Lecture 9

CUDA (I)

1

CUDA

Compute **U**nified **D**evice **A**rchitecture

2

Outline

- CUDA Architecture
- CUDA programming model
- CUDA-C

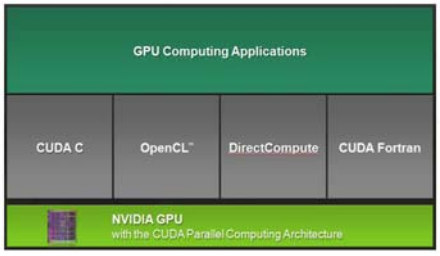
3

CUDA Architecture

4

CUDA™: a General-Purpose Parallel Computing Architecture

- CUDA is designed to support various languages or APIs

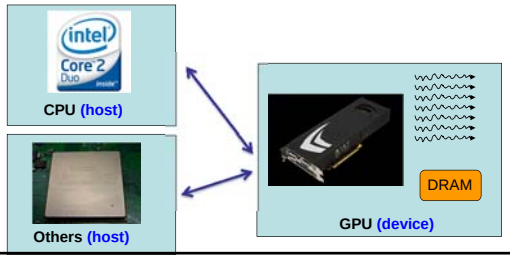


5

NVIDIA_CUDA_Programming_Guide_3.0

CUDA Device and Threads

- Host: usually the general purpose CPU running the computer system
- Host memory: DRAM on the host
- Initiates the execution of kernels
- Device: coprocessor to the CPU/Host
- Device memory: memory on the device
- Runs many threads in parallel

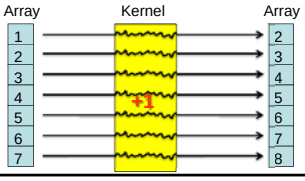


CUDA Programming Model

7

Programming Concepts

- CUDA considers “many-core” processors
- Need **lots and lots of threads** to make it efficient!
- Kernel
  - Kernels are executed by devices through **threads**. One kernel may be executed by many threads.
- Thread
  - Executes a kernel
  - **threadIdx**



Single thread vs. multiple threads

```
Single thread + loop
int a[100]={0};
for(int i=0;i<100;i++) {
    a[i]++;
}
```

9

Single thread vs. multiple threads

```
100 threads, each thread has an
Id starts from 0
a[id]++;
```

10

Thread Block

- Each thread has its unique ID within a block
- One thread block contains many threads arranged in 1D, 2D, or 3D
  - Process data in linear array, in matrices, or in volumes
  - Threads get their id by threadIdx.x, threadIdx.y, threadIdx.z
- Holds up to 512 threads in current GPU
  - The limit exists because threads in a block are expected to execute on the same processor core
  - Threads in the same block can cooperate
- Threads within a block can cooperate via **shared memory**, **atomic operations** and **barrier synchronization**

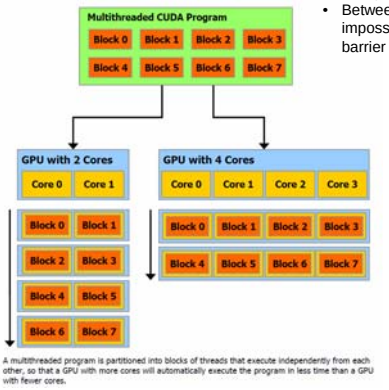
11

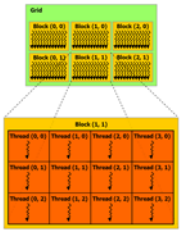
Grid

- Thread blocks are arranged into grid (1D or 2D)
- So CUDA programs have grid, each grid point has a thread block, and each thread block contains threads arranged in 1D, 2D, or 3D
- Dimensional Limits:
  - Grid: 65535 x 65536
  - Thread block : 512 x 512 x 64 (the total number must be < 512)
- NO cooperation between thread blocks
- This grid → blocks → threads alllows CUDA programs scale on different hardware.
- Programmers can use **blockIdx, blockDim,** and **threadIdx** to identify their global location/ranking/id to deduce its portion of work (similar to rank in MPI)
  - blockIdx.x, blockIdx.y, blockDim.x, blockDim.y, blockDim.z
  - threadIdx.x, threadIdx.y, threadIdx.z

12

- Thread blocks are executed in any order as scheduler see fits.
- Between thread blocks, it is impossible to cooperate beyond barrier synchronization issued by host.



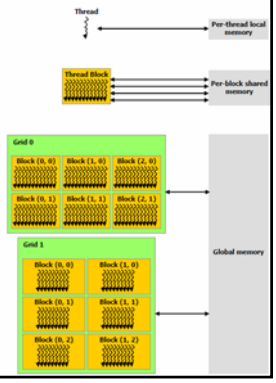


13

NVIDIA_CUDA_Programming_Guide_3.0

Programming Concepts

- Memory Hierarchy
  - Local memory
  - Shared memory
  - Global memory (R/W)
    - Constant memory (R)
    - Texture memory (R)
- Global memories can be accessed by both device & host, but they are optimized for different purposes..
- Need to choose appropriate memory types to use. But for now, just keep this in mind.



Refresh

- Kernel
- Thread
- Thread Block
- Grid
- Memory
  - Local, shared, global / constant / texture

15

CUDA-C

16

Examples

0. Memory Allocation & Initialization
1. Data Movement
2. Vector Increment
3. Matrix Addition

17

CUDA-C

- CUDA-C extends C/C++
  - Declspecs (declaration specifications):
    - Functions declaration: global, device, host
    - Variable declaration: shared, local, constant
  - Built-in variables
    - threadIdx, blockIdx, blockDim
  - Intrinsics (intrinsic function)
    - __synthreads, cudaThreadSynchronize(), …
  - Runtime API
    - Memory, execution management, etc.
  - Launching kernels

intrinsic function: functions handled by compilers instead of library functions. (http://en.wikipedia.org/wiki/Intrinsic_function /)

18

Examples

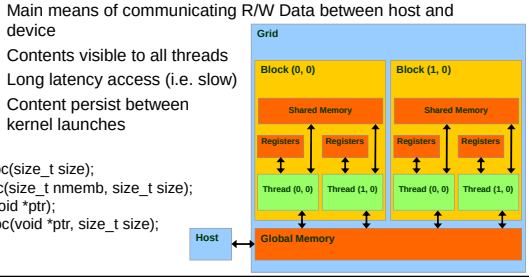
0. Memory Allocation & Initialization
1. Data Movement

19

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

21

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;
}
```

22

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)

26

CUDA Reference manual 3.0: Section 4.8: memory management

02.cu

Vector increment

27

02.cu

Vector increment

1 2 3 4 5 6 7 8 9 10

```
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;
}
```

28

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);
}
```

29

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**

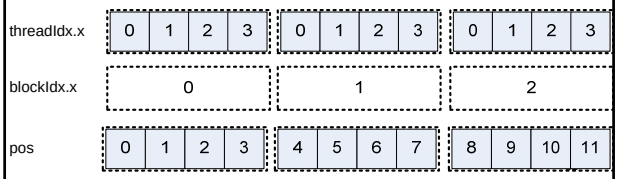
30

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.

32

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})

33

03?.cu

Matrix addition

34

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];
    }
}
```

36

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);
}
```

37

03c.cu

Matrix addition with 2D memory allocation

38

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)

39

CUDA Reference manual: Section 4.8: memory management

03c.cu

Matrix addition with 2D layout

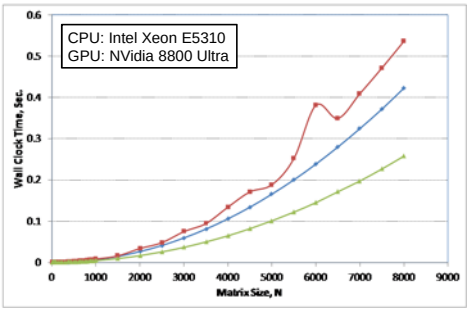
```
__global__ void matrixAdd( const int n, const int pitch,
        DT *ptrC, DT *ptrA, DT *ptrB)  {

    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];
    }
}
```

40

```
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);
}
```

41

Performance



42

Lesson Learned

- When dealing 2D data, use 2D memory allocation to ensure proper alignment of data.
- cudaMallocPitch, cudaMemcpy2D
- The pitch parameter in CUDA API is similar to leading dimensions in BLAS, except that pitch uses “bytes” while leading dimensions use # of elements as its unit.

Readings

- No programming assignment this week!
  - Serial implementation of the term project due on May. 10, 2011
- NVIDIA CUDA Programing Guide (4.0RC2)
  - Chap 1, 2 (p.1 – 14)
  - Most of Section 3.2 (p. 18 – 46)
  - Chap 4 (p. 85-88), optional
  - Chap 5 (p. 89 – 102)
- Quickly browse Appendix B, C (p.107 – 144)

http://140.118.5.6:8000/doc/CUDA/4.0RC2/CUDA_C_Programming_Guide.pdf