GPU Memories
Traditional Program Structure

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

```c
Serial Code (host)

Parallel Kernel (device)
KernelA<<< nBlk, nThr >>>(args);

Serial Code (host)

Parallel Kernel (device)
KernelB<<< nBlk, nThr >>>(args);
```

Slide credit: Hwu & Kirk
Memory Hierarchy in CUDA Programs

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

- **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)
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)
Tiled Matrix-Matrix Multiplication (V)

__shared__ float A_s[TILE_DIM][TILE_DIM];
__shared__ float B_s[TILE_DIM][TILE_DIM];  --- 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();  --- Threads wait for each other to finish loading before computing

    // Compute with tile
    for(unsigned int i = 0; i < TILE_DIM; ++i) {
        sum += A_s[threadIdx.y][i]*B_s[i][threadIdx.x];
    }
    __syncthreads();  --- Threads wait for each other to finish computing before loading

}

C[row*N + col] = sum;
Performance Considerations
Performance Considerations

- Main bottlenecks
  - Global memory access
  - CPU-GPU data transfers

- Memory access
  - Latency hiding
    - Occupancy
  - Memory coalescing
  - Data reuse
    - Shared memory usage

- SIMD (Warp) Utilization: Divergence

- Other considerations
  - Atomic operations: Serialization
  - Data transfers between CPU and GPU
    - Overlap of communication and computation
Memory Access
Latency Hiding via Warp-Level FGMT

- **Warp**: A set of threads that execute the same instruction (on different data elements)

- **Fine-grained multithreading**
  - One instruction per thread in pipeline at a time (No interlocking)
  - Interleave warp execution to hide latencies

- Register values of all threads stay in register file

- FGMT enables long latency tolerance
  - Millions of pixels

Slide credit: Tor Aamodt
Latency Hiding and Occupancy

- **FGMT** can hide long latency operations (e.g., memory accesses)
- **Occupancy**: ratio of active warps to the maximum number of warps per GPU core
Occupyancy

- GPU core, a.k.a. SM, resources (typical values)
  - Maximum number of warps per SM (64)
  - Maximum number of blocks per SM (32)
  - Register usage (256KB)
  - Shared memory usage (64KB)

- Occupancy calculation
  - Number of threads per block (defined by the programmer)
  - Registers per thread (known at compile time)
  - Shared memory per block (defined by the programmer)
CUDA Occupancy Calculator

| A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V |
| 1 | CUDA Occupancy Calculator | | | | | | | | | | | | | | | | | | | | | | |
| 2 | | | | | | | | | | | | | | | | | | | | | | |
| 3 | Just follow steps 1, 2, and 3 below (or click here for help) | | | | | | | | | | | | | | | | | | | | | | |
| 4 | 1) Select Compute Capability (clocks) | 8.6 | | | | | | | | | | | | | | | | | | | | | |
| 5 | 2) Select Shared Memory Configuration (bytes) | 64038 | | | | | | | | | | | | | | | | | | | | | |
| 6 | 3) Select CUDA version | 11.1 | | | | | | | | | | | | | | | | | | | | | |
| 7 | | | | | | | | | | | | | | | | | | | | | | |
| 8 | 1) Enter your resource usage: | | | | | | | | | | | | | | | | | | | | | | |
| 9 | 1a) Threads Per Block | 250 | | | | | | | | | | | | | | | | | | | | | |
| 10 | 1b) Registers Per Thread | 25 | | | | | | | | | | | | | | | | | | | | | |
| 11 | 2a) Shared Memory Per Block (bytes) | 200 | | | | | | | | | | | | | | | | | | | | | |
| 12 | (Don't edit anything below this line) | | | | | | | | | | | | | | | | | | | | | | |
| 13 | 3) GPU Occupancy Data is displayed here and in the graphs: | | | | | | | | | | | | | | | | | | | | | | |
| 14 | Active Threads per Multiprocessor | 1536 | | | | | | | | | | | | | | | | | | | | | |
| 15 | Active Warps per Multiprocessor | 48 | | | | | | | | | | | | | | | | | | | | | |
| 16 | Active Thread Blocks per Multiprocessor | 6 | | | | | | | | | | | | | | | | | | | | | |
| 17 | Occupancy of each Multiprocessor | 169% | | | | | | | | | | | | | | | | | | | | | |
| 18 | | | | | | | | | | | | | | | | | | | | | | |
| 19 | Physical Limits for GPU Compute Capability: | 6.6 | | | | | | | | | | | | | | | | | | | | | |
| 20 | Threads per Warp | 32 | | | | | | | | | | | | | | | | | | | | | |
| 21 | Max Warps per Multiprocessor | 48 | | | | | | | | | | | | | | | | | | | | | |
| 22 | Max Threads Blocks per Multiprocessor | 1536 | | | | | | | | | | | | | | | | | | | | | |
| 23 | Maximum Thread Block Size | 1024 | | | | | | | | | | | | | | | | | | | | | |
| 24 | Registers per Multiprocessor | 64038 | | | | | | | | | | | | | | | | | | | | | |
| 25 | Max Registers per Thread Block | 65536 | | | | | | | | | | | | | | | | | | | | | |
| 26 | Max Registers per Thread | 255 | | | | | | | | | | | | | | | | | | | | | |
| 27 | Shared Memory per Multiprocessor (bytes) | 65536 | | | | | | | | | | | | | | | | | | | | | |
| 28 | Max Shared Memory per Block | 65536 | | | | | | | | | | | | | | | | | | | | | |
| 29 | Register allocation unit size | 256 | | | | | | | | | | | | | | | | | | | | | |
| 30 | Register allocation granularity | warp | | | | | | | | | | | | | | | | | | | | | |
| 31 | Shared Memory allocation unit size | 128 | | | | | | | | | | | | | | | | | | | | | |
| 32 | Warp allocation granularity | 4 | | | | | | | | | | | | | | | | | | | | | |
| 33 | Shared Memory Per Block (bytes) (CUDA runtime use) | 1024 | | | | | | | | | | | | | | | | | | | | | |
| 34 | Allocated Resources | Per Block | Limit Per SM | | | | | | | | | | | | | | | | | | | | | |
| 35 | Warps (Threads Per Block / Threads Per Warp) | 8 | 48 | | | | | | | | | | | | | | | | | | | | | |
| 36 | Registers (Warp limit per SM due to per-warp reg count) | 128 | 512 | | | | | | | | | | | | | | | | | | | | | |
| 37 | Shared Memory (Bytes) | 2048 | 65536 | | | | | | | | | | | | | | | | | | | | | |
| 38 | (Max SM is an alternative for choosing Multiprocessor) | | | | | | | | | | | | | | | | | | | | | | |
| 39 | Max. SM is an alternative for choosing Multiprocessor | | | | | | | | | | | | | | | | | | | | | | |
| 40 | Maximum Thread Blocks Per Multiprocessor | Blocks | Warp/Block | Warp/SM | | | | | | | | | | | | | | | | | | | | | |
| 41 | Limited by Max Warps or Max Blocks per Multiprocessor | 9 | 48 | | | | | | | | | | | | | | | | | | | | | |
| 42 | Limited by Registers per Multiprocessor | 4 | | | | | | | | | | | | | | | | | | | | | |
| 43 | Limited by Shared Memory per Multiprocessor | 256 | | | | | | | | | | | | | | | | | | | | | |
| 44 | Occupancy = (Max SM or Max Warps per SM) / Occupancy | 48 / 9 | 100% | | | | | | | | | | | | | | | | | | | | | |
| 45 | CUDA Occupancy Calculator | | | | | | | | | | | | | | | | | | | | | | |
| 46 | Version: | 11.1 | | | | | | | | | | | | | | | | | | | | | |
| 47 | Copyright and License | | | | | | | | | | | | | | | | | | | | | | |

CUDA Occupancy Calculator (II)

CUDA Occupancy Calculator
The CUDA Occupancy Calculator allows you to compute the multiprocessor occupancy of a GPU by a given CUDA kernel.

Overview
The CUDA Occupancy Calculator allows you to compute the multiprocessor occupancy of a GPU by a given CUDA kernel. The multiprocessor occupancy is the ratio of active warps to the maximum number of warps supported on a multiprocessor of the GPU. Each multiprocessor on the device has a set of N registers available for use by CUDA program threads. These registers are a shared resource that are allocated among the thread blocks executing on a multiprocessor.

The CUDA compiler attempts to minimize register usage to maximize the number of thread blocks that can be active in the machine simultaneously. If a program tries to launch a kernel for which the registers used per thread times the thread block size is greater than N, the launch will fail.

Click CUDA Occupancy Calculator[XLS] to download the spreadsheet.
Memory Layout of a Matrix in C
The DRAM Subsystem
The Top-Down View
DRAM Subsystem Organization

- Channel
- DIMM
- Rank
- Chip
- Bank
- Row/Column
The DRAM Subsystem

“Channel”

DIMM (Dual in-line memory module)

Processor

Memory channel

Memory channel
Breaking down a DIMM (module)

DIMM (Dual in-line memory module)

Rank 0: collection of 8 chips

Rank 1
Breaking down a Rank

Rank 0

Chip 0

Chip 1

... Chip 7

<0:7> <8:15> <56:63>

Data <0:63>

<0:63>

<0:63>
Breaking down a Chip
Inside a DRAM Chip

- Subarray (2D Array of DRAM Cells)
- Sense Amplifiers
- Row Buffer
- DRAM Bank
- DRAM Chips
- DRAM Module
DRAM Cell Operation

1. ACTIVATE (ACT)
2. READ/WRITE
3. PRECHARGE (PRE)
DRAM Cell Operation - ACTIVATE

1. Raise wordline

2. Capacitor charge shares charge with bitline

3. Enable sense amplifier

4. Amplify deviation in the bitline

5. Capacitor charge is restored

6. Row buffer stores the cell value

1. ACTIVATE (ACT)

2. READ/WRITE

3. PRECHARGE (PRE)
DRAM Cell Operation – READ/WRITE

1. ACTIVATE (ACT)

2. READ/WRITE

3. PRECHARGE (PRE)

Read/Write the value latched in sense amplifier
DRAM Cell Operation - PRECHARGE

1. Lower wordline
2. Precharge bitline for next access
3. Disable sense amplifier

- storage capacitor
- access transistor
- bitline
- $\frac{1}{2} V_{DD}$

1. ACTIVATE (ACT)
2. READ/WRITE
3. PRECHARGE (PRE)
DRAM Bank Operation

Access Address:
(Row 0, Column 0)
(Row 0, Column 1)
(Row 0, Column 85)
(Row 1, Column 0)

Row address 0

Columns

Row decoder

Rows

Row 1

Row Buffer

CONFLICT!

Column address 0

Column mux

Data

27
DRAM Burst

- **Accessing data in different bursts (rows)**
  - Need to access the array again

  Timeline:

  ![Timeline diagram showing different bursts](image)

- **Accessing data in the same burst (row)**
  - No need to access the array again, just the multiplexer

  Timeline:

  ![Timeline diagram showing same burst](image)

- **Accessing data in the same burst is faster than accessing data in different bursts**
Recall: Memory Banking

- Memory is divided into banks that can be accessed independently; banks share address and data buses (to minimize pin cost)
- Can start and complete one bank access per cycle
- Can sustain N concurrent accesses if all N go to different banks

![Diagram of memory banks and buses]

Picture credit: Derek Chiou
Multiple Banks (Interleaving) and Channels

- Multiple banks
  - Enable concurrent DRAM accesses
  - Bits in address determine which bank an address resides in
- Multiple independent channels serve the same purpose
  - But they are even better because they have separate data buses
  - Increased bus bandwidth

- Enabling more concurrency requires reducing
  - Bank conflicts
  - Channel conflicts
- How to select/randomize bank/channel indices in address?
  - Lower order bits have more entropy
  - Randomizing hash functions (XOR of different address bits)
Latency Hiding with Multiple Banks

- With one bank, time still wasted in between bursts

- Latency can be hidden by having multiple banks

- Need many threads to simultaneously access memory to keep all banks busy
  - Achieved with having high occupancy in GPU cores (SMs)
    - Similar idea to hiding pipeline latency in the core

Slide credit: Izzat El Hajj
Lecture on Memory Organization & Technology

Breaking down a Chip

DEPARTMENT OF INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING (D-ITET)
Digital Design & Comp. Arch. - Lecture 22: Memory Organization & Technology (ETH Zürich, Spring ’21)

Onur Mutlu Lectures
19.6K subscribers

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

Memory Coalescing (I)

- When threads in the same warp access consecutive memory locations in the same burst, the accesses can be combined and served by one burst
  - One DRAM transaction is needed
  - Known as memory coalescing

- If threads in the same warp access locations not in the same burst, accesses cannot be combined
  - Multiple transactions are needed
  - Takes longer to service data to the warp
  - Sometimes called memory divergence
When accessing global memory, we want to make sure that concurrent threads access nearby memory locations.

Peak bandwidth utilization occurs when all threads in a warp access one cache line (or several consecutive cache lines).

Slide credit: Hwu & Kirk
Uncoalesced Memory Accesses

Access direction in Kernel code

Time Period 1

Time Period 2

Slide credit: Hwu & Kirk
Coalesced Memory Accesses

Access direction in Kernel code

Slide credit: Hwu & Kirk
**Array of Structures vs. Structure of Arrays**

Tenemos 2 data layouts principales (AoS y SoA) y uno nuevo propuesto (ASTA).

ASTA permite transformar uno en otro más rápido y facilita hacerlo in-place, para ahorrar memoria.

La granularidad en ASTA, es decir, el ancho del tile, estará relacionado con la granularidad de acceso a la memoria (warp_size = 32, por ejemplo).

Convertir entre los distintos layouts, en realidad es transponer. Por ejemplo, AoS a ASTA:

Transponer es permutar (los números representan posiciones en la memoria y los colores, tipo de dato):

```c
struct foo{
    float a[8];
    float b[8];
    float c[8];
    int d[8];
} A;
```

```c
struct foo{
    float a;
    float b;
    float c;
    int d;
} A[8];
```
CPUs Prefer AoS, GPUs Prefer SoA

- Linear and strided accesses

GPU

Throughput (GB/s)

0.0 1.0 2.0 3.0 4.0 5.0

Stride (Structure size)

1 2 4 8 16 32 64 128 256 512 1024

GPU

Throughput (GB/s)

0.0 1.0 2.0 3.0 4.0 5.0

Stride (Structure size)

1 2 4 8 16 32 64 128 256 512 1024

CPU

1CPU 2CPU 4CPU

Throughput (GB/s)

0.0 1.0 2.0 3.0 4.0 5.0

Stride (Structure size)

1 2 4 8 16 32 64 128 256 512 1024

AMD Kaveri A10-7850K

Sung+, “DL: A data layout transformation system for heterogeneous computing,” INPAR 2012
Use Shared Memory to Improve Coalescing

Original Access Pattern

Tiled Access Pattern

Copy into scratchpad memory

Perform multiplication with scratchpad values

Slide credit: Hwu & Kirk
Data Reuse

- Same memory locations accessed by neighboring threads

```
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.

```
__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];
    }
}
```
Shared Memory

- Shared memory is an **interleaved (banked) memory**
  - Each bank can service one address per cycle

- Typically, 32 banks in NVIDIA GPUs
  - Successive 32-bit words are assigned to successive banks
    - Bank = Address % 32

- Bank conflicts are **only possible within a warp**
  - No bank conflicts between different warps
Shared Memory Bank Conflicts (I)

- Bank conflict free

Linear addressing: stride = 1

Random addressing 1:1
Shared Memory Bank Conflicts (II)

- N-way bank conflicts

2-way bank conflict: stride = 2

8-way bank conflict: stride = 8

Slide credit: Hwu & Kirk
Use Shared Memory to Improve Coalescing

Original Access Pattern

Tiled Access Pattern

Copy into scratchpad memory

Perform multiplication with scratchpad values

Slide credit: Hwu & Kirk
Reducing Shared Memory Bank Conflicts

- Bank conflicts are only possible within a warp
  - No bank conflicts between different warps

- If strided accesses are needed, some optimization techniques can help
  - Padding
  - Randomized mapping
  - Hash functions
SIMD Utilization
Threads Can Take Different Paths in Warp-based SIMD

- Each thread can have **conditional control flow instructions**
- Threads can execute different control flow paths

Slide credit: Tor Aamodt
Control Flow Problem in GPUs/SIMT

- A GPU uses a SIMD pipeline to save area on control logic
  - Groups scalar threads into warps

- Branch divergence occurs when threads inside warps branch to different execution paths

This is the same as conditional/predicated/masked execution. Recall the Vector Mask and Masked Vector Operations?
SIMD Utilization

- Intra-warp divergence

```c
Compute(threadIdx.x);
if (threadIdx.x % 2 == 0){
    Do_this(threadIdx.x);
}
else{
    Do_that(threadIdx.x);
}
```
Increasing SIMD Utilization

- **Divergence-free execution**

```cpp
Compute(threadIdx.x);
if (threadIdx.x < 32){
    Do_this(threadIdx.x * 2);
}
else{
    Do_that((threadIdx.x%32)*2+1);
}
```
Vector Reduction: Naïve Mapping (I)

Thread 0  Thread 2  Thread 4  Thread 6  Thread 8  Thread 10

0  1  2  3  4  5  6  7  8  9  10  11 ...

1  0+1  2+3  4+5  6+7  8+9  10+11

2  0...3  4..7  8..11

3  0..7  8..15

iterations

Slide credit: Hwu & Kirk
Program with **low SIMD utilization**

```c
__shared__ float partialSum[];

unsigned int t = threadIdx.x;

for (int stride = 1; stride < blockDim.x; stride *= 2) {
    __syncthreads();

    if (t % (2*stride) == 0)
        partialSum[t] += partialSum[t + stride];
}
```
Divergence-Free Mapping (I)

- All active threads belong to the same warp

Slide credit: Hwu & Kirk
Program with high SIMD utilization

```c
__shared__ float partialSum[]

unsigned int t = threadIdx.x;

for (int stride = blockDim.x; stride > 0; stride >>= 1){
    __syncthreads();
    if (t < stride)
        partialSum[t] += partialSum[t + stride];
}
```
Recommended Readings

  - Chapter 5: Performance considerations
GPU Performance Considerations

Dr. Juan Gómez Luna
Prof. Onur Mutlu
ETH Zürich
Fall 2021
4 November 2021