CSC266 Introduction to Parallel Computing using GPUs

GPU Architecture II (Memory)

Sreepathi Pai
November 1, 2017

URCS
Outline

Basics

Global Memory

Shared Memory

L1, L2, Texture, and Constant Caches
Outline

Basics

Global Memory

Shared Memory

L1, L2, Texture, and Constant Caches
Memories

- Where data can be stored
- Two main, real spaces:
  - Off-chip: “Global” or “device” memory
  - On-chip: “Shared” memory
- Lots of “memory spaces” built on top of these two:
  - where data is stored (i.e., global or shared)
  - how data is accessed (i.e., which cache)
  - to whom data is visible (e.g., all threads or only thread block)
Outline

Basics

Global Memory

Shared Memory

L1, L2, Texture, and Constant Caches
Global Memory

- GPU RAM, usually tens of gigabytes
- High-bandwidth access from GPU
  - 300 GB/s to 1TB/s (depending on the GPU)
- Memory allocated using `cudaMalloc` (and equivalents)
- Contains following memory spaces:
  - Global memory space
  - Local memory space
  - Texture memory space
Memory spaces in Global Memory

- **Global Memory Space**
  - Used for all GPU data accessed through pointers
  - Or for variables marked `__device__`
  - Data is accessed through L2 cache
  - Data is visible to all threads

- **Local Memory space**
  - Automatically used for *thread-local* data
  - E.g., register spills by compiler
  - Data is visible only to single thread
  - Data is accessed through L1/L2 cache

- **Texture Memory Space**
  - Used for *read-only texture* data
  - Data visible to all threads
  - Data accessed through texture unit (or separate cache)
  - More later today ...
struct pt {
    int x;
    int y;
};

__global__
void aos_kernel(int n_pts, struct pt *p) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    int nthreads = blockDim.x * gridDim.x;

    for(int i = tid; i < n_pts; i += nthreads) {
        p[i].x = i;
        p[i].y = i * 10;
    }
}

In main():

struct pt *p;
cudaMalloc(&p, ...)

struct pt {
    int *x;
    int *y;
};

__global__
void soa_kernel(int n_pts, struct pt p) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    int nthreads = blockDim.x * gridDim.x;

    for(int i = tid; i < n_pts; i += nthreads) {
        p.x[i] = i;
        p.y[i] = i * 10;
    }
}

In main():

struct pt p;
cudaMalloc(&p.x, ...)
cudaMalloc(&p.y, ...)
AoS vs SoA for GPU programs

- Array of Structures
- Structure of Arrays
- Which is better for CPU?
- Which is better for GPU?
- $p[i].x$ memory bandwidth utilization?
SoA memory layout

- p.x[i] memory bandwidth utilization?
AoS vs SoA Performance

![Graph showing performance comparison between AoS and SoA](image-url)
AoS vs SoA: Number of Memory Transactions

![Graph showing the comparison between AoS and SoA in terms of the number of memory transactions.](image)
Assigning Work to Threads

Blocked:

\[
\begin{align*}
\text{start} &= \text{tid} \times \text{blksize}; \\
\text{end} &= \text{start} + \text{blksize}; \\
\text{for}(i = \text{start}; i < N \land i < \text{end}; i++) \\
& \quad \text{a}[i] = \text{b}[i] + \text{c}[i]
\end{align*}
\]

Interleaved:

\[
\begin{align*}
\text{start} &= \text{tid}; \\
\text{for}(i = \text{start}; i < N; i+=\text{nthreads}) \\
& \quad \text{a}[i] = \text{b}[i] + \text{c}[i]
\end{align*}
\]

Which, if any, is faster?
Blocking vs Interleaved

![Graph showing the comparison between Blocked and Striped methods over vector size and time.]
Global memory accessed in 32-byte chunks on current GPUs

Ideally, each warp accesses four 32-byte chunks

- Bandwidth underutilized otherwise
- L2 cache is unable to exploit temporal locality

Hence, for GPUs, prefer:

- SoA over AoS
- Assign work in interleaved vs blocked (if this affects memory access pattern)
Outline

Basics

Global Memory

Shared Memory

L1, L2, Texture, and Constant Caches
“Shared Memory” is on-chip software-managed cache, also known as a scratchpad

- 48K maximum size/thread block
  - GPU can have up to 128K

- Partitioned among thread blocks

- `__shared__` qualifier places variables in shared memory

- Can be used for communicating between threads of the same thread block

```c
__shared__ int x;
if(threadIdx.x == 0)
    x = 1;
__syncthreads(); // required!
printf("%d\n", x);
```
Using shared memory for communication

- (Contrived) Find maximum number in each 256 element chunk of large array

```c
__global__ void chunkmax(int *a, int N, int *out) {
  int tid = threadIdx.x + blockIdx.x * blockDim.x
  ...
}
main() {
  cudaMallocManaged(&a, N*sizeof(int));
  // fillup a
  blocks = (N + 255) / 256;
  cudaMallocManaged(&out, blocks*sizeof(int))
  chunkmax<<<blocks, 256>>>(a, N, out);
  cudaDeviceSynchronize();
  // print out
}
```
for(int block = 0; block < blocks; block++) {
    blockmax = a[block*256];
    for(int elem = block * 256;
        elem < (block + 1) * 256 && elem < N;
        elem++)
    {
        if(a[elem] > blockmax)
            blockmax = a[elem];
    }
    out[block] = blockmax;
}
• (Contrived) Find maximum number in each 256 element chunk of large array

```c
__global__ void chunkmax(int *a, int N, int *out) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x

    __shared__ int tbmax;

    if (tid < N)
        return;

    if(threadIdx.x == 0)
        tbmax = a[tid]

    __syncthreads()

    if(a[tid] > tbmax)
        tbmax = a[tid]

    __syncthreads()

    if(threadIdx.x == 0)
        out[blockIdx.x] = tbmax
}
```
__global__ void chunkmax(int *a, int N, int *out) {

__shared__ int tbmax;

...

if(threadIdx.x == 0)
  tbmax = a[tid]

__syncthreads()
int mycopy;

do {
  mycopy = tbmax;
  __syncthreads();

  if (a[tid] > mycopy)
    tbmax = a[tid];
  __syncthreads();
} while(a[tid] > tbmax);

if(threadIdx.x == 0)
  out[blockIdx.x] = tbmax
}
__global__ void chunkmax(int *a, int N, int *out) {
  ...

  if(threadIdx.x == 0)
    tbmax = a[tid]

  __syncthreads()
  int mycopy;

  do {
    mycopy = tbmax;
    __syncthreads();

    if (a[tid] > mycopy)
      tbmax = a[tid];
    __syncthreads();
  } while(mycopy != tbmax);

  if(threadIdx.x == 0)
    out[blockIdx.x] = tbmax
}
Using Shared Memory as a cache (SGEMM)

```c
__shared__ float c_sub[BLOCKSIZE][BLOCKSIZE];

// calculate c_sub
__syncthreads();

// write out c_sub to memory
```

- Read Section 3.2.3 in CUDA C Programming Guide
Outline

Basics

Global Memory

Shared Memory

L1, L2, Texture, and Constant Caches
Exploiting Spatial Locality: Texture Caches

- Textures are 2-D images that are “wrapped” around 3-D models
- Exhibit 2-D locality, so textures have a separate cache
- GPU contains a texture fetch unit that non-graphics programs can also use
  - Step 1: map arrays to textures
  - Step 2: replace array reads by tex1Dfetch(), tex2Dfetch()
- Catch: Only read-only data can be cached
  - you can write to the array, but it may not become visible through the texture in the same kernel call
  - i.e. texture caches are not coherent with GPU memory
- Easiest way to use textures:
  - const __restrict__ *
  - Compiler will automatically use texture cache for marked arrays
Constant Data Cache

- 64KB of "constant" data
  - not written by kernel
- Suitable for read-only, "broadcast" data
- All threads in a warp read the same constant data item at the same time
  - what type of locality is this?
- Uses: Filter coefficients
  - 2dconv: convolution matrix entries
- Lab 7 uses __constant__ for image properties
Summary of Memory Performance

- Layout data structures in memory to maximize bandwidth utilization
- Assign work to threads to maximize bandwidth utilization
- Rethink caching strategies
  - identify readonly data
  - identify blocks that you can load into shared memory
  - identify tables of constants