Lecture 8

CUDA (II)

Review

- **Kernel**: computation/program executed by threads.
- **Thread**: many threads run the same kernel to process data of different parts.
- **Thread Block**: Threads are organized into 1D, 2D, or 3D blocks.
- **Thread Grid**: Blocks are organized into 1D or 2D grids.
- Threads within a block can cooperate via **shared memory**, **atomic operations** and **barrier synchronization.**

Examples

- Memory allocations & data movements
- Vector Increment
- Vector scaling
- Matrix addition

Examples

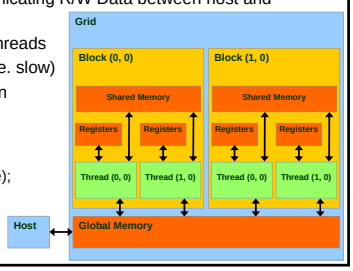
0. Memory Allocation & Initialization
1. Data Movement

00.cu

Memory allocation & initialization

- Before we use GPU to do any computation, we need data on GPU first. Unless, data are generated on GPU.
- **Global memory**
  - Main means of communicating R/W Data between host and device
  - Contents visible to all threads
  - Long latency access (i.e. slow)
  - Content persist between kernel launches

```
void *malloc(size_t size);
void *calloc(size_t nmemb, size_t size);
void free(void *ptr);
void *realloc(void *ptr, size_t size);
```



00.cu

Memory allocation & initialization

```
double host(const unsigned long size) {
 double t1=0;
 int count=0;
 stopWatch timer;

 char *data
 data  = malloc(size);
 do {
  timer.start();
  memset(data, 0, size);

  timer.stop();
  t1 += timer.elapsedTime();
  count++;
 } while(t1<1.0);
 free(data);
 return t1/count;
}
```

void* malloc(size_t size)
allocate dynamic memory

void *memset(void *s, int c, size_t n);
fill memory with a constant byte

void free(void *ptr);
free the memory space pointed by ptr

00.cu

Memory allocation & initialization

```
double host(const unsigned long size) {
  double t1=0;
  int count=0;
  stopWatch timer;

  char *data;
  data  = malloc(size);
  do {
    timer.start();
    memset(data, 0, size);

    timer.stop();
    t1 += timer.elapsedTime();
    count++;
  } while(t1<1.0);
  free(data);
  return t1/count;
}
```

```
double device(const unsigned long size) {
  double t2=0;
  int count=0;
  stopWatch timer;

  char *data;
  cudaMalloc((void**) &data, size);
  do {
    timer.start();
    cudaMemset(data, 0, size);
    cudaThreadSynchronize();
    timer.stop();
    t2 += timer.elapsedTime();
    count++;
  } while(t2 < 1.0);
  cudaFree(data);
  return t2/count;
}
```

7

00.cu

Bandwidth on erasing data

I7 950 3.07GHz + GeFore GT220		Xeon E5310 1.6GHz + GeForce 8800Ultra		Q9550 2.83GHz + GeForce GTX280		I7 950 3.07GHz + NVidia Tesla c2050	
Host	Device	Host	Device	Host	Device	Host	Device
11.04	16.13	2.73	50.27	5.42	66.00	11.05	104.52

Config #0 Config #1 Config #2 Config #3

01.cu

Memory copy between two memory blocks

```
__host__ double memop2(void *dst, void *src, const unsigned long size, cudaMemcpyKind dir) {
    double t = 0;
    int count=0;
    stopWatch timer;
    do {
        timer.start();
        memcpy(dst, src, size);

        timer.stop();
        t += timer.elapsedTime();
        count++;
    } while(t < T_MIN);

    return t/count;
}
```

```
cudaMemcpy(dst, src, size, dir);
cudaThreadSynchronize();
```

```
enum cudaMemcpyKind {
    cudaMemcpyHostToDevice,
    cudaMemcpyDeviceToHost,
    cudaMemcpyDeviceToDevice,
    cudaMemcpyHostToHost
}
```

Example 1. Memory performance

Copy

Config #0

```
Host2Device:0.0440951secs., bandwidth=5.66957GB/sec.
Device2Host:0.0454369secs., bandwidth=5.50213GB/sec.
Device2Devi:0.0244666secs., bandwidth=20.436GB/sec.
Host2Host  :0.0403944secs., bandwidth=12.378GB/sec.
```

Config #1

```
Host2Device:0.0867472secs., bandwidth=2.88194GB/sec.
Device2Host:0.0808961secs., bandwidth=3.09038GB/sec.
Device2Devi:0.046854secs., bandwidth=10.6715GB/sec.
Host2Host  :0.103345secs., bandwidth=4.83814GB/sec.
```

Config #2

```
Host2Device:0.0800946secs., bandwidth=3.12131GB/sec.
Device2Host:0.0800039secs., bandwidth=3.12485GB/sec.
Device2Devi:0.0349145secs., bandwidth=14.3207GB/sec.
Host2Host  :0.0471595secs., bandwidth=10.6023GB/sec.
```

Config #3

```
Host2Device:0.0889612secs., bandwidth=5.62043GB/sec.
Device2Host:0.0822179secs., bandwidth=6.0814GB/sec.
Device2Devi:0.0121208secs., bandwidth=82.5028GB/sec.
Host2Host  :0.0716965secs., bandwidth=13.9477GB/sec.
```

CUDA-C

APIs for memory management

- cudaError_t cudaMalloc (void *devPtr, size_t size)
- cudaError_t cudaMallocHost (void **ptr, size_t size)
  - Allocates size bytes of host memory that is **page-locked/pinned** and accessible to the device.
- cudaError_t cudaMemcpy (void *dst, const void *src, size_t count, enum cudaMemcpyKind kind)
  - cudaMemcpyHostToDevice,
  - cudaMemcpyDeviceToHost,
  - cudaMemcpyDeviceToDevice,
  - cudaMemcpyHostToHost
- cudaError_t cudaFree (void *devPtr)
- cudaError_t cudaMemset (void *devPtr, int value, size_t count)

11

CUDA Reference manual 3.0: Section 4.8: memory management

02.cu

Vector increment

12

02.cu
Vector increment

```
int main() {
  const int n = 10;

  float *a = new float[n];
  for(int i=0;i<n;i++) a[i] = i;

  GPUINC_INT(n, a);
  // CPUINC    (n, a);

  for(int i=0;i<n;i++) cout << a[i] << " ";

  return 0;
}
```

1 2 3 4 5 6 7 8 9 10

13

02.cu
Vector increment

```
void GPUINC_INT(const int n, float *A) {
  float *ptrA;
  const int size = n*sizeof(float);

  cudaMalloc(&ptrA, size);
  cudaMemcpy(ptrA, A, size, cudaMemcpyHostToDevice);

  dim3 blocks(NX);
  dim3 grids( (n+NX-1)/NX );
  kernel <<< grids, blocks >>> (n, ptrA);

  cudaMemcpy(A, ptrA, size, cudaMemcpyDeviceToHost);
  cudaFree(ptrA);
}
```

14

CUDA-C language extension
new data types and function invocation

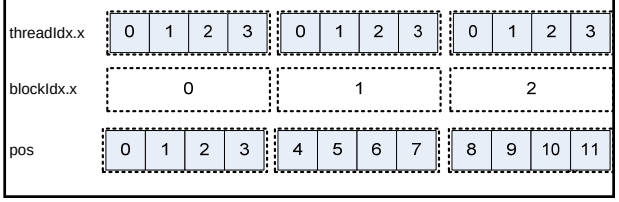
- #define NX 4
- dim3 blocks(NX);
  - Define block size to be **NX * 1 * 1**
- dim3 grids((n+NX-1)/NX);
  - Define grid size to be **(n+NX-1)/NX * 1**
- **kernel <<< grids, blocks >>> (n, ptrA)**;
  - Invoke device function **kernel** with thread configuration of **<<<grids, blocks>>>**
  - The function takes parameters **n, ptrA**

15

02.cu
Vector increment

```
void CPUINC(const int n, float *A) {

  for(int i=0;i<n;i++) {
    A[i]++;
  }
}
```

```
__global__
void kernel(const int n, float *ptrA) {
  const int pos = blockIdx.x*blockDim.x + threadIdx.x;
  if(pos<n) {
    ptrA[pos]++;
  }
}
```



CUDA-C language extension
new declspecs

- **__global__** void kernel(const int n, float *ptrA) { … }
  - Defines a "global" function kernel to run on the device
  - global functions are invoked by host
  - global function must return void
- For functions running on devices
  - No static variables
  - No recursion
  - No variable number of arguments

No function pointers for __device__ functions

	runs on	callable from
__device__ float DeviceFunc()	device	device
__global__ void KernelFunc()	device	host
__host__ float HostFunc()	host	host

__host__ and __device__ can be used together to create two versions of the same function. One running on the host, and one running on the device.

17

CUDA-C language extension
new built-in variables accessible from kernel

- **blockIdx**
  - To identify a specific block within a launched thread grid of 1D or 2D
  - blockIdx.x, blockIdx.y (limit: {65535, 65535})
  - blockIdx.z - unused
- **blockDim**
  - To find the dimensions of launched thread block of 1D, 2D, or 3D
  - blockDim.x, blockDim.y, blockDim.z (limit: {512, 512, 64})
- **threadIdx**
  - To identify a specific thread within a block of 1D, 2D, or 3D
  - threadIdx.x, threadIdx.y, threadIdx.z (limit: {512, 512, 64})

18

04?.cu

Vector scaling
(1, 2, 3) * 2 → (2,4,6)

19

Example: vector scaling

Vanilla C code vs. CUDA Kernel function

```
void dscal    (const float scale, const int n, float *v)
{
  int i;
  for(i=0;i<n;i++) {
    v[i] *= scale;
  }
}
```

```
#define THREAD_PER_BLOCK 512

__global__
void dscal_dev(const float scale, const int n, float *v)
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if(i < n) v[i] *= scale;
}
```

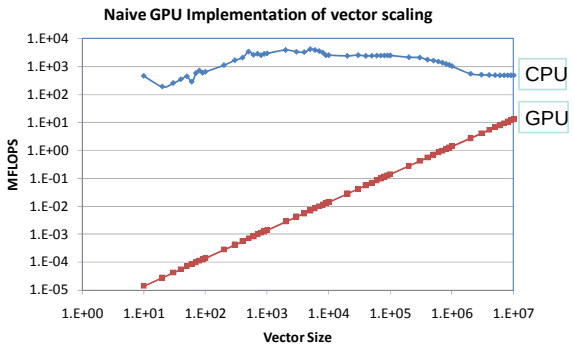
Example: vector scaling

CUDA code - interface

```
void dscal_interface(const float scale, const int n, float *v) {
  float *v_d;
  const int bytes = n * sizeof(float);

1 cudaMalloc((void**) &v_d, bytes);
  cudaMemcpy((void*)v_d,(void*)v,bytes,cudaMemcpyHostToDevice);

  dim3 blocks(THREAD_PER_BLOCK);
2 dim3 grids ((n+THREAD_PER_BLOCK-1) / THREAD_PER_BLOCK);
  dscal_device <<<grids, blocks>>> (scale, n, v_d);

3 cudaMemcpy((void*)v,(void*) v_d,bytes,cudaMemcpyDeviceToHost);
  cudaFree(v_d);
}
```

1. Allocate memory on device and copy data from host memory to device memory.
2. Divide work into a) grids of blocks and b) blocks of threads, and then execute the kern
3. Copy computed results from device memory to host memory.

Preliminary Results

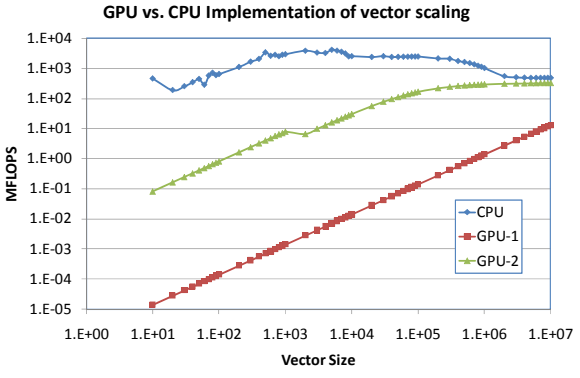


Example: vector scaling

Optimization #1 – use pinned memory

```
float *vec;
vec = (float*) malloc(sizeof(float) * size);
assert(vec);

for(i=0;i<size;i++) {
  vec[i] = 1.0f;
}
. . .
free(vec);
```

```
float *vec;
cudaMallocHost((void**) &vec, sizeof(float)*size);
assert(vec);

for(i=0;i<size;i++) {
  vec[i] = 1.0f;
}
. . .
cudaFree(vec);
```

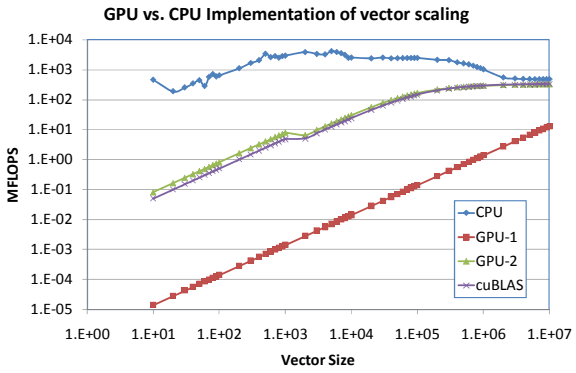


Example: vector scaling

Optimization #2? – use cuBLAS

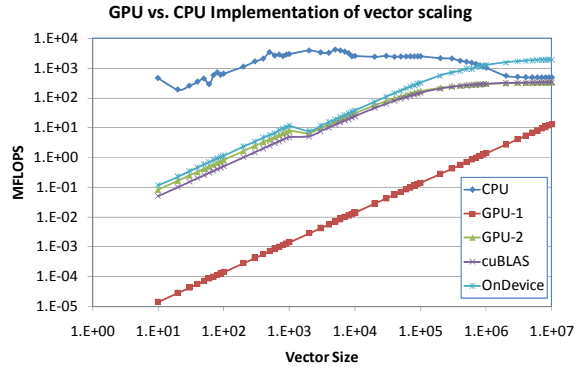
```
void dscal_cublas(const float scale,const int size,float *vec)
{
  cublasStatus stat;
  float *vec_d;
  cublasInit();

  stat = cublasAlloc(size, sizeof(float), (void**)&vec_d);
  cublasSetVector(size, sizeof(float), vec, 1, vec_d, 1);

  cublasSscal(size, scale, vec_d, 1);

  cublasGetVector(size, sizeof(float), vec_d, 1, vec, 1);
  cublasFree(vec_d);
}
```

BLAS stands for Basic Linear Algebra Subprograms: a library specification for performing vector, matrix-vector, matrix-matrix operations.



Where is the performance of GPU?

```
__global__
void dscal_device(const float scale, const int n, float *v) {
  int where = blockIdx.x * blockIdx.x + threadIdx.x;
  if(where < n) {
    v[where] = 1.0f;
    v[where] *= scale;
  }
}

void dscal_interface(const float scale, const int n, float *v) {
  float *v_d;

  cudaMalloc((void**) &v_d, n*sizeof(float));

  dim3 blocks(THREAD_PER_BLOCK);
  dim3 grids ((n+THREAD_PER_BLOCK-1) / THREAD_PER_BLOCK);
  dscal_device<<<grids, blocks>>> (scale, n, v_d);

  cudaMemcpyDeviceToHost);
  cudaFree(v_d);
}
```



What have we learned so far?

- GPGPU using CUDA is easy if you know C/C++.
  - Loops are replaced by “configuration” of threads
- Copying data between host and device kills performance
- Use of pinned (page-locked) memory boosts performance
  - Data in page-locked memory are transferred using DMA
- Do everything on device whenever possible
- Need to compute lots of data to show performance.

- GPU programs do NOT necessarily run faster than CPU.

03?.cu

Matrix addition

30

03a.cu (CPU version)

```
void CPUADD(const int n, DT *C, DT *A, DT *B) {
    for(int i=0;i<n;i++) {
        for(int j=0;j<n;j++) {
            int where = i * n + j;
            C[where] = A[where] + B[where];
        }
    }
}
```

31

03b.cu

```
#define DT float
int main(int argc, char **argv) {
    const int n = 10;
    DT *A = genMatrix(n);
    DT *B = genMatrix(n);
    DT *C;
    cudaMallocHost(&C, n*n*sizeof(DT));

    GPUADD_INT(n, C, A, B);

    cudaFree(A);
    cudaFree(B);
    cudaFree(C);
    return 0;
}
```

```
DT *genMatrix(const int n) {
  DT *ptr;
  cudaMallocHost(&ptr, n*n*sizeof(DT));
  for(int i=0;i<n*n;i++) {
      ptr[i]=((double)rand())/RAND_MAX;
  }
  return ptr;
}
```

03b.cu

```
__global__
void matrixAdd(
    const int n, DT *ptrC, DT *ptrA, DT *ptrB) {

    const int i = blockIdx.y * blockDim.y + threadIdx.y;
    const int j = blockIdx.x * blockDim.x + threadIdx.x;

    if(i<n && j<n) {
        const int where = i*n+j;
        ptrC[where] = ptrA[where] + ptrB[where];
    }
}
```

33

03b.cu

```
void GPUADD_INT(const int n, DT *C, DT *A, DT *B) {
    DT *ptrA, *ptrB, *ptrC;
    const int size = sizeof(DT)*n*n;
    cudaMalloc(&ptrA, size);
    cudaMalloc(&ptrB, size);
    cudaMalloc(&ptrC, size);
    cudaMemcpy(ptrA, A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(ptrB, B, size, cudaMemcpyHostToDevice);
    dim3 blocks(UNIT_X, UNIT_Y);
    dim3 grids((n+UNIT_X-1)/UNIT_X,(n+UNIT_Y-1)/UNIT_Y));
    matrixAdd <<< grids, blocks >>> (n, ptrC, ptrA, ptrB);
    cudaMemcpy(C, ptrC, size, cudaMemcpyDeviceToHost);
    cudaFree(ptrA);
    cudaFree(ptrB);
    cudaFree(ptrC);
}
```

34

03c.cu

Matrix addition with 2D memory allocation

35

CUDA-C

APIs for memory management

- cudaError_t cudaMallocPitch (void **devPtr, size_t *pitch, size_t width, size_t height)
  - *Allocates at least width in Bytes & height bytes of linear memory on the device and returns in devPtr a pointer to the allocated memory. The function may* **pad** *the allocation to ensure that corresponding pointers in any given row will continue to meet the* ***alignment requirements for coalescing*** *as the address is updated from row to row. The pitch returned in pitch by cudaMallocPitch() is the* ***width in bytes*** *of the allocation*
  - *T* pElement = (T*)((char*)BaseAddress + Row * pitch) + Column;*
  - Pitch is similar to leading dimensions in BLAS discussed.
- cudaError_t cudaMemcpy2D (void *devPtr, size_t dpitch, const void *src, size_t spitch, size_t width, size_t height, enum cudaMemcpyKind kind)

36

CUDA Reference manual: Section 4.8: memory management

03c.cu

Matrix addition with 2D layout

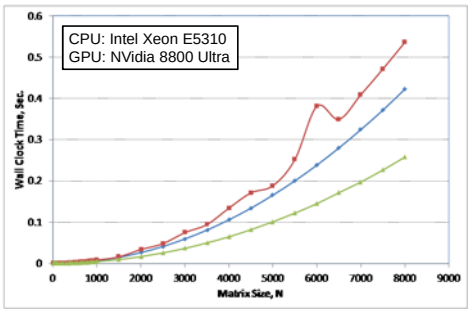
```
__global__ void matrixAdd( const int n, const int pitch,
        DT *ptrC, DT *ptrA, DT *ptrB)  {

    const int i = blockIdx.y * blockDim.y + threadIdx.y;
    const int j = blockIdx.x * blockDim.x + threadIdx.x;

    if(i<n && j<n) {
            DT *C = (DT*) ((char*)ptrC + i*pitch) + j;
            DT *A = (DT*) ((char*)ptrA + i*pitch) + j;
            DT *B = (DT*) ((char*)ptrB + i*pitch) + j;
            C[0] = A[0] + B[0];
    }
}
```

37

```
void GPUADD_INT2(const int n, DT *C, DT *A, DT *B) {
DT *ptrA, *ptrB, *ptrC;
size_t pitch;
size_t size = n * sizeof(DT);

cudaMallocPitch((void**) &ptrA, &pitch, size, n) ;
cudaMallocPitch((void**) &ptrB, &pitch, size, n) ;
cudaMallocPitch((void**) &ptrC, &pitch, size, n) ;
cudaMemcpy2D(ptrA,pitch,A,size,size,n,cudaMemcpyHostToDevice);
cudaMemcpy2D(ptrB,pitch,B,size,size,n,cudaMemcpyHostToDevice);

dim3 blocks(UNIT_X, UNIT_Y);
dim3 grids((n+UNIT_X-1)/UNIT_X,(n+UNIT_Y-1)/UNIT_Y);
matrixAdd <<< grids, blocks >>> (n, pitch, ptrC, ptrA, ptrB);

cudaMemcpy2D(C,size,ptrC,pitch,size,n,cudaMemcpyDeviceToHost);
cudaFree(ptrA);
cudaFree(ptrB);
cudaFree(ptrC);
}
```

38

Performance



03b

03a

03c

39

Lesson Learned

- When dealing with 2D data (matrices, 2D arrays), use 2D memory allocation to ensure proper alignment of data.
- cudaMallocPitch, cudaMemcpy2D

40

Assignment #5

Due: 5/15/2012

Assignment #6

This program will continue your saga to find collision pairs of spheres. You are to write programs to:

1) Get a user-specified N, D, Seed from command line arguments.
2) On host, specify random seed (Seed) using srand(), and then generate N random spheres (3 coordinate components + radius), centers of these spheres should be in the region of [-D:D, -D:D, -D:D] and each sphere is numbered incrementally starting from 0. (of course, you need to use dynamic arrays to store all the data!)
3) Write a global function that takes arguments 1) N, the number of spheres, and 2) data for spheres. (i.e. x, y, z, r). This function will then launch a kernel to count the number of pairs of spheres that collides.
   - **There are several challenges on writing this function (to be discussed in the next lecture!) Better start working on this one early!**

Discuss

1. Discuss the efficiency of storing N sphere data using various data layout strategies. Which of the following gives the best performance?
   1. Option 1: double xyz[N][3], r[N];
   2. Option 2: double xyzr[N][4];
   3. Option 3: double x[N], y[N], z[N], r[N];
   4. You may also explorer other possibilities (e.g. using struct & class)
2. Is CUDA version faster or slower than the CPU version? How much faster/slower is your CUDA version than the pure CPU version?

Note

- All discussions need to consider different program sizes (e.g. N=10, 50, 100, 200, 500, 1000, 2000, 5000, 10000, …)
- All discussions should be facilitated by charts, and then tell stories from these charts.
- All your work must reside in folder HW06 in your home directory in our cluster system (ssh into 140.118.5.6:222)
- All your programs must be made by simply typing make in HW06 folder (i.e. you must write a Makefile)
- For evaluating performance or efficiency, use the timing class that is provided to you:
  - /home/courses/stopWatch.h, /home/courses/stopWatch.o