CISC 372: Parallel Computing
CUDA, part 2

Stephen F. Siegel

Department of Computer and Information Sciences
University of Delaware

Memory management in CUDA

- there is memory on the device, and memory on the host
  - they are separate!
- a C pointer can point to either kind of memory
  - a variable of type `int*` might point to memory on host, or to memory on device
  - a device pointer should only be dereferenced on the device
  - a host pointer should only be dereferenced on the host
  - it is up to you (the programmer) to keep it straight!
- there are functions that are usually called on the host to...
  - allocate global memory on the device
    - and get a pointer to the newly allocated memory
  - free allocated memory on the device
  - copy memory from host to device and from device to host
- typically, the device global memory is allocated before invoking the kernel
  - and freed after the kernel terminates
- memory is copied before or after kernel invocations

Functions: memory management

- see NVIDIA CUDA Runtime API
  - `http://docs.nvidia.com/cuda/cuda-runtime-api/index.html`
  - the constant `cudaSuccess` of enum type `cudaError_t` indicates success
- `cudaError_t cudaMalloc ( void** devPtr, size_t size )`
  - allocate memory on the device
- `cudaError_t cudaFree ( void* devPtr )`
  - frees memory on the device
- `cudaError_t cudaMemcpy ( void* dst, const void* src, size_t count, cudaMemcpyKind kind )`
  - copy memory
- memory copy kinds
  - `cudaMemcpyHostToHost`
  - `cudaMemcpyHostToDevice`
  - `cudaMemcpyDeviceToHost`
  - `cudaMemcpyDeviceToDevice`

Example: vector addition

- two array of doubles are added pointwise into a third array
- CUDA is used to do the work:
  1. host allocates memory on device for all three arrays
  2. host copies the two arrays from host to device
  3. host launches a kernel with multiple blocks and multiple threads per block
  4. kernel writes the result into the previously allocated device global memory
  5. after kernel terminates, host copies result from device to host

add.cu: kernel code: cyclic distribution of tasks

```
__global__ void vec_add(int n, double * a, double * b, double * c) {
  int nthreads = gridDim.x * blockDim.x;
  int tid = blockDim.x * blockIdx.x + threadIdx.x;

  for (int i=tid; i<n; i+=nthreads)
    c[i] = a[i] + b[i];
}
```

- recall:
  - `gridDim.x` : number of blocks
  - `blockIdx.x` : ID number of this block
  - `blockDim.x` : number of threads per block
  - `threadIdx.x` : local ID number of this thread within its block
- therefore

```
blockDim.x * blockIdx.x + threadIdx.x
```

gives a unique global thread ID number to the thread

add.cu: host code, part 1: host initialization

```
const int nblocks = 45;
const int threadsPerBlock = 128;
int main(int argc, char * argv[]) {
  int N, err;
  assert(argc == 2);
  N = atoi(argv[1]);
  assert(N>=1);
  printf("N = %d\n", N);  fflush(stdout);
  double * a = (double*)malloc(N*sizeof(double));  assert(a);
  double * b = (double*)malloc(N*sizeof(double));  assert(b);
  double * c = (double*)malloc(N*sizeof(double));  assert(c);
  for (int i=0; i<N; i++) {
    a[i] = sin(i);
    b[i] = cos(i);
  }
  printf("Host initialization complete.\n");  fflush(stdout);
```

add.cu: host code, part 2: device initialization

```
double * a_dev, * b_dev, * c_dev, start_time = mytime();
err = cudaMalloc((void**)&a_dev, N*sizeof(double));
assert(err == cudaSuccess);
err = cudaMalloc((void**)&b_dev, N*sizeof(double));
assert(err == cudaSuccess);
err = cudaMalloc((void**)&c_dev, N*sizeof(double));
assert(err == cudaSuccess);
err = cudaMemcpy(a_dev, a, N*sizeof(double), cudaMemcpyHostToDevice);
assert(err == cudaSuccess);
err = cudaMemcpy(b_dev, b, N*sizeof(double), cudaMemcpyHostToDevice);
assert(err == cudaSuccess);
printf("Device initialization complete.\n");  fflush(stdout);
```

- convention: names of variables pointing to memory on device end in `_dev`
- first arg to `cudaMalloc` points to where the result should be returned
  - must be explicitly cast to `void**`
- good practice: always check each function returns `cudaSuccess`

add.cu: host code, part 3: kernel invocation, wrap-up

```
vec_add<<<nblocks, threadsPerBlock>>>(N, a_dev, b_dev, c_dev);
cudaMemcpy(c, c_dev, N*sizeof(double), cudaMemcpyDeviceToHost);
printf("Result obtained.  Time: %lf\n", mytime() - start_time);
cudaFree(a_dev);
cudaFree(b_dev);
cudaFree(c_dev);
free(a);
free(b);
free(c);
```

- `cudaMemcpy` blocks until previous kernel invocations have terminated
  - hence, no need to call `cudaDeviceSynchronize()`
- allocated device memory must be freed, just as allocated host memory must be freed

Makefile for add.cu

```
NAME = add
ROOT = ../../..
include $(ROOT)/common.mk
all: $(NAME).exec
test: $(NAME).exec
        $(CUDARUN) ./$< 200000000
$(NAME).exec: $(NAME).cu
        $(NVCCC) -o $@ $<
.PHONY: all test
```

Makefile for add.cu

```
NAME = add
ROOT = ../../..
include $(ROOT)/common.mk
all: $(NAME).exec
test: $(NAME).exec
        $(CUDARUN) ./$< 200000000
$(NAME).exec: $(NAME).cu
        $(NVCCC) -o $@ $<
.PHONY: all test
```

```
siegel@grendel:~/372/code/src/cuda/add$ make test
srun --unbuffered -n 1 --gres=gpu:1 ./add.exec 200000000
N = 200000000
Host initialization complete.
Device initialization complete.
Result obtained.  Time: 1.673229
```

Example: `diffuse1d.cu`

- CUDA version of `diffuse1d.c`, plain text 1d-diffusion
- the kernel is used to compute the update for a single time step
- multiple blocks and threads per block are used
- each thread is responsible for at most one cell of the array
  - counter-intuitive, but often the easiest and best way in CUDA to break up the work
- number of threads per block: fixed (1024)
- number of blocks: computed based on `nx`
  - just big enough to guarantee there are at least `nx` $-\,2$ threads
- two copies of the temperature array are allocated in device global memory
  - they persist through the entire life of the application
- memory is only copied from host to device after a time step in which a frame will be written

Example: `diffuse1d.cu`

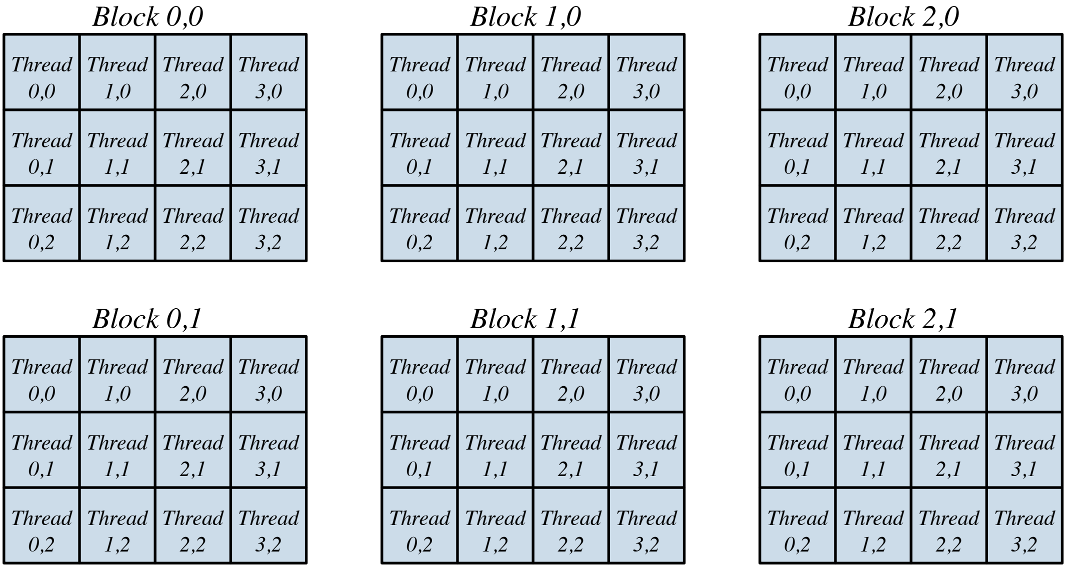
- CUDA version of `diffuse1d.c`, plain text 1d-diffusion
- the kernel is used to compute the update for a single time step
- multiple blocks and threads per block are used
- each thread is responsible for at most one cell of the array
  - counter-intuitive, but often the easiest and best way in CUDA to break up the work
- number of threads per block: fixed (1024)
- number of blocks: computed based on `nx`
  - just big enough to guarantee there are at least `nx` $-\,2$ threads
- two copies of the temperature array are allocated in device global memory
  - they persist through the entire life of the application
- memory is only copied from host to device after a time step in which a frame will be written

Question: why not allow the kernel to do multiple time steps?

1d, 2d, and 3d kernels

- so far, we have dealt only with 1-dimensional grids of 1-dimensional blocks, but...
  - grids can be 1, 2, or 3-dimensional
  - blocks can be 1, 2, or 3-dimensional
- this only affects how we number the blocks and threads
  - a block's ID number is actually an ordered triple (i, j, k)
  - for a 2d-grid, k is always 0
  - for a 1d-grid, j and k are always 0
  - a thread's ID number is also an ordered triple
- nothing else is changed
  - blocks still cannot communicate or coordinate
  - threads within a block are all "equal", communicate through shared variables and synchronize

Example: CUDA 2d grid of 2d blocks



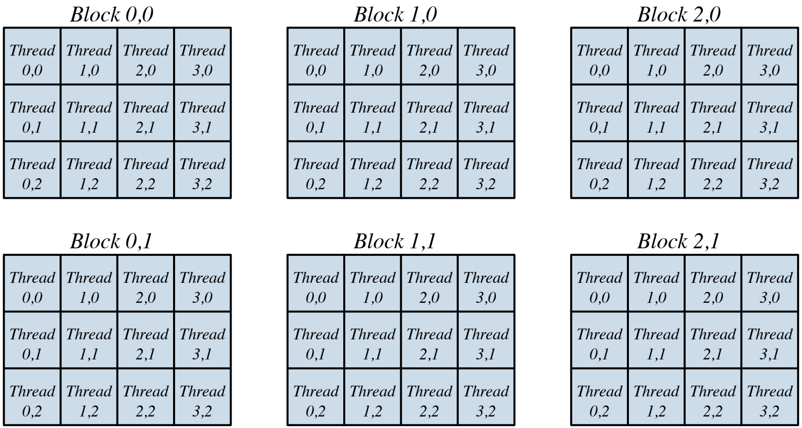
Built-in vector types

- there are vector types corresponding to the basic integer and floating-point types
- `int1`, `int2`, `int3`, `int4`, `uint1`, ...
- `int3` means an ordered triple of integers
- these are C structs
- the fields are `x`, `y`, `z`, and `w`
- `dim3`: type based on `uint3`, an ordered triple of unsigned ints
  - `dim3 threadsPerBlock(N,N);`
  - declares variable `threadsPerBlock` to have type `dim3` and value $(N, N, 1)$
  - an expression of type `dim3` may be used as a configuration parameter in the triple angle brackets when invoking a kernel: `MatAdd<<<numBlocks,threadsPerBlock>>>(A,B,C);`

Built-in variables

- `gridDim`: the dimensions of the grid
  - type: `dim3`, basically a 3-tuple of positive integers
  - the dimensions of the grid tell you how many block there are in the x, y, and z dimensions
- `blockIdx`: block index in grid
  - type: `uint3`
  - tells the x, y, and z coordinates of this block in the grid
- `blockDim`: the dimensions of the block
  - type: `dim3`
  - tells the number of threads in the x, y, and z dimensions
- `threadIdx`: thread index in block
  - type: `uint3`
  - tells the x, y, and z coordinates of this thread in the block

Example: grid and block dimensions



$\texttt{gridDim} = (3, 2, 1)$ $\texttt{blockDim} = (4, 3, 1)$

Kernel invocation

```
kernelName<<<numBlocks, threadsPerBlock>>>( arg1,arg2,...);
```

- *numBlocks*: can be either `int` or `dim3`
  - `int`: the number of blocks arranged in a 1d-grid (in x direction)
  - `dim3`: the number of blocks and their configuration in a grid
  - there is some limit on the dimensions of the grid, but for all practical purposes they are unlimited
- *threadsPerBlock*: ditto, but…
  - the total number of threads in the block is the product $n =$ `blockDim.x * blockDim.y * blockDim.z`
  - n can be at most 1024
  - n should be a multiple of 32

Example of 2d-grid of 2d-blocks: `add2d.cu`

- given n, m, and two $n \times m$ matrices `a` and `b`
- compute matrix sum `c`
- CUDA kernel: each thread gets at most one element
- number of threads per block: fixed at 32*32
- number of blocks in x-direction: $\lceil n/32 \rceil$
- number of blocks in y-direction: $\lceil m/32 \rceil$

add2d.cu: kernel code

```
__global__ void mat_add(int n, int m, double * a, double * b, double * c) {
  int x = blockDim.x*blockIdx.x + threadIdx.x;
  int y = blockDim.y*blockIdx.y + threadIdx.y;

  if (x < n && y < m) {
    const int idx = x*m + y;
    c[idx] = a[idx] + b[idx];
  }
}
```

- each thread computes its (x, y) global coordinate in the grid
  - natural generalization of the "global thread ID" in the 1-dim case
- matrices stored as 1d-arrays, use standard index computation
- each thread must check that it is "in bounds"
  - because n, m are not necessarily exact multiples of 32

add2d.cu: host code, part 1

```
/* Number of threads per block in x and y direction. 32*32=1024 */
const int nthreadsx = 32, nthreadsy = 32;
int main(int argc, char * argv[]) {
  assert(argc == 3);
  int err, n = atoi(argv[1]), m = atoi(argv[2]);
  assert(n>=1 && m>=1);
  const int nblocksx = n/nthreadsx + (n%nthreadsx != 0);
  const int nblocksy = m/nthreadsy + (m%nthreadsy != 0);
  const dim3 blockDim(nthreadsx, nthreadsy), gridDim(nblocksx, nblocksy);
  printf("size=(%d, %d), nblocks=(%d, %d), nthreads=(%d, %d)\n",
         n, m, nblocksx, nblocksy, nthreadsx, nthreadsy);
  fflush(stdout);
  double * a = (double*)malloc(n*m*sizeof(double));  assert(a);
  double * b = (double*)malloc(n*m*sizeof(double));  assert(b);
  double * c = (double*)malloc(n*m*sizeof(double));  assert(c);
  for (int i=0; i<n; i++)
    for (int j=0; j<m; j++) { a[i*m+j] = sin(i*m+j); b[i*m+j] = cos(i*m+j); }
  printf("Host initialization complete.\n");  fflush(stdout);
```

add2d.cu: host code, part 2

```
  mat_add<<<gridDim, blockDim>>>(n, m, a_dev, b_dev, c_dev);
  cudaMemcpy(c, c_dev, n*m*sizeof(double), cudaMemcpyDeviceToHost);
  printf("Result obtained.  Time: %lf\n", mytime() - start_time);
  cudaFree(a_dev);
  cudaFree(b_dev);
  cudaFree(c_dev);
  free(a);
  free(b);
  free(c);
}
```

Makefile for add2d.cu

```
NAME = add2d
ROOT = ../../..
include $(ROOT)/common.mk
all: $(NAME).exec
test: $(NAME).exec
        $(CUDARUN) ./$< 20000 10000
$(NAME).exec: $(NAME).cu Makefile
        $(NVCCC) -o $@ $<
.PHONY: all test
```

Makefile for add2d.cu

```
NAME = add2d
ROOT = ../../..
include $(ROOT)/common.mk
all: $(NAME).exec
test: $(NAME).exec
	$(CUDARUN) ./$< 20000 10000
$(NAME).exec: $(NAME).cu Makefile
	$(NVCCC) -o $@ $<
.PHONY: all test
```

```
siegel@grendel:~/372/code/src/cuda/add2d$ make test
nvcc  -I../../../include  -L../../../lib -o add2d.exec add2d.cu
srun --unbuffered -n 1 --gres=gpu:1 ./add2d.exec 20000 10000
size=(20000, 10000), nblocks=(625, 313), nthreads=(32, 32)
Host initialization complete.
Device initialization complete.
Result obtained.  Time: 1.754350
```