeCell, edgeCell, edgeCell);
  VSCopy (confH.region, 1.f / powf (density, 1.f/3.f), initUcell);
  confH.nMol = VProd (initUcell);
  velMag = sqrtf (3.f * temperature);
  nBlock = (confH.nMol + nThread - 1) / nThread;
  if (nBlock * nThread != confH.nMol)
     printf ("\n ** nMol not a multiple of nThread !\n");
  confH.szPropArray = 2 * nBlock;
  confH.rrCut = rCut * rCut;
}

// allocate arrays
void AllocArraysHost ()
{

  AllocMem (confH.rMol, confH.nMol, float4);
  AllocMem (confH.rvMol, confH.nMol, float4);
  AllocMem (confH.raMol, confH.nMol, float4);
  AllocMem (confH.prop, confH.szPropArray, float);
}

void AllocArraysGpu ()
{
  cudaError_t err;

  CMalloc (confD.rMol, confD.nMol, float4);
  CMalloc (confD.rvMol, confD.nMol, float4);
  CMalloc (confD.raMol, confD.nMol, float4);
  CMalloc (confD.prop, confD.szPropArray, float);
  if (err != cudaSuccess) CheckErrGpu ("AllocArrays", "Mem alloc failed");
}

// free arrays
void FreeArraysHost ()
{
  FreeMem (confH.rMol);
  FreeMem (confH.rvMol);
  FreeMem (confH.raMol);
  FreeMem (confH.prop);
}

void FreeArraysGpu ()
{
  CFree (confD.rMol);
  CFree (confD.rvMol);
  CFree (confD.raMol);
  CFree (confD.prop);
}

// single md step
void SingleStep ()
{
  float velFac;

  ++ stepCount;
  PerfTime (0);
  LeapfrogStep1Gpu ();
  PerfTime (1);
  ApplyBoundaryCondGpu ();
  PerfTime (2);
  ComputeForcesGpu ();
  PerfTime (3);
  LeapfrogStep2Gpu ();
  PerfTime (4);
  EvalPropsGpu ();
  kinEnergy = 0.5f * vvSum / confH.nMol;
  totEnergy = kinEnergy + 0.5f * uSum / confH.nMol;
  PerfTime (5);
  if (stepCount < stepEquil) {
    kinEnInitSum += kinEnergy;
```

```
    if (stepCount % stepInitTemp == 0) {
      velFac = velMag / sqrtf (2.f * kinEnInitSum / stepInitTemp);
      AdjustInitTempGpu (velFac);
      kinEnInitSum = 0.f;
    }
  }
  totEnergySum += totEnergy;
  kinEnergySum += kinEnergy;
}

// compute force and potential energy
#define VWrapD(r, d, t)                    \
   if (fabsf (r.t) >= 0.5f * d.t) r.t -= copysignf (d.t, r.t);

#define VWrap(r, d) {  \
   VWrapD (r, d, x);   \
   VWrapD (r, d, y);   \
   VWrapD (r, d, z);   \
 }

#if ! USE_SM
__global__ void ComputeForcesDev ()
{
  float4 raMolP;
  float3 dr;
  float fcVal, rr, rri, rri3, uMolP;
  int mId, j;

  mId = blockIdx.x * blockDim.x + threadIdx.x;
  if (mId < conf.nMol) {
    VZero (raMolP);
    uMolP = 0.;
    for (j = 0; j < conf.nMol; j ++) {
      VSub (dr, conf.rMol[mId], conf.rMol[j]);
      VWrap (dr, conf.region);
      rr = VLenSq (dr);
      if (rr < conf.rrCut) {
        if (rr > 0.) rri = 1.f / rr;
        else rri = 0.;
        rri3 = rri * rri * rri;
        fcVal = 48.f * rri3 * (rri3 - 0.5f) * rri;
        VVSAdd (raMolP, fcVal, dr);
        uMolP += 4.f * rri3 * (rri3 - 1.f) + 1.f;
      }
    }
    uMolP -= 1.f;
    raMolP.w = uMolP;
    conf.raMol[mId] = raMolP;
  }
}
#else
__global__ void ComputeForcesDev ()
{
  extern __shared__ float4 sh_r[];
  float4 raMolP;
  float3 dr;
  float fcVal, rr, rri, rri3, uMolP;
  int mId, nb, j;

  mId = blockIdx.x * blockDim.x + threadIdx.x;
  if (mId < conf.nMol) {
    VZero (raMolP);
    uMolP = 0.;
    for (j = 0; j < conf.nMol; j += blockDim.x) {
      sh_r[threadIdx.x] = conf.rMol[j + threadIdx.x];
      __syncthreads ();
      for (nb = 0; nb < blockDim.x; nb ++) {
        VSub (dr, conf.rMol[mId], sh_r[nb]);
        VWrap (dr, conf.region);
        rr = VLenSq (dr);
        if (rr < conf.rrCut) {
          if (rr > 0.) rri = 1.f / rr;
          else rri = 0.;
```

```
          rri3 = rri * rri * rri;
          fcVal = 48.f * rri3 * (rri3 - 0.5f) * rri;
          VVSAdd (raMolP, fcVal, dr);
          uMolP += 4.f * rri3 * (rri3 - 1.f) + 1.f;
        }
      }
      __syncthreads ();
    }
    uMolP -= 1.f;
    raMolP.w = uMolP;
    conf.raMol[mId] = raMolP;
  }
}
#endif

void ComputeForcesGpu ()
{
#if ! USE_SM
  ComputeForcesDev <<< nBlock, nThread >>> ();
#else
  ComputeForcesDev <<< nBlock, nThread, nThread * sizeof (float4) >>> ();
#endif
  cudaDeviceSynchronize ();
  CheckErrGpu ("ComputeForces", "Kernel exec failed");
}

// integrate over timestep - stage 1
__global__ void LeapfrogStep1Dev ()
{
  int mId;

  mId = blockIdx.x * blockDim.x + threadIdx.x;
  if (mId < conf.nMol) {
    VVSAdd (conf.rvMol[mId], 0.5f * conf.deltaT, conf.raMol[mId]);
    VVSAdd (conf.rMol[mId], conf.deltaT, conf.rvMol[mId]);
  }
}

void LeapfrogStep1Gpu ()
{
  LeapfrogStep1Dev <<< nBlock, nThread >>> ();
  cudaDeviceSynchronize ();
  CheckErrGpu ("LeapfrogStep1", "Kernel exec failed");
}

// integrate over timestep - stage 2
__global__ void LeapfrogStep2Dev ()
{
  int mId;

  mId = blockIdx.x * blockDim.x + threadIdx.x;
  if (mId < conf.nMol) {
    VVSAdd (conf.rvMol[mId], 0.5f * conf.deltaT, conf.raMol[mId]);
  }
}

void LeapfrogStep2Gpu ()
{
  LeapfrogStep2Dev <<< nBlock, nThread >>> ();
  cudaDeviceSynchronize ();
  CheckErrGpu ("LeapfrogStep2", "Kernel exec failed");
}

// handle periodic boundaries
__global__ void ApplyBoundaryCondDev ()
{
  int mId;

  mId = blockIdx.x * blockDim.x + threadIdx.x;
  if (mId < conf.nMol) {
    VWrap (conf.rMol[mId], conf.region);
  }
}
```

```
void ApplyBoundaryCondGpu ()
{
  ApplyBoundaryCondDev <<< nBlock, nThread >>> ();
  cudaDeviceSynchronize ();
  CheckErrGpu ("ApplyBoundaryCond", "Kernel exec failed");
}

// adjust velocities for required temperature
__global__ void AdjustInitTempDev (float velFac)
{
  int mId;

  mId = blockIdx.x * blockDim.x + threadIdx.x;
  if (mId < conf.nMol) {
    VScale (conf.rvMol[mId], velFac);
  }
}

void AdjustInitTempGpu (float velFac)
{
  AdjustInitTempDev <<< nBlock, nThread >>> (velFac);
  cudaDeviceSynchronize ();
  CheckErrGpu ("AdjustInitTemp", "Kernel exec failed");
}

// evaluate properties
__global__ void EvalPropsDev ()
{
  extern __shared__ float sh_p[];
  int mId, ib, k;

  k = 2 * threadIdx.x;
  mId = blockIdx.x * blockDim.x + threadIdx.x;
  if (mId < conf.nMol) {
    sh_p[k + 0] = VLenSq (conf.rvMol[mId]);
    sh_p[k + 1] = conf.raMol[mId].w;
  } else {
    sh_p[k + 0] = 0.f;
    sh_p[k + 1] = 0.f;
  }
  __syncthreads ();
  for (ib = blockDim.x >> 1; ib > 0; ib >>= 1) {
    if (ib > threadIdx.x) {
      sh_p[k + 0] += sh_p[2 * (threadIdx.x + ib) + 0];
      sh_p[k + 1] += sh_p[2 * (threadIdx.x + ib) + 1];
    }
    __syncthreads ();
  }
  if (threadIdx.x == 0) {
    conf.prop[2 * blockIdx.x + 0] = sh_p[0];
    conf.prop[2 * blockIdx.x + 1] = sh_p[1];
  }
}

void EvalPropsGpu ()
{
  int m;

  EvalPropsDev <<< nBlock, nThread, 2 * nThread * sizeof (float) >>> ();
  cudaDeviceSynchronize ();
  CheckErrGpu ("EvalProps", "Kernel exec failed");
  cudaMemcpy (confH.prop, confD.prop, 2 * nBlock * sizeof (float),
     cudaMemcpyDeviceToHost);
  CheckErrGpu ("DevToHost", "failed");
  vvSum = 0.f;
  uSum = 0.f;
  for (m = 0; m < nBlock; m ++) {
    vvSum += confH.prop[2 * m + 0];
    uSum += confH.prop[2 * m + 1];
  }
}
```

```
// initialize mol variables
void InitState ()
{
  float4 c;
  float3 w, gap, vSum;
  int n, nx, ny, nz;

  VDiv (gap, confH.region, initUcell);
  n = 0;
  for (nz = 0; nz < initUcell.z; nz ++) {
    for (ny = 0; ny < initUcell.y; ny ++) {
      for (nx = 0; nx < initUcell.x; nx ++) {
        VSet (c, nx + 0.5f, ny + 0.5f, nz + 0.5f);
        VMul (c, c, gap);
        VVSAdd (c, -0.5f, confH.region);
        confH.rMol[n ++] = c;
      }
    }
  }
  VZero (vSum);
  for (n = 0; n < confH.nMol; n ++) {
    VRandR (&w);
    VSCopy (confH.rvMol[n], velMag, w);
    VVAdd (vSum, confH.rvMol[n]);
  }
  VScale (vSum, -1.f / confH.nMol);
  for (n = 0; n < confH.nMol; n ++) VVAdd (confH.rvMol[n], vSum);
  for (n = 0; n < confH.nMol; n ++) VZero (confH.raMol[n]);
  stepCount = 0;
  totEnergySum = 0.f;
  kinEnergySum = 0.f;
  kinEnInitSum = 0.f;
  cudaMemcpy (confD.rMol, confH.rMol, confH.nMol * sizeof (float4),
     cudaMemcpyHostToDevice);
  cudaMemcpy (confD.rvMol, confH.rvMol, confH.nMol * sizeof (float4),
     cudaMemcpyHostToDevice);
  cudaMemcpy (confD.raMol, confH.raMol, confH.nMol * sizeof (float4),
     cudaMemcpyHostToDevice);
  CheckErrGpu ("HostToDev", "failed");
}

// random number generation
void InitRand (int randSeed)
{
  srand48 ((long int) randSeed);
}

// generate randomly directed unit vector
void VRandR (float3 *p)
{
  float s, x, y;

  s = 2.f;
  while (s > 1.) {
    x = 2.f * drand48 () - 1.f;
    y = 2.f * drand48 () - 1.f;
    s = x * x + y * y;
  }
  p->z = 1.f - 2.f * s;
  s = 2.f * sqrtf (1.f - s);
  p->x = s * x;
  p->y = s * y;
}

// clock functions
#define NTIME_ACCUM  12

void PerfTime (int code)
{
  static double tVal, tValP, tAccum[NTIME_ACCUM];
  double t, tTot;
  int time[2];
  int k;
```

```
  if (code >= 0) {
    GetCurSecMusec (time);
    t = 1e6 * time[1] + time[0];
    tValP = tVal;
    tVal = t;
    if (code > 0 && code < NTIME_ACCUM)
       tAccum[code] += tVal - tValP;
  } else if (code == -1) {
    for (k = 0; k < NTIME_ACCUM; k ++) {
      tAccum[k] = 0.;
    }
  } else if (code == -2) {
    tTot = 0.;
    for (k = 1; k < NTIME_ACCUM; k ++) tTot += tAccum[k];
    if (tTot == 0.) tTot = 1.;
    printf (" Time fracs: ");
    for (k = 1; k < NTIME_ACCUM; k ++) {
      if (tAccum[k] == 0.) break;
      printf (" %.3f", tAccum[k] / tTot);
    }
    printf ("\n");
  }
}

void GetCurSecMusec (int *time)
{
  struct timeval tv;

  gettimeofday (&tv, NULL);
  time[1] = tv.tv_sec;
  time[0] = tv.tv_usec;
}

// gpu error check
void CheckErrGpu (const char *fn, const char *txt)
{
  cudaError_t err;

  err = cudaGetLastError ();
  if (err != cudaSuccess) {
    printf ("CUDA error (%s): %s: %s\n", fn, txt, cudaGetErrorString (err));
    exit (-1);
  }
}
```