P&S Heterogeneous Systems

GPU Memory Hierarchy

Dr. Juan Gómez Luna
Prof. Onur Mutlu
ETH Zürich
Spring 2023
31 March 2023
GPU Programming
Recommended Readings (II)

Traditional Program Structure

- CPU threads and GPU kernels
  - Sequential or modestly parallel sections on CPU
  - Massively parallel sections on GPU

Serial Code (host)

Parallel Kernel (device)

KernelA<<< nBlk, nThr >>>(args);

Serial Code (host)

Parallel Kernel (device)

KernelB<<< nBlk, nThr >>>(args);

Slide credit: Hwu & Kirk
void vecadd(float* A, float* B, float* C, int N) {

    // Allocate GPU memory
    float *A_d, *B_d, *C_d;
    cudaMalloc((void**) &A_d, N*sizeof(float));
    cudaMalloc((void**) &B_d, N*sizeof(float));
    cudaMalloc((void**) &C_d, N*sizeof(float));

    // Copy data to GPU memory
    cudaMemcpy(A_d, A, N*sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(B_d, B, N*sizeof(float), cudaMemcpyHostToDevice);

    // Perform computation on GPU
    const unsigned int numThreadsPerBlock = 512;
    const unsigned int numBlocks = N/numThreadsPerBlock;

    vecadd_kernel<<<numBlocks, numThreadsPerBlock>>>(A_d, B_d, C_d, N);
    // Copy data from GPU memory
    cudaMemcpy(C, C_d, N*sizeof(float), cudaMemcpyDeviceToHost);

    // Deallocate GPU memory
    cudaFree(A_d);
    cudaFree(B_d);
    cudaFree(C_d);
}
NVIDIA H100: Thread Block Clusters

- GPUs grow beyond 100 GPU cores (SMs): a new level in the software hierarchy can improve execution efficiency
  - Programmatic control of locality at a granularity larger than a single thread block on a single SM
- Thread blocks in the same cluster can synchronize and exchange data
- Thread blocks in the same cluster are guaranteed to be concurrently scheduled
  - Thread blocks in the same cluster run on the SMs within a GPU Processing Cluster (GPC)
  - Data sharing via SM-to-SM network in a GPC

https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/
GPU Memories
NVIDIA H100 Block Diagram

144 cores on the full GH100
60MB L2 cache

https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/
Memory in the GPU Architecture

Slide credit: Izzat El Hajj
Memory in the H100 GPU Architecture

Slide credit: Izzat El Hajj
CUDA Variable Type Qualifiers

<table>
<thead>
<tr>
<th>Variable declaration</th>
<th>Memory</th>
<th>Scope</th>
<th>Lifetime</th>
</tr>
</thead>
<tbody>
<tr>
<td>int LocalVar;</td>
<td>register</td>
<td>thread</td>
<td>thread</td>
</tr>
<tr>
<td>int localArr[N];</td>
<td>global</td>
<td>thread</td>
<td>thread</td>
</tr>
<tr>
<td><strong>device</strong> <strong>shared</strong> int SharedVar;</td>
<td>shared</td>
<td>block</td>
<td>block</td>
</tr>
<tr>
<td><strong>device</strong>            int GlobalVar;</td>
<td>global</td>
<td>grid</td>
<td>application</td>
</tr>
<tr>
<td><strong>device</strong> <strong>constant</strong> int ConstantVar;</td>
<td>constant</td>
<td>grid</td>
<td>application</td>
</tr>
</tbody>
</table>

- **__device__** is optional when used with **__shared__**, or **__constant__**
- **Recall** `cudaMalloc(...)` allocates memory from the host
  - Constant memory can also be allocated and initialized from the host
- **Automatic variables** without any qualifier reside in a **register**
  - Except arrays that reside in global memory
Memory Hierarchy in CUDA Programs

- **Grid (Device)**
- **Block (0, 0)**
  - Shared memory
  - Registers
  - Thread (0, 0)
  - Thread (1, 0)

- **Block (1, 0)**
  - Shared memory
  - Registers
  - Thread (0, 0)
  - Thread (1, 0)

- **Global / Texture & Surface memory**
- **Constant memory**

- **Host**
Data Reuse

- Same memory locations accessed by neighboring threads

```c
for (int i = 0; i < 3; i++)
    for (int j = 0; j < 3; j++)
        sum += gauss[i][j] * Image[(i+row-1)*width + (j+col-1)];
```
To take advantage of data reuse, we divide the input into tiles that can be loaded into shared memory.

```c
__shared__ int l_data[(L_SIZE+2)*(L_SIZE+2)];
...
Load tile into shared memory
__syncthreads();
for (int i = 0; i < 3; i++){
    for (int j = 0; j < 3; j++){
        sum += gauss[i][j] * l_data[(i+l_row-1)*(L_SIZE+2)+j+l_col-1];
    }
}
```
Synchronization Function

- `void __syncthreads();`
- Synchronizes all threads in a block

- Once all threads in a block have reached this point, execution resumes normally

- Used to avoid RAW / WAR / WAW hazards when accessing shared or global memory
Example: Matrix-Matrix Multiplication (I)

\[ C = A \times B \]
Example: Matrix-Matrix Multiplication (II)

Parallelization approach: assign one thread to each element in the output matrix (C)

\[ C = A \times B \]
Example: Matrix-Matrix Multiplication (III)

```c
__global__ void mm_kernel(float* A, float* B, float* C, unsigned int N) {

    unsigned int row = blockIdx.y*blockDim.y + threadIdx.y;
    unsigned int col = blockIdx.x*blockDim.x + threadIdx.x;

    float sum = 0.0f;
    for(unsigned int i = 0; i < N; ++i) {
        sum += A[row*N + i]*B[i*N + col];
    }
    C[row*N + col] = sum;
}
```

\[ C = A \times B \]
Reuse in Matrix-Matrix Multiplication (I)

\[ C = A \times B \]

Some of the threads in the same thread block use the same input data.
Reuse in Matrix-Matrix Multiplication (II)

\[ C = A \times B \]

Some of the threads in the same thread block use the same input data.
Tiled Matrix-Matrix Multiplication (I)

\[ C = A \times B \]

**Step 1:** Load the first tile of each input matrix to shared memory (each thread loads one element)

Slide credit: Izzat El Hajj
Tiled Matrix-Matrix Multiplication (II)

\[ C_{\text{tile}} = A_{\text{tile}} \times B_{\text{tile}} \]

**Step 2:** Each thread computes its partial sum from the tiles in shared memory (threads wait for each other to finish)

Slide credit: Izzat El Hajj
Tiled Matrix-Matrix Multiplication (III)

\[ C = A \times B \]

...repeat for the next tile

Slide credit: Izzat El Hajj
Tiled Matrix-Matrix Multiplication (IV)

C = A x B

...and the next tile

Slide credit: Izzat El Hajj
Tiled Matrix-Matrix Multiplication (V)

 Declare arrays in shared memory

unsigned int row = blockIdx.y*blockDim.y + threadIdx.y;
unsigned int col = blockIdx.x*blockDim.x + threadIdx.x;

float sum = 0.0f;

for(unsigned int tile = 0; tile < N/TILE_DIM; ++tile) {

    // Load tile to shared memory
    A_s[threadIdx.y][threadIdx.x] = A[row*N + tile*TILE_DIM + threadIdx.x];
    B_s[threadIdx.y][threadIdx.x] = B[(tile*TILE_DIM + threadIdx.y)*N + col];
    __syncthreads();

    // Compute with tile
    for(unsigned int i = 0; i < TILE_DIM; ++i) {
        sum += A_s[threadIdx.y][i]*B_s[i][threadIdx.x];
    }
    __syncthreads();

} // Compute with tile

C[row*N + col] = sum;

Slide credit: Izzat El Hajj
Recommended Readings (II)

  - Chapter 5 - Memory architecture and data locality
Longer Lecture

NVIDIA H100 Tensor Memory Accelerator

- Asynchronous memory copy with LDGSTS instruction vs. TMA

TMA unit reduces addressing overhead

A single thread per warp issues the TMA operation

Support for different tensor layouts (1D-5D)

HetSys Course: Lecture 4: GPU Memory Hierarchy (Fall 2022)

https://youtu.be/ynlGJ1utk4c
P&S Heterogeneous Systems

GPU Memory Hierarchy

Dr. Juan Gómez Luna
Prof. Onur Mutlu

ETH Zürich
Spring 2023
31 March 2023