P&S Heterogeneous Systems

GPU Performance Considerations

Dr. Juan Gómez Luna
Prof. Onur Mutlu
ETH Zürich
Spring 2022
12 April 2022
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)

\[ \text{KernelA} \lll \text{nBlk}, \text{nThr} \rrr (\text{args}); \]

Serial Code (host)

Parallel Kernel (device)

\[ \text{KernelB} \lll \text{nBlk}, \text{nThr} \rrr (\text{args}); \]
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**
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
  - CPU-GPU data transfers
  - Global memory access

- 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
Occupancy

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

Just follow steps 1, 2, and 3 below or click here for help.

1. Select Compute Capability (click) 8.6
   - select computes capability
2. Select Shared Memory Size (bytes) 65536
   - select shared memory size
3. Select CUDA version 11.1
   - select CUDA version

[Click Here for detailed instructions on how to use this occupancy calculator.]

For more information on NVIDIA CUDA, visit http://developer.nvidia.com/cuda

Your chosen resource usage is indicated by the red triangle on the graph. The yellow data points represent the range of possible block sizes, register counts, and shared memory allocation.

### Impact of Varying Block Size

- **My Block Size**: 32

![Impact of Varying Block Size](image)

### Impact of Varying Shared Memory Usage Per Block

- **Shared Memory**: 48

![Impact of Varying Shared Memory](image)

### Impact of Varying Register Count Per Thread

- **My Register Count**: 32

![Impact of Varying Register Count](image)

### Allocated Resources

<table>
<thead>
<tr>
<th>Resource</th>
<th>Per Block</th>
<th>Limit Per SM</th>
<th>Limitable</th>
<th>Allocatable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blocks/SM</td>
<td>64</td>
<td>48</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>Warps/Block</td>
<td>96</td>
<td>48</td>
<td>96</td>
<td>96</td>
</tr>
<tr>
<td>Registers</td>
<td>69</td>
<td>64</td>
<td>69</td>
<td>69</td>
</tr>
<tr>
<td>Shared Memory (bytes)</td>
<td>2048</td>
<td>65536</td>
<td>32</td>
<td>32</td>
</tr>
</tbody>
</table>

Note: SM is an abbreviation for Streaming Multiprocessor

### Maximum Thread Blocks Per Multiprocessor

<table>
<thead>
<tr>
<th>Multiprocessor</th>
<th>Blocks/SM</th>
<th>Warps/Block</th>
<th>Warps/SM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Limited by Max Warps/SM</td>
<td>48</td>
<td>48</td>
<td></td>
</tr>
<tr>
<td>Limited by Max Blocks per Multiprocessor</td>
<td>96</td>
<td>96</td>
<td></td>
</tr>
<tr>
<td>Limited by Shared Memory</td>
<td>64</td>
<td>64</td>
<td></td>
</tr>
</tbody>
</table>

- **Physical Warps/SM**: 48
- **Occupancy**: 48 / 48 = 100%

---

[Link to NVIDIA CUDA Occupancy Calculator](https://docs.nvidia.com/cuda/cuda-occupancy-calculator/CUDA_Occupancy_Calculator.xls)
CUDA Occupancy Calculator (II)

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

[Deprecated] Excel based Occupancy Calculator is deprecated. Occupancy calculator is available in Nsight Compute. Please refer to Nsight Compute Occupancy Calculator documentation for more details on usage.

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.
CUDA Occupancy Calculator (III)

9. Occupancy Calculator

NVIDIA Nsight Compute provides an Occupancy Calculator that allows you to compute the multiprocessor occupancy of a GPU for a given CUDA kernel. It offers feature parity to the CUDA Occupancy Calculator spreadsheet.

The Occupancy Calculator can be opened directly from a profile report or as a new activity. The occupancy calculator data can be saved to a file using File > Save. By default, the file uses the .ncu-occ extension. The occupancy calculator file can be opened using File > Open File.

1. Launching from the Connection Dialog

Select the Occupancy Calculator activity from the connection dialog. You can optionally specify an occupancy calculator data file, which is used to initialize the calculator with the data from the saved file. Click the Launch button to open the Occupancy Calculator.

https://docs.nvidia.com/nsight-compute/NsightCompute/index.html#occupancy-calculator
Memory Layout of a Matrix in C

M

\[ \begin{array}{cccc}
M_{0,0} & M_{1,0} & M_{2,0} & M_{3,0} \\
M_{0,1} & M_{1,1} & M_{2,1} & M_{3,1} \\
M_{0,2} & M_{1,2} & M_{2,2} & M_{3,2} \\
M_{0,3} & M_{1,3} & M_{2,3} & M_{3,3} \\
\end{array} \]

Slide credit: Hwu & Kirk
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
Breaking down a Rank

Diagram showing the connection between Rank 0 and multiple chips (Chip 0, Chip 1, ..., Chip 7). Arrows indicate the data flow between Rank 0 and the chips. The data range is indicated as <0:63> for data transfer.
Breaking down a Chip
Inside a DRAM Chip

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

**Key Components:**
- **DRAM Chips**
- **DRAM Module**

**Key Terms:**
- **DRAM Chips**
- **DRAM Bank**
- **DRAM Cells**
- **Access Transistor**
- **Storage Capacitor**
- **Bitline**
- **Wordline**
DRAM Cell Operation

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

1. Raise wordline
2. Cap storage capacitor with bitline charge
3. Enable sense amplifier
4. Amplify deviation in the bitline
5. Cap storage 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
1. Lower wordline
2. Precharge bitline for next access
3. Disable sense amplifier

1. ACTIVATE (ACT)
2. READ/WRITE
3. PRECHARGE (PRE)
Access Address:
(Row 0, Column 0)
(Row 0, Column 1)
(Row 0, Column 85)
(Row 1, Column 0)

Row address 0

Columns

Rows

Row 1

Row Buffer

CONFLICT!

Column address 05

Column mux

Data
DRAM Burst

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

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

- **Accessing data in the same burst is faster than accessing data in different bursts**

Timeline:

Slide credit: Izzat El Hajj
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
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
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
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 (or several consecutive cache lines).

---

*Slide credit: Hwu & Kirk*
Uncoalesced Memory Accesses

Access direction in Kernel code

Time Period 1

T1 T2 T3 T4

M0,0 M1,0 M2,0 M3,0
M0,1 M1,1 M2,1 M3,1
M0,2 M1,2 M2,2 M3,2
M0,3 M1,3 M2,3 M3,3

Time Period 2

T1 T2 T3 T4

M0,0 M1,0 M2,0 M3,0
M0,1 M1,1 M2,1 M3,1
M0,2 M1,2 M2,2 M3,2
M0,3 M1,3 M2,3 M3,3
Coalesced Memory Accesses

Access direction in Kernel code

Time Period 1
T₁ T₂ T₃ T₄

Time Period 2
T₁ T₂ T₃ T₄

M₀,₀ M₁,₀ M₂,₀ M₃,₀
M₀,₁ M₁,₁ M₂,₁ M₃,₁
M₀,₂ M₁,₂ M₂,₂ M₃,₂
M₀,₃ M₁,₃ M₂,₃ M₃,₃

M₀,₄ M₁,₄ M₂,₄ M₃,₄

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

Structure of Arrays (SoA)

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

Array of Structures (AoS)

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

- Linear and strided accesses

**Throughput (GB/s)**

**Stride (Structure size)**

---

**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

```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
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

![Diagram showing different paths for threads in a warp](image-url)

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**

```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  Thread 2  Thread 4  Thread 6  Thread 8  Thread 10

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

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

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

0..7  8..15

iterations

Slide credit: Hwu & Kirk
Vector Reduction: Naïve Mapping (II)

- 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
Divergence-Free Mapping (II)

- 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];
}
```
Atomic Operations
Atomic Operations (I)

- CUDA provides **atomic instructions** on shared memory and global memory
  - They perform **read-modify-write** operations atomically

- Arithmetic functions
  - Add, sub, max, min, exch, inc, dec, CAS

    ```
    int atomicAdd(int*, int);
    ```

    - Return value (old value)
    - Pointer to shared memory or global memory
    - Value to add

- Bitwise functions
  - And, or, xor

- Datatypes: int, uint, ull, float (half, single, double)*

* Datatypes for different atomic operations in [https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#atomic-functions](https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#atomic-functions)
Atomic Operations (II)

- Atomic operations serialize the execution if there are atomic conflicts

No atomic conflict = concurrent updates

Atomic conflict = serialized updates
Uses of Atomic Operations

- **Computation**
  - Atomics on an array that will be the output of the kernel
  - Example
    - Histogram, reduction

- **Synchronization**
  - Atomics on memory locations that are used for synchronization or coordination
  - Example
    - Counters, locks, flags...

- Use them to prevent *data races* when more than one thread need to update the same memory location
Histograms are widely used in image processing

- Some computation before voting in the histogram may be needed

```
For (each pixel i in image I){
    Pixel = I[i]  // Read pixel
    Pixel’ = Computation(Pixel)  // Optional computation
    Histogram[Pixel’]++  // Vote in histogram bin
}
```

- Parallel threads frequently incur atomic conflicts in image histogram computation
Optimized Parallel Reduction

- 7 versions in CUDA samples: Tree-based reduction in shared memory
  - Version 0: No whole warps active
  - Version 1: Contiguous threads, but many bank conflicts
  - Version 2: No bank conflicts
  - Version 3: First level of reduction when reading from global memory
  - Version 4: Warp shuffle or unrolling of final warp
  - Version 5: Warp shuffle or complete unrolling
  - Version 6: Multiple elements per thread sequentially

https://docs.nvidia.com/cuda/cuda-samples/index.html#cuda-parallel-reduction
Reduction with Atomic Operations

- 3 new versions of reduction based on 3 previous versions
  - Version 0: No whole warps active
  - Version 3: First level of reduction when reading from global memory
  - Version 6: Multiple elements per thread sequentially
- New versions 7, 8, and 9
  - Replace the for loop (tree-based reduction) with one shared memory atomic operation per thread
Asynchronous Data Transfers between CPU and GPU
CUDA Streams

- **CUDA streams** (command queues in OpenCL)
- Sequence of operations that are performed in order
  - 1. Data transfer CPU-GPU
  - 2. Kernel execution
    - D input data instances, B blocks
    - \#Streams: \( \frac{D}{\#\text{Streams}} \) data instances, \( \frac{B}{\#\text{Streams}} \) blocks
  - 3. Data transfer GPU-CPU
Asynchronous Transfers between CPU & GPU

- Computation divided into #Streams
  - D input data instances, B blocks
  - #Streams
    - D/#Streams data instances
    - B/#Streams blocks

- Estimates
  
  \[ t_E + \frac{t_T}{\#Streams} \quad t_T + \frac{t_E}{\#Streams} \]
  
  \[ t_E \geq t_T \text{ (dominant kernel)} \quad t_T > t_E \text{ (dominant transfers)} \]
Overlap of Data Transfers and Kernel Execution

Code for devices that do not support concurrent data transfers

```c
// Create streams
int number_of_streams = 32;
cudaStream_t stream[number_of_streams]; // Stream declaration
for(int i = 0; i < number_of_streams; ++i)
    cudaStreamCreate(&stream[i]); // Stream creation

// CPU-GPU data transfers
for (int i = 0; i < number_of_streams; ++i)
    cudaMemcpyAsync(inputDevPtr + i * size, hostPtr + i * size, size,
                    cudaMemcpyHostToDevice, stream[i]);

// Kernel launches
for (int i = 0; i < number_of_streams; ++i)
    MyKernel<<<num_blocks / number_of_streams, num_threads, 0, stream[i]>>> (outputDevPtr + i * size, inputDevPtr + i * size, size);

// GPU-CPU data transfers
for (int i = 0; i < number_of_streams; ++i)
    cudaMemcpyAsync(hostPtr + i * size, outputDevPtr + i * size, size,
                     cudaMemcpyDeviceToHost, stream[i]);

cudaDeviceSynchronize(); // Explicit synchronization

// Destroy streams
for (int i = 0; i < number_of_streams; ++i)
    cudaStreamDestroy(stream[i]); // Stream destruction
```

Check CUDA programming guide
https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#streams

Applications with independent computation on different data instances can benefit from asynchronous transfers.

For instance, video processing.

Use Case: Video Processing

- Non-streamed execution: A sequence of 6 frames is transferred to device. 6 x b blocks compute on the sequence of frames.
- Streamed execution: A chunk of 2 frames is transferred to device. 2 x b blocks compute on the chunk, while the second chunk is being transferred. Execution time saved thanks to streams.

256-bin histogram calculation

Video Processing: Performance Results (II)

- RGB-to-grayscale conversion

Performance Considerations

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

- 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
Recommended Readings

  - Chapter 5: Performance considerations
  - Chapter 18 - Programming a heterogeneous computing cluster, Section 18.5
GPU Performance Considerations

Dr. Juan Gómez Luna
Prof. Onur Mutlu
ETH Zürich
Spring 2022
12 April 2022