P&S Heterogeneous Systems

GPU Performance Considerations

Dr. Juan Gómez Luna
Prof. Onur Mutlu

ETH Zürich
Fall 2021
4 November 2021
GPU Memories
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);
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
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)

Declare arrays in shared memory

__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
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

---

**4 active warps**

- Warp 0
  - Instruction 3
- Warp 1
  - Instruction 2
- Warp 2
  - Instruction 1
- Warp 3
  - Instruction 1

**Warp 0**
- Instruction 4 (Long latency)

---

**2 active warps**

- Warp 0
  - Instruction 3
- Warp 1
  - Instruction 2

- Warp 0
  - Instruction 4 (Long latency)

- Warp 0
  - Instruction 5
Occupancy

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

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)

Side view

Front of DIMM

Rank 0: collection of 8 chips

Back of DIMM

Rank 1
Breaking down a Rank

Rank 0

Chip 0
Chip 1
... Chip 7

Data <0:63>
Breaking down a Chip

Chip 0

8 banks

<0:7>
Inside a DRAM Chip

- **Subarray (2D Array of DRAM Cells)**
- **Sense Amplifiers**
- **Row Buffer**
- **Bitline**
- **Wordline**
- **DRAM Cells**
- **Access Transistor**
- **Storage Capacitor**

**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 is restored charge with bitline

3. Enable sense amplifier

4. Amplify deviation in the bitline

5. Amplify deviation in the bitline

6. Row buffer stores the cell value
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

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
Row decoder
Columns
Rows

Column address 85
Column mux
Row Buffer
Data

HIT
HIT
CONFLICT!
DRAM Burst

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

  ![Timeline for accessing data in different bursts](image1)

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

  ![Timeline for accessing data in the same burst](image2)

- **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 banking]

**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 SMs
    - Similar idea to hiding pipeline latency in the core

Slide credit: Izzat El Hajj
Breaking down a Chip
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

Slide credit: Izzat El Hajj
Memory Coalescing (II)

- 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.

![Diagram showing coalesced and not coalesced memory access]

Thread 1
Thread 2
Uncoalesced Memory Accesses

Access direction in Kernel code

Time Period 1

Time Period 2

M_0,0 M_1,0 M_2,0 M_3,0
M_0,1 M_1,1 M_2,1 M_3,1
M_0,2 M_1,2 M_2,2 M_3,2
M_0,3 M_1,3 M_2,3 M_3,3

T_1 T_2 T_3 T_4

M_0,0 M_1,0 M_2,0 M_3,0
M_0,1 M_1,1 M_2,1 M_3,1
M_0,2 M_1,2 M_2,2 M_3,2
M_0,3 M_1,3 M_2,3 M_3,3

Slide credit: Hwu & Kirk
Coalesced Memory Accesses

Access direction in Kernel code

Time Period 1
\( T_1 \) \( T_2 \) \( T_3 \) \( T_4 \)

Time Period 2
\( T_1 \) \( T_2 \) \( T_3 \) \( T_4 \)

\( M \)

\( M_{0,0} \) \( M_{1,0} \) \( M_{2,0} \) \( M_{3,0} \)
\( M_{0,1} \) \( M_{1,1} \) \( M_{2,1} \) \( M_{3,1} \)
\( M_{0,2} \) \( M_{1,2} \) \( M_{2,2} \) \( M_{3,2} \)
\( M_{0,3} \) \( M_{1,3} \) \( M_{2,3} \) \( M_{3,3} \)

Slide credit: Hwu & Kirk
**AoS vs. SoA**

- **Array of Structures vs. Structure of Arrays**

  ```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];
  ```

  

Sung+, “DL: A data layout transformation system for heterogeneous computing,” INPAR 2012
CPUs Prefer AoS, GPUs Prefer SoA

- Linear and strided accesses

**Throughput (GB/s)** vs **Stride (Structure size)**

**GPU**

- AMD Kaveri A10-7850K

**CPU**

- 1CPU, 2CPU, 4CPU

---

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

```java
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];
    }
}
```
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

Slide credit: Hwu & Kirk
Shared Memory Bank Conflicts (II)

- **N-way bank conflicts**

2-way bank conflict: stride = 2

8-way bank conflict: stride = 8
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
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**

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

- **Divergence-free execution**

```c
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

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

Thread 2

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

Thread 4

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

Thread 6

4..7 8..11

Thread 8

8..11

Thread 10

8..15

iterations

comments:

0...7

1

2

3

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

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
<th>Thread 2</th>
<th>...</th>
<th>Thread 14</th>
<th>Thread 15</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>...</td>
</tr>
<tr>
<td>iteration 1</td>
<td>0+16</td>
<td></td>
<td></td>
<td></td>
<td>15+31</td>
</tr>
<tr>
<td>iteration 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>iteration 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Slide credit: Hwu & Kirk
Divergence-Free Mapping (II)

- Program with **high SIMD utilization**

```c
__shared__ float partialSum[]

unsigned int t = threadIdx.x;

for (int stride = blockDim.x; stride > 1; stride >>= 1){
    __syncthreads();

    if (t < stride)
        partialSum[t] += partialSum[t + stride];
}
```
Recommended Readings

  - Chapter 5: Performance considerations