CUDA Programming
Shared memory, divergence, synchronisation

Oguz Kaya

Maître de Conférences
Université Paris-Saclay et l'Équipe ParSys du LISN, Orsay, France

Objectives

- Re-zoom on GPU memory architecture
- Using “shared-memory” for fast repetitive memory accesses
- Synchronization of threads
- Warp divergence concept for efficient branch management

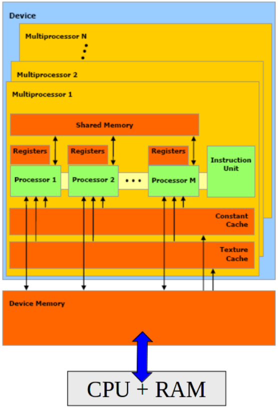
Outline

Outline

1. GPU memory architecture
2. Using shared memory
3. Divergence

GPU memory architecture

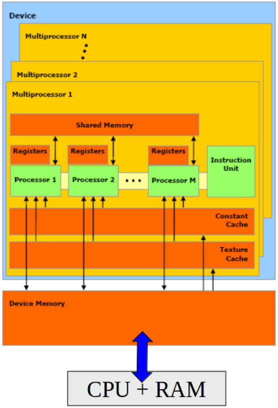
There are many memory types within a GPU.



- Global memory (RAM): 8-48GB, 500-2000 GB/s, 300-400 cycles per access
- Registers: 65536/SM, immediate access (1 cycle)
- L1 cache: 64-192KB/SM, $\approx$ 4 cycles per access
- L2 cache: 8-40MB, latency/bandwidth worse than L1
- Constant cache: Constant memory per SM

GPU memory architecture

There are many memory types within a GPU.



- GPU shared memory:
  - Shared by all threads within **the same block**
  - 32-128KB/SM
  - Same as L1 cache technology
  - $\approx$ 4 cycle access latency
  - Bandwidth comparable to registeres if latency can be hidden
  - No need for coalescence (yet bank conflicts might be a problem, coming later)
  - Using shared memory might reduce L1 capacity by the same amount (i.e., unified L1+shared memory)

Outline

Example: Power computation on an array

Given an array `A[N]` and an integer `k`, replace `A[i]` with `A[i]`^{k}.

```
#include <cstdio>
#include "cuda.h"

#define N 1024
#define BLOCKSIZE 128
float A[N];

__device__ float dA[N];

__global__ void powerArray(int n, int k)
{
  int i = threadIdx.x + blockIdx.x * blockDim.x;
  if (i < n) {
    float c = 1.0;
    for (int j = 0; j < k; j++) { c *= dA[i]; }
    dA[i] = c;
  }
}

int main(int argc, char **argv)
{
  // Initialisation
  for (int i = 0; i < N; i++) { A[i] = i; }
  // Copier le tableau vers le GPU
  cudaMemcpyToSymbol(dA, A, N * sizeof(float), 0,
      cudaMemcpyHostToDevice);
  int blockSize = 128;
  int numBlocks = N / blockSize;
  if (N % blockSize) numBlocks++;
  powerArray<<<numBlocks, blockSize>>>(N, 4);
  // Recopier le tableau vers le CPU
  cudaMemcpyFromSymbol(A, dA, N * sizeof(float), 0,
      cudaMemcpyDeviceToHost);
  printf("%lf\n", A[2]);
  return 0;
}
```

- Blocks/threads 1D
- Each thread updates 1 element with k multiplications
- $dA[i]$ is accessed k times. Can we do better?
  - Put $dA[i]$ in a register (i.e., `float temp = dA[i];`)
  - Use shared memory

Example: Power computation on an array

Given an array `A[N]` and an integer `k`, replace `A[i]` with `A[i]`^{k}.

```
#include <cstdio>
#include "cuda.h"

#define N 1024
#define BLOCKSIZE 128
float A[N];

__device__ float dA[N];

__global__ void powerArray(int n, int k)
{
  int i = threadIdx.x + blockIdx.x * blockDim.x;
  // BLOCKSIZE == blockDim.x
  __shared__ float data[BLOCKSIZE];
  if (i < n) {
    data[threadIdx.x] = dA[i];
    float c = 1.0;
    for (int j = 0; j < k; j++) {
      c *= data[threadIdx.x];
    }
    dA[i] = c;
  }
}

int main(int argc, char **argv)
{
  /// ...
  powerArray<<<numBlocks, blockSize>>>(N, 4);
  /// ...
  printf("%lf\n", A[2]);
  return 0;
}
```

- Add prefix **__shared__** to define an array in shared memory
- Size must be a **constante** known at compile time (not possible to use **blockDim.x** for instance)
- `dA[i]` is accessed **1 time** then reused **k times** in the shared memory
- Shared array is freed when the block terminates, no need to **deallocate**.
- Use **__syncthreads()** to wait for all threads in the block, before using loaded elements.

Outline

Branching in a GPU kernel

Threads are executed by groupes of 32 in a **warp**.

- All threads in the warp execute the same instruction simultaneously
- If there is a **branching**: `if (cond) f(); else g();`
  - If all threads in warp satisfy **cond**, they only execute `f()` simultaneously.
  - If all threads in warp fail **cond**, they only execute `g()` simultaneously.
  - If there is **at least one thread** that satisfies **cond** and **at least one thread** that fails **cond**
    - First, `f()` is executed by those who satisfy **cond**, the rest idles
    - Then, `g()` is executed by those that fail **cond**, the rest idles
    - Total warp execution time is therefore the sum of two regions

Example: Two types of divergences

```
__device__ void kernel()
{
  // ...
  // Divergence 1
  if (threadIdx.x % 2 == 0) {
    f();
  } else {
    g();
  }
  // ...
  // Divergence 2
  if (threadIdx.x < SIZE / 2) {
    f();
  } else {
    g();
  }
  // ...
}
```

- Suppose $f()$ and $g()$ take F and G seconds respectively when executed by a single thread.
- Divergence 1 concerns **all warps** in the block, hence the total execution time of each warp will be $F + G$.
- Divergence 2 concerns **only one warp** in the block, other warps will take either F or G seconds in the execution.

Example: Two types of divergences

```
__device__ void kernel()
{
  // ...
  // Divergence 1
  if (threadIdx.x % 2 == 0) {
    f();
  } else {
    g();
  }
  // ...
  // Divergence 2
  if (threadIdx.x < SIZE / 2) {
    f();
  } else {
    g();
  }
  // ...
}
```

- For good performance, we need to
  - either avoid branching as much as possible
  - or make it so that few warps diverge within a block

Contact

Oguz Kaya
Université Paris-Saclay and LRI, Paris, France
oguz.kaya@lri.com
www.oguzkaya.com