GPU Programming
Recommended Readings (1)

Upcoming Recommended Reading

  - Available on 1 August 2022
Recommended Readings (II)

- **CUDA Programming Guide**

**CUDA C++ Programming Guide**

The programming guide to the CUDA model and interface.

**Changes from Version 11.3**

- Added **Graph Memory Nodes**.
- Formalized **Asynchronous SIMT Programming Model**.

1. **Introduction**

   1.1. **The Benefits of Using GPUs**

   The Graphics Processing Unit (GPU) provides much higher instruction throughput and memory bandwidth than the CPU within a similar price and power envelope. Many applications leverage these higher capabilities to run faster on the GPU than on the CPU (see [GPU Applications](https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html)). Other computing devices, like FGPA’s, are also very energy efficient, but offer much less programming flexibility than GPUs.

   This difference in capabilities between the GPU and the CPU exists because they are designed with different goals in mind. While the CPU is designed to excel at executing a sequence of operations, called a thread, as fast as possible and can execute a few tens of these threads in parallel, the GPU is designed to excel at executing thousands of them in parallel (amortizing the slower single-thread performance to achieve greater throughput).

   The GPU is specialized for highly parallel computations and therefore designed such that more transistors are devoted to data processing rather than data caching and flow control. The schematic [Figure 1](https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html) shows an example distribution of chip resources for a CPU versus a GPU.
GPU Computing

- Computation is **offloaded to the GPU**
- Three steps
  - CPU-GPU data transfer (1)
  - GPU kernel execution (2)
  - GPU-CPU data transfer (3)
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
Traditional Program Structure in CUDA

- Function prototypes
  ```
  float serialFunction(...);
  __global__ void kernel(...);
  ```

- main()
  1. Allocate memory space on the device – `cudaMalloc(&d_in, bytes)`;
  2. Transfer data from host to device – `cudaMemcpy(d_in, h_in, ...)`;
  3. Execution configuration setup: #blocks and #threads
  4. Kernel call – `kernel<<<execution configuration>>>(args...)`;
  5. Transfer results from device to host – `cudaMemcpy(h_out, d_out, ...)`;

- Kernel – `__global__ void kernel(type args, ...)`
  - Automatic variables transparently assigned to `registers`
  - `Shared memory: __shared__`
  - Intra-block synchronization: `__syncthreads()`;

Slide credit: Hwu & Kirk
CUDA Programming Language

- **Memory allocation**
  
  ```
  cudaMalloc((void**)&d_in, #bytes);
  ```

- **Memory copy**
  
  ```
  cudaMemcpy(d_in, h_in, #bytes, cudaMemcpyHostToDevice);
  ```

- **Kernel launch**
  
  ```
  kernel<<< #blocks, #threads >>>(args);
  ```

- **Memory deallocation**

  ```
  cudaFree(d_in);
  ```

- **Explicit synchronization**

  ```
  cudaDeviceSynchronize();
  ```
Vector Addition (I)

- Our first GPU programming example
- We assign one GPU thread to each element-wise addition
Vector Addition (II)

- The whole set of threads is called a grid
- We need a way to assign threads to GPU cores
We group threads into blocks.
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);
}

const unsigned int numThreadsPerBlock = 512;
Boundary Conditions

- **What if the size of the input is not a multiple of the number of threads per block?**
  - Solution: use the ceiling to launch extra threads then omit the threads after the boundary

```c
const unsigned int numBlocks = (N + numThreadsPerBlock - 1)/numThreadsPerBlock;
```

- **Kernel code**
  ```c
  __global__ void vecadd_kernel(float* A, float* B, float* C, int N) {
    int i = blockDim.x*blockIdx.x + threadIdx.x;
    if(i < N) {
      C[i] = A[i] + B[i];
    }
  }
  ```
Images are 2D data structures
- height x width
- Image[j][i], where 0 ≤ j < height, and 0 ≤ i < width
Row-major layout

Image[j][i] = Image[j x width + i]

Stride = width

Image[0][1] = Image[0 x 8 + 1]
Image[1][2] = Image[1 x 8 + 2]
Indexing and Memory Access: 2D Grid

- **2D blocks**
  - `gridDim.x, gridDim.y`

  Row = `blockIdx.y * blockDim.y + threadIdx.y`
  Row = \(1 \times 2 + 1 = 3\)

  Col = `blockIdx.x * blockDim.x + threadIdx.x`
  Col = \(0 \times 2 + 1 = 1\)

  `Image[3][1] = Image[3 \times 8 + 1]`
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)

https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/
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

https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/
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

Thread - Thread block - Thread block cluster - Grid

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/
NVIDIA H100 Core

48 TFLOPS Single Precision*
24 TFLOPS Double Precision*
800 TFLOPS (FP16, Tensor Cores)*

* Preliminary performance estimates

https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/
Example of data movement between GPU global memory (DRAM) and GPU cores.

A100 improves SM bandwidth efficiency with a new load-global-store-shared asynchronous copy instruction that bypasses L1 cache and register file (RF). Additionally, A100’s more efficient Tensor Cores reduce shared memory (SMEM) loads.

A100 feature: Direct copy from L2 to scratchpad, bypassing L1 and register file.

TMA unit reduces addressing overhead

A single thread per warp issues the TMA operation

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

https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/
**NVIDIA H100 Distributed Shared Memory**

- **Shared memory virtual address space** distributed across the blocks of a cluster
- **Load, store, and atomic operations to other SM’s shared memory**

Thread block clusters and distributed shared memory (DSMEM) are leveraged via `cooperative_groups` API

TMA unit supports copies across thread blocks in a cluster

Asynchronous transaction barriers

https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/
Memory in the H100 GPU Architecture

Approximate access times:
- Control: ≈1 cycle
- Registers: ≈5 cycles
- Constant Cache: ≈5 cycles
- Shared Memory: ≈5 cycles
- L1 Cache: ≈5 cycles
- L2 Cache: ≈500 cycles
- Global Memory: ≈500 cycles

Direct copy between SMs: 3 TB/s

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)];
    }
}
```
Data Reuse: Tiling

- 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];
    }
}
```
void __synctreads();

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
Tiling/Blocking in On-chip Memories

- **Tiling or Blocking**
  - Divide loops operating on arrays into computation chunks so that each chunk can hold its data in the cache (or other on-chip memory, e.g., scratchpad)
  - Avoids cache conflicts between different chunks of computation
  - Essentially: *Divide the working set so that each piece fits in the cache*
  - Let’s first see an example for CPUs
Naïve Matrix Multiplication (I)

- Matrix multiplication: $C = A \times B$
- Consider two input matrices $A$ and $B$ in row-major layout
  - $A$ size is $M \times P$
  - $B$ size is $P \times N$
  - $C$ size is $M \times N$
**Naïve Matrix Multiplication (II)**

- **Naïve implementation** of matrix multiplication has **poor cache locality**

```c
#define A(i,j) matrix_A[i * P + j]
#define B(i,j) matrix_B[i * N + j]
#define C(i,j) matrix_C[i * N + j]

for (i = 0; i < M; i++) { // i = row index
    for (j = 0; j < N; j++) { // j = column index
        C(i, j) = 0; // Set to zero
        for (k = 0; k < P; k++) // Row x Col
            C(i, j) += A(i, k) * B(k, j);
    }
}
```

Consecutive accesses to B are far from each other, in **different cache lines**. Every access to B is likely to cause a cache miss.
Tiled Matrix Multiplication (I)

- We can achieve better cache locality by computing on smaller tiles or blocks that fit in the cache
  - Or in the scratchpad memory and register file if we compute on a GPU

---


Kirk & Hwu, "Chapter 5 - Performance considerations," in "Programming Massively Parallel Processors (Third Edition)", 2017. [https://doi.org/10.1016/B978-0-12-811986-0.00005-4](https://doi.org/10.1016/B978-0-12-811986-0.00005-4)
Tiled Matrix Multiplication (II)

- **Tiled implementation** operates on submatrices (tiles or blocks) that fit fast memories (cache, scratchpad, RF)

```c
#define A(i,j) matrix_A[i * P + j]
#define B(i,j) matrix_B[i * N + j]
#define C(i,j) matrix_C[i * N + j]

for (I = 0; I < M; I += tile_dim){
    for (J = 0; J < N; J += tile_dim){
        Set_to_zero(&C(I, J)); // Set to zero
        for (K = 0; K < P; K += tile_dim)
            Multiply_tiles(&C(I, J), &A(I, K), &B(K, J));
    }
}
```

Multiply small submatrices (tiles or blocks) of size `tile_dim x tile_dim`

---


Kirk & Hwu, "Chapter 5 - Performance considerations," in "Programming Massively Parallel Processors (Third Edition)", 2017. [https://doi.org/10.1016/B978-0-12-811986-0.00005-4](https://doi.org/10.1016/B978-0-12-811986-0.00005-4)
Lecture on Advanced Caches

Digital Design & Computer Arch.
Lecture 24a: Advanced Caches

Prof. Onur Mutlu

ETH Zürich
Spring 2021
28 May 2021

Digital Design and Computer Architecture, ETH Zürich, Spring 2021 (https://safari.ethz.ch/digitaltechnik...)

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)

\[ C = A \times B \]
__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 x 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.

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

- **Sometimes**, we are lucky:
  - The thread finds the data in the L1 cache because it was recently loaded by another thread

- **Sometimes**, we are not lucky:
  - The data gets evicted from the L1 cache before another thread tries to load it

- **Solution:**
  - Let the threads work together to load part of the data and ensure that all threads that need it use it before loading more data
  - Use shared memory to ensure data stays close
  - Optimizing called **tiling** because divides input to tiles
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)
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).
Tiled Matrix-Matrix Multiplication (III)

\[ C = A \times B \]

...repeat for the next tile
Tiled Matrix-Matrix Multiplication (IV)

\[ C = A \times B \]

...and the next tile

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

```c
__shared__ float A_s[TILE_DIM][TILE_DIM];
__shared__ float B_s[TILE_DIM][TILE_DIM];

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();

}

C[row*N + col] = sum;
```

---

Slide credit: Izzat El Hajj
Tiled Matrix Multiplication on GPU

\[
A \times B = C
\]

\[
\begin{array}{c}
(H \times P) \\
(P \times N)
\end{array}
\]

\[
H \times N
\]

\[
\begin{align*}
& t_x = 0, t_y = 0 \\
& t_x = 1, t_y = 0 \\
& b_x = 0, b_y = 0
\end{align*}
\]
Recommended Readings

  - Chapter 4: Memory and data locality
P&S Heterogeneous Systems

GPU Memory Hierarchy

Dr. Juan Gómez Luna
Prof. Onur Mutlu

ETH Zürich
Spring 2022
5 April 2022