Lecture 20
Parallel Processing
Introduction to Parallel Processing

- Parallel Processing: divides a problem into smaller tasks executed concurrently by multiple processing units: (1) speedup (2) enables solving larger or problems.

- Real-world Applications
  - Scientific simulations (e.g., climate modeling, molecular dynamics)
  - Image and video processing (e.g., rendering, compression)
  - Machine learning and AI (e.g., ChatGPT, training large language models)
Parallel Processing Techniques
Unlock the Power of High Performance Computing

- Optimize locality and reduce branching overhead
  - Loop reordering
  - Loop tiling
  - Loop unrolling

- SIMD (single instruction, multiple data) programming

- Multithreading

- CUDA programming

- Distributed Memory Programming
Parallel Processing Techniques
Unlock the Power of High Performance Computing

- Optimize locality and reduce branching overhead
  - Loop reordering
  - Loop tiling
  - Loop unrolling

- SIMD (single instruction, multiple data) programming

- Multithreading

- CUDA programming

- Distributed Memory Programming
Why Locality and Branching Overhead Improvement

What are locality and branching and how to improve?

- **Locality:**
  - Access the same set of memory locations repeatedly.
  - **Spatial Locality:**
    - Accessing memory locations that are close together repeatedly.
  - **Temporal Locality:**
    - Reuse of data or instructions that have been accessed recently.
  ✓ e.g., loop reordering, loop tiling

- **Branching:**
  - The mispredicted branches lead to wasted processing resources and idle time.
  - Restructure the code to minimize branching and improve parallelism.
  ✓ e.g., loop unrolling
Optimize locality and reduce branching overhead

Loop Reordering

- Improve data locality of caches
  - Data movement (cache miss) is much more expense
  - Chuck of memory is fetched at a time (cache line)

- Reduce cache miss by loop reordering
  - Change the order of loop iteration variables
  - e.g., i, j, k -> i, k, j

```python
for i in range(0, N):
    for j in range(0, N):
        for k in range(0, N):
            C[i][j] += A[i][k] * B[k][j]
```

- Reorder (i, j, k) as (i, k, j)

```python
for i in range(0, N):
    for k in range(0, N):
        for j in range(0, N):
            C[i][j] += A[i][k] * B[k][j]
```

- Example:
  - Poor data locality!
    - Assume stored in row-major order
  - Good data locality
    - Assume stored in row-major order

- Reorder (i, j, k) as (i, k, j)
Optimize locality and reduce branching overhead
Loop reordering: Speed up Matrix Multiplication

```c
void MatmulOperator::naive_mat_mul(const struct matmul_params *params)
{
    int i, j, k;
    // ... Get A[][], B[][], C[][] from params

    for (i = 0; i < C->row; i++)
        for (j = 0; j < C->column; j++){
            float acc = 0;
            for (k = 0; k < A->column; k++)
                acc += A[i][k] * B[k][j];
            C[i][j] = acc;
        }
}
```

```c
void MatmulOperator::mat_mul_reordering(const struct matmul_params *params)
{
    int i, j, k;
    // ... Get A[][], B[][], C[][] from params

    for (i = 0; i < C->row; i++)
        for (k = 0; k < A->column; k++)
        {
            float acc = 0;
            float Aik = A[i][k];
            for (j = 0; j < C->column; j++)
                acc += Aik * B[k][j];
            C[i][j] = acc;
        }
}
```

Without Loop reordering:

```c
naive_mat_mul: 24296 ms
```

With Loop reordering:

```c
mat_mul_reordering: 1979 ms
```

12x speed up

*Results are measured on Intel Xeon 4114*
Optimize locality and reduce branching overhead

Loop tiling: Reduce cache miss

- What if data is much larger than cache size?
  - Data in cache will be evicted before reuse -> cache miss ↑
  - e.g., B is much larger than cache size

- How loop tiling reduces cache miss?
  - Partition the loop iteration space
  - Fit elements accessed in the loop into cache
  - Ensure data stays in the cache until it is reused

```
for i in range(0, N):
    for k in range(0, N):
        for j in range(0, N):
            C[i][j] += A[i][k] * B[k][j]
```

Accessed elements in B = \( N^2 \)

\[\begin{bmatrix}
  b_{00} & b_{01} & \cdots & \cdots \\
  b_{10} & b_{11} & \cdots & \cdots \\
  \vdots & \vdots & \ddots & \vdots \\
  \vdots & \vdots & \ddots & \vdots \\
\end{bmatrix}\]

\[\rightarrow T_j = \text{TILE\_SIZE}\]

\[\text{for } j_t \text{ in range}(0, N, T_j):\]

\[\text{for } i \text{ in range}(0, N):\]

\[\text{for } k \text{ in range}(0, N):\]

\[\text{for } j \text{ in range}(j_t, j_t + T_j):\]

\[C[i][j] += A[i][k] * B[k][j]\]

\[\begin{bmatrix}
  b_{00} & b_{01} & \cdots & \cdots \\
  b_{10} & b_{11} & \cdots & \cdots \\
  \vdots & \vdots & \ddots & \vdots \\
  \vdots & \vdots & \ddots & \vdots \\
\end{bmatrix}\]

\[\text{TILE\_SIZE}\]

Accessed elements in B: \( N \times \text{TILE\_SIZE} \)
Optimize locality and reduce branching overhead

Loop tiling: Reduce cache miss

\[ T_j = \text{TILE}_\text{SIZE} \]

for \( j_t \) in range(0, \( N, T_j \)):

for \( i \) in range(0, \( N \)):

for \( k \) in range(0, \( N \)):

for \( j \) in range(\( j_t, j_t + T_j \)):

\[ C[i][j] += A[i][k] \times B[k][j] \]

\[ T_j = T_k = \text{TILE}_\text{SIZE} \]

\[ \rightarrow \text{for } k_t \text{ in range}(0, \( N, T_k \)):\]

\[ \rightarrow \text{for } j_t \text{ in range}(0, \( N, T_j \)):\]

\[ \text{for } i \text{ in range}(0, \( N \)):\]

\[ \rightarrow \text{for } k \text{ in range}(k_t, k_t + T_k):\]

\[ \text{for } j \text{ in range}(j_t, j_t + T_j):\]

\[ C[i][j] += A[i][k] \times B[k][j] \]

\[ N \times \text{TILE}_\text{SIZE} \rightarrow \text{TILE}_\text{SIZE}^2 \]

\[ N^2 \rightarrow N \times \text{TILE}_\text{SIZE} \]
Optimize locality and reduce branching overhead
Loop tiling: Reduce cache miss

\[ T_j = T_k = \text{TILE\_SIZE} \]

for \( k_t \) in range \((0, N, T_k)\):
  for \( j_t \) in range \((0, N, T_j)\):
    for \( i \) in range \((0, N)\):
      for \( k \) in range \((k_t, k_t + T_k)\):
        for \( j \) in range \((j_t, j_t + T_j)\):
          \[ C[i][j] += A[i][k] \times B[k][j] \]

Accessed elements in \( A \)

\[ \text{TILE\_SIZE} \times \text{TILE\_SIZE} \]

\[ T_j = T_k = T_i = \text{TILE\_SIZE} \]

\[ \rightarrow \text{for } i_t \text{ in range}(0, N, T_i): \]
  for \( k_t \) in range \((0, N, T_k)\):
    for \( j_t \) in range \((0, N, T_j)\):
      \[ \rightarrow \text{for } i \text{ in range}(i_t, i_t + T_i): \]
      for \( k \) in range \((k_t, k_t + T_k)\):
        for \( j \) in range \((j_t, j_t + T_j)\):
          \[ C[i][j] += A[i][k] \times B[k][j] \]

\[ \text{N} \times \text{TILE\_SIZE} \rightarrow \text{TILE\_SIZE}^2 \]
Optimize locality and reduce branching overhead

Loop tiling: Reduce cache miss

for i in range(0, N):
    for k in range(0, N):
        for j in range(0, N):
            for i_t in range(0, N, T_i):
                for k_t in range(0, N, T_k):
                    for j_t in range(0, N, T_j):
                        C[i][j] += A[i][k] * B[k][j]

T_j = T_k = T_i = TILE_SIZE

for i_t in range(0, N, T_i):
    for k_t in range(0, N, T_k):
        for j_t in range(0, N, T_j):
            C[i][j] += A[i][k] * B[k][j]

Accessed elements in A: \(N^2 \rightarrow \text{TILE\_SIZE}^2\)
Accessed elements in B: \(N^2 \rightarrow \text{TILE\_SIZE}^2\)
Accessed elements in C: \(N^2 \rightarrow \text{TILE\_SIZE}^2\)

- The tile size can be determined according to the cache size
- Fitting accessed data into cache -> reuse data in cache -> cache miss
Optimize locality and reduce branching overhead
Loop tiling: Speed up Matrix Multiplication

```c
#define BLK_SIZE 32
void MatmulOperator::mat_mul_tiling(const struct matmul_params *params)
{
    int i, j, k;
    // ... Get A[][], B[][], C[][] from params

    for (ti = 0; ti < C->row; ti += BLK_SIZE){ // ... Tile i by BLK_SIZE
        for (tk = 0; tk < A->column; tk += BLK_SIZE){
            for (tj = 0; tj < C->column; tj += BLK_SIZE){
                for (i = ti; i < ti + BLK_SIZE; i++){
                    for (k = tk; k < tk + BLK_SIZE; k++){
                        Aik = data_A[i * A->column + k];
                        for (j = tj; j < tj + BLK_SIZE; j++){
                            data_C[i * C->column + j] += Aik * data_B[k * B->column + j];
                        }
                    }
                }
            }
        }
    }
}
```

Naive implementation: \textit{naive\_mat\_mul}: 24296 ms
Loop tiling: \textit{mat\_mul\_tiling}: 1269 ms
19x speed up
Optimize locality and reduce branching overhead

Loop Unrolling: Reduce branching overheads

- Overheads of loop control
  - Arithmetic operations for pointers (e.g., i, j, k)
  - End of loop test (e.g., k < N)
  - Branch prediction

- Reducing the overheads by loop unrolling
  - Replicate the loop body a number of times
  - Tradeoff between binary size and reduced overheads

```python
for i in range(0, N):
    for j in range(0, N):
        for k in range(0, N):
            C[i][j] += A[i][k] * B[k][j]
```

```
for i in range(0, N):
    for j in range(0, N):
        for k in range(0, N, 4): # step 1->4
            C[i][j] += A[i][k] * B[k][j]
```

- Arithmetic operations for pointers: $N^3 \rightarrow \frac{1}{4}N^3$
- Number of loop tests: $N^3 \rightarrow \frac{1}{4}N^3$
- Code size of the most inner loop: 1 -> 4

```python
for i in range(0, N):
    for j in range(0, N):
        for k in range(0, N, 4): # step 1->4
            C[i][j] += A[i][k] * B[k][j]
            C[i][j] += A[i][k+1] * B[k+1][j]
            C[i][j] += A[i][k+2] * B[k+2][j]
            C[i][j] += A[i][k+3] * B[k+3][j]
```
void MatmulOperator::mat_mul_unrolling(const struct matmul_params *params) {
    int i, j, k;
    // ... Get A[][], B[][], C[][] from params

    for (i = 0; i < C->row; i++){
        for (j = 0; j < C->column; j += 8){ // unroll j by 8
            float[8] acc = 0;
            for (k = 0; k < A->column; k += 4){ // unroll k by 4
                acc[0] += A[i][k] * B[k][j];
                acc[0] += A[i][k+1] * B[k+1][j];
                acc[0] += A[i][k+2] * B[k+2][j];
                acc[0] += A[i][k+3] * B[k+3][j];
                acc[1] += A[i][k] * B[k][j+1];
                acc[1] += A[i][k+1] * B[k+1][j+1];
                acc[1] += A[i][k+3] * B[k+3][j+1];
                // ...
            }
            C[i][j] = acc[0];
            C[i][j+1] = acc[1];
            // ...
        }
    }
}
Parallel Processing Techniques
Unlocking the Power of High Performance Computing

- Optimize locality and reduce branching overhead
  - Loop reordering
  - Loop tiling
  - Loop unrolling

- **SIMD (single instruction, multiple data) programming**
  - Multithreading
  - CUDA programming
  - Distributed Memory Programming
Recap: Instruction Set Architecture (ISA)

- An instruction set architecture acts as an interface between the software and the hardware.

- Examples of an instruction set:
  - ADD - Add two numbers together.
  -COMPARE - Compare numbers.
  - JUMP - Jump to a designated PC
  - JUMP IF - Conditional statement that jumps to a designated RAM address.
  - LOAD - Load information from RAM to the CPU.
  - STORE - Store information to RAM.
  - IN/OUT - I/O for a device, e.g., monitor.

So far, Single Instruction Single Data. Can we manipulate multiple data in one instruction?
SIMD Programming

• SIMD (Single Instruction Multiple Data)
  • A parallel processing paradigm that applies a single instruction to multiple data elements simultaneously.
  • Commonly used in modern processors to exploit data-level parallelism.
• Key Features:
  • Vector Registers:
    • Specialized registers that can hold and process multiple data elements.
  • Vector Operations:
    • Arithmetic and logical operations that work on entire vectors.
• With SIMD programming, we can
  • Increase computational throughput and speed
  • Improve energy efficiency

![Image showing difference between a conventional and SIMD instructions]
SIMD Programming
Example: SSE and NEON intrinsics

- **SSE**: `_mm_load_ps/_mm_mul_ps/_mm_add_ps`
  - mm: multimedia
  - load/mul/add: load/multiply/add
  - ps: packed single-precision

- **NEON**: `vld1q_f32/vmulq_f32/vaddq_f32`:
  - v: vector
  - ld/mul/add: load/multiply/add
  - 1: number of vector
  - q: quadword

---

### Arithmetic operations: $N$

- $B[k]$: 32-bit

- $C := A[k] \times B[k]$ (SISD programming)

### Arithmetic operations: $N/4$

- $A[k*4]$:
  - 128-bit

- $B[k*4]$
  - 128-bit

- $C := _{mm\_mul\_ps}( _{mm\_load\_ps}(A[k*4]), _{mm\_load\_ps}(B[k*4]))$ (with SSE)

- $C := \text{vmulq\_f32}(\text{vld1q\_f32}(A[k*4]), \text{vld1q\_f32}(B[k*4]))$ (with NEON)

---

// SISD programming
for $k$ in range(0, $N$):
  $C := A[k] \times B[k]$

// with SSE
for $k$ in range(0, $N/4$):
  $C := _{mm\_mul\_ps}( _{mm\_load\_ps}(A[k*4]), _{mm\_load\_ps}(B[k*4]))$

// with NEON
for $k$ in range(0, $N/4$):
  $C := \text{vmulq\_f32}(\text{vld1q\_f32}(A[k*4]), \text{vld1q\_f32}(B[k*4]))$
SIMD Programming
Matrix Multiplication on x86 and ARM

preprocessing(); // Initialize A, B, C and transpose B as transpose_tmp

for (i = 0; i < C->row; i++)
    for (j = 0; j < C->column; j++) {
        float accumulators[4] = {0, 0, 0, 0};
        __m128 *acc = (__m128*)accumulators; // initialize four 32-bit accumulators
        for (k = 0; k < A->column; k += 4) {
            __m128 val = _mm_mul_ps(_mm_load_ps(&A[i][k]), _mm_load_ps(&transpose_tmp[j][k]));
            // accumulators[0:4] = accumulators[0:4] + val[0:4];
            *acc = _mm_add_ps(*acc, val);
        }
    }

SSE intrinsics
_mm_load_ps/_mm_mul_ps/_mm_add_ps:
• mm: multimedia
• load/mul/add: load/multiply/add
• ps: packed single-precision

ARM NEON intrinsics
vld1q_f32/vmulq_f32/vaddq_f32:
• v: vector
• ld/mul/add: load/multiply/add
• 1: number of vector
• q: quadword

preprocessing(); // Initialize A, B, C and transpose B as transpose_tmp

for (i = 0; i < C->row; i++)
    for (j = 0; j < C->column; j++) {
        float accumulators[4] = {0, 0, 0, 0};
        float32x4_t *acc = (float32x4_t*)accumulators; // initialize four 32-bit accumulators
        for (k = 0; k < A->column; k += 4) {
            float32x4_t val = vmulq_f32(vld1q_f32(&A[i][k]), vld1q_f32(&transpose_tmp[j][k]));
            // accumulators[0:4] = accumulators[0:4] + val[0:4];
            *acc = vaddq_f32(*acc, val);
        }
    }
## SIMD Programming

### Loop tiling: Speed up Matrix Multiplication

```c
inline void simd_mul_fp_128(const float *a, const float *b, float *c){
    __m128 val = _mm_mul_ps(_mm_load_ps(a), _mm_load_ps(b));
    __m128 acc = _mm_add_ps(_mm_load_ps(c), val);
    _mm_store_ps(c, acc);
}

void MatmulOperator::mat_mul_transpose_simd(const struct matmul_params *params){
    int i, j, k;
    // ... Get A[], B[], C[] from params

    // Transpose the B
    for (i = 0; i < B->column; i++)
        for (j = 0; j < B->row; j++)
            transpose_tmp[i][j] = B[j][i];

    for (i = 0; i < C->row; i++)
        for (j = 0; j < C->column; j++)
            {
                float accumulators[4] = {};
                for (k = 0; k < A->column; k += 4)
                    simd_mul_fp_128(&A[i][k], &transpose_tmp[j][k], accumulators);
            }
}
```

**Naive implementation:** naive_mat_mul: 24296 ms

**Loop tiling:** mat_mul_transpose_simd: 4484 ms

5.4x speed up
Parallel Processing Techniques
Unlocking the Power of High Performance Computing

- Optimize locality and reduce branching overhead
  - Loop reordering
  - Loop tiling
  - Loop unrolling

- SIMD (single instruction, multiple data) programming

- Multithreading
  - CUDA programming
  - Distributed Memory Programming
Multithreading
Shared-Memory Programming

- Multithreading is the concurrent execution of multiple threads within a single process.
- A thread is the smallest unit of execution in a program.
- Threads share the same memory space and resources but have their own stack and program counter.
- Different threads can run on separate CPU cores, improving performance and allowing parallelism.

“What Is Multithreading In OS? Understanding The Details” [Link]
Multithreading
Benefits of Multithreading

- **Improved Performance:**
  - Multiple threads can execute tasks simultaneously, increasing overall program speed.

- **Responsiveness:**
  - A program can remain responsive to user input without blocking.

- **Resource Utilization:**
  - Threads can share resources, reducing the overhead of creating multiple processes.

- **Simplified Program Structure:**
  - Multithreading can help break down complex problems into simpler, smaller tasks.
Multithreading
Examples of Multithreading Techniques and Approaches

- **Pthreads:**
  - A C library for creating and managing POSIX threads.

- **OpenMP:**
  - An API for C, C++, and Fortran to support parallel programming using shared-memory model.

“PThreads Programming” [Link]; “OpenMP” [Link]
Example Code of Multithreading by Using Pthreads

```c
int main() {
    // Initiate the threads
    pthread_t threads[NUM_THREADS];
    ThreadData thread_data[NUM_THREADS];

    // Create threads and assign work
    for (int i = 0; i < NUM_THREADS; ++i) {
        thread_data[i].thread_id = i;
        pthread_create(&threads[i], nullptr, mat_mul_multithreading, &thread_data[i]);
    }

    // Join threads to wait for their completion
    for (int i = 0; i < NUM_THREADS; ++i) {
        pthread_join(threads[i], nullptr);
    }

    return 0;
}
```

```c
struct ThreadData {
    int thread_id;
};
```

```c
void* mat_mul_multithreading(void* arg) {
    ThreadData* data = static_cast<ThreadData*>(arg);
    int thread_id = data->thread_id;
    int rows_per_thread = SIZE_MATRIX / NUM_THREADS;

    // Indicate the starting and ending rows of each thread
    int start_row = thread_id * rows_per_thread;
    int end_row = (thread_id + 1) * rows_per_thread;

    // Each thread only conducts a part of the mat_mul
    for (int i = start_row; i < end_row; ++i) {
        for (int j = 0; j < SIZE_MATRIX; ++j) {
            for (int k = 0; k < SIZE_MATRIX; ++k) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }

    return nullptr;
}
```

Naive implementation on CPU: **naive_mat_mul: 24296 ms**

Naive multithreading (4 threads): **mat_mul_multithreading: 5864 ms**

4.1x speed up
Multithreading
Introduction of OpenMP

- Open Multi-Processing (OpenMP):
  - a shared-memory parallel programming model
  - API for C, C++, and Fortran
  - Portable across different platforms and operating systems

- OpenMP Compiler Directives:
  - #pragma omp parallel: create a parallel region
  - #pragma omp for: parallelize a for loop
  - #pragma omp sections: define parallel sections

“OpenMP” [Link]
Multithreading
Example Code of Multithreading by Using OpenMP

```cpp
int main() {
    const int N = 100; // Size of matrix

    // Initialize two matrices with random integers
    std::vector<std::vector<int>> A(N, std::vector<int>(N));
    std::vector<std::vector<int>> B(N, std::vector<int>(N));
    std::vector<std::vector<int>> C(N, std::vector<int>(N, 0));

    // Set the number of threads to use in the parallel region
    omp_set_num_threads(4);

    // Parallelize the loop with OpenMP
    #pragma omp parallel for
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < N; ++j) {
            for (int k = 0; k < N; ++k) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }

    return 0;
}
```

The code of using OpenMP is cleaner than that of using Pthreads
(Easy integration with existing code)

Indicate the number of threads

In OpenMP, we use #pragma to indicate where to parallelize

![Diagram of matrix multiplication showing parallelization with OpenMP]
Multithreading
Challenges and Issues with Multithreading

- **Thread Synchronization:**
  - Ensuring threads access shared resources in a coordinated manner to avoid conflicts.

- **Deadlocks:**
  - Situations where threads are blocked, waiting for resources held by other blocked threads.

- **Starvation:**
  - Occurs when a thread is unable to obtain the resources it needs due to other threads holding onto them.

- **Race Conditions:**
  - Undesirable situations when the behavior of a program depends on the relative timing of events.

“Lock it, Block it, but Don’t Deadlock it” [Link]; “Race Condition vs Data Races in Java” [Link]
Parallel Processing Techniques
Unlocking the Power of High Performance Computing

- Optimize locality and reduce branching overhead
  - Loop reordering
  - Loop tiling
  - Loop unrolling

- SIMD (single instruction, multiple data) programming

- Multithreading

- CUDA programming

- Distributed Memory Programming
CUDA Programming
Introduction

- The Graphics Processing Unit (GPU) provides much higher instruction throughput and memory bandwidth.

- CUDA is introduced by Nvidia in 2006 as a general purpose parallel computing platform and programming model that leverages the parallel compute engine in NVIDIA GPUs.

- CUDA is a C-like language to express programs that run on GPUs using the compute-mode hardware interface.

CUDA Programming

Hierarchy of CUDA Threads

- Thread IDs can be up to 3-dimensional (2D example below)
- Multi-dimensional thread ids are convenient for problems that are naturally N-D

```cpp
const int Nx = 12;
const int Ny = 6;

dim3 threadsPerBlock(4, 3);
dim3 numBlocks(Nx/threadsPerBlock.x, Ny/threadsPerBlock.y);
// assume A, B, C are allocated Nx x Ny float arrays

// this call will launch 72 CUDA threads:
// 6 thread blocks of 12 threads each
matrixAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
```
CUDA Programming
Programming Model

const int Nx = 12;
const int Ny = 6;
dim3 threadsPerBlock(4, 3);
dim3 numBlocks(Nx/threadsPerBlock.x, Ny/threadsPerBlock.y);

// assume A, B, C are allocated Nx x Ny float arrays

// this call will launch 72 CUDA threads:
// 6 thread blocks of 12 threads each
matrixAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);

// kernel definition (runs on GPU)
__global__ void matrixAdd(float A[Ny][Nx], float B[Ny][Nx],
float C[Ny][Nx])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    C[j][i] = A[j][i] + B[j][i];
}

Host code runs on CPU: Serial execution
Bulk launch of many CUDA threads “launch a grid of CUDA thread blocks”
Kernel code runs on CUDA device: Parallel execution
Each thread computes its overall grid thread id from its position in its block (threadIdx) and its block’s position in the grid (blockIdx)
CUDA Programming

Memory Model

- Distinct host and device address spaces
- Data can be moved between address spaces

```c
float* A = new float[N]; // allocate buffer in host memory

// populate host address space pointer A
for (int i=0; i<N; i++)
    A[i] = (float)i;

int bytes = sizeof(float) * N;

float* deviceA;
cudaMalloc(&deviceA, bytes) // Allocate buffer in device address space

// populate deviceA
cudaMemcpy(deviceA, A, bytes, cudaMemcpyHostToDevice);
```

Access `deviceA[i]` from host will be invalid (not host address space)
CUDA Programming
Memory Model

- Three distinct types of address spaces visible to kernels
  - Private memory (per-thread), shared memory (per-block), global memory
  - Different address spaces -> different locality -> different load/store overheads

![Diagram showing CUDA memory model]

- Per-block shared memory
  - Readable/writable by all threads in block
- Per-thread private memory
  - Readable/writable by thread
- Device global memory
  - Readable/writable by all threads
CUDA Programming
Speed up Matrix Multiplication

```c
__global__ void matrixMultiplyShared(const float *A, const float *B, float *C, int A_row, int A_column, int B_column) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    __shared__ float As[TILE_SIZE][TILE_SIZE]; // per-block allocation for tiling A
    __shared__ float Bs[TILE_SIZE][TILE_SIZE]; // per-block allocation for tiling B

    float value = 0;

    for (int i = 0; i < A_column / TILE_SIZE; i++) { // per-block allocation for tiling A
        As[threadIdx.y][threadIdx.x] = A[(blockIdx.y * TILE_SIZE + threadIdx.y) * A_column + TILE_SIZE * i + threadIdx.x];
        Bs[threadIdx.y][threadIdx.x] = B[(i * TILE_SIZE + threadIdx.y) * B_column + blockIdx.x * TILE_SIZE + threadIdx.x];
    }

    __syncthreads(); // Wait for memory loading

    for (int k = 0; k < TILE_SIZE; k++)
        value += As[threadIdx.y][k] * Bs[k][threadIdx.x];

    __syncthreads(); // Wait for multiplication and accumulation
}

C[row * B_column + col] = value;
```

---

Naive implementation on CPU:

```
naive_mat_mul: 24296 ms
```

CUDA implementation on 2080Ti:

```
cuda kernel: 6.796 ms
mat_mul_cudpp: 258 ms
94x end-to-end speed up
```
CUDA Programming on Tensor Cores
Motivation: Higher throughput and more data types

A CUDA core performs 1 FP32 / 2 FP16 multiply-accumulates (MACs) in each cycle; in contrast, a tensor core can finish an entire matrix multiplication (4x4x4 for Turing; 8x4x8 for Ampere) in FP16 each cycle.

- FP32 (CUDA Core)
- TF32 (Tensor Core)
- FP16 (Tensor Core)
- INT8/FP8/Sparse FP16 (Tensor Core)

TF32 is only available in tensor cores. FP8, INT8 and FP16 computation is much faster in tensor cores.

60x higher throughput

H100 TFLOPS

3,958
1,979
989
67
CUDA Programming on Tensor Cores
Motivation: Higher throughput and more data types

Tensor cores are much faster than CUDA cores for $N \times N \times N$ matrix multiplication workloads when $N$ is large.
Matrix multiplication intrinsics (MMA)
A single instruction for MMA on 16x8x16 tiles

float D[4];
uint32_t const A[4];
uint32_t const B[2];
float const C[4];

// Example targets 16-by-8-by-16 Tensor Core operation
asm(
    "mma.sync.aligned.m16n8k16.row.col.f32.f16.f16.f32 "
    "{ %0, %1, %2, %3 },"
    "{ %4, %5, %6, %7 },"
    "{ %8, %9 },"
    "{ %10, %11, %12, %13 };"
    : 
    "=f"(D[0]), "=f"(D[1]), "=f"(D[2]), "=f"(D[3])
    :
    "r"(B[0]), "r"(B[1]),
    "f"(C[0]), "f"(C[1]), "f"(C[2]), "f"(C[3])
);
Step 1. Break down input/output operands to 16x16 and 16x8 tiles. This results in two tiles for operand A, four tiles for operand B and two tiles for operand C.
**Matrix multiplication intrinsics (MMA)**

Compute 16x16x32 MMA using 16x8x16 intrinsics

**Step 2.** Launch mma instruction between tile 0 of A and tile 0 of B; accumulate to output tile 0.
Matrix multiplication intrinsics (MMA)
Compute 16x16x32 MMA using 16x8x16 intrinsics

Step 3. Launch mma instruction between tile 1 of A and tile 2 of B; accumulate to output tile 0.
Matrix multiplication intrinsics (MMA)

Compute 16x16x32 MMA using 16x8x16 intrinsics

**Step 4.** mma instruction between tile 0 of A and tile 1 of B; accumulate to output tile 1.
**Matrix multiplication intrinsics (MMA)**

Compute 16x16x32 MMA using 16x8x16 intrinsics

---

**Step 5** mma instruction between tile 1 of A and tile 3 of B; accumulate to output tile 1.
Matrix multiplication intrinsics (MMA)
Compute 16x16x32 MMA using 16x8x16 intrinsics

Tile 0 (16x16)  Tile 1 (16x16)  Tile 0 (16x8)  Tile 1 (16x8)
Tile 2 (16x8)  Tile 3 (16x8)

Note: The order of applying these four MMA intrinsics does not matter.
Parallel Processing Techniques
Unlocking the Power of High Performance Computing

- Optimize locality and reduce branching overhead
  - Loop reordering
  - Loop tiling
  - Loop unrolling

- SIMD (single instruction, multiple data) programming

- Multithreading

- CUDA programming

- Distributed Memory Programming
Distributed Memory Programming

Why do we need distributed programming?

• Distributed memory programming allows each process to have its private memory space, and data is shared through explicit message passing between processes.

• **Advantages:**
  • Scalability: Easily applicable to large-scale parallel systems.
  • Flexibility: Supports complex and irregular data structures.

• **Challenges:**
  • Communication overhead: Can limit performance if not managed efficiently.
  • Load balancing: Ensuring equal distribution of work among processes.

• **Common applications:** High-performance computing, Large-scale simulations (e.g., climate, physics, etc.), big data processing and analytics.
Distributed Memory Programming
Example: Distributed Matrix Multiplication with OPEN MPI

```c
int main(int argc, char** arg){
    int N = 1024, rank, size;
    double *A, *B, *C;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    //get the size of nodes
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    //get the rank of the current node
    if (rank == 0){
        //initialize A, B, C on the root node …
    }
    int local_N = N / size;
    double *local_A = (double *)malloc(local_N * N * sizeof(double));
    double *local_C = (double *)malloc(local_N * N * sizeof(double));
    MPI_Scatter(A, local_N * N, MPI_DOUBLE, local_A, local_N * N, MPI_DOUBLE, 0,
                MPI_COMM_WORLD);
    MPI_Bcast(B, N * N, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    matrix_mult(local_A, B, local_C, N);
    // compute partial output (local_C)
    MPI_Gather(local_C, local_N * N, MPI_DOUBLE, C, local_N * N, MPI_DOUBLE, 0,
               MPI_COMM_WORLD);
    //free memory…
}
```

MPI_Comm_size() gets the number of total nodes
MPI_Comm_rank() gets the rank of the current nodes, i.e., node id

Each node allocate memory locally for partial input and output

MPI_Scatter to distribute matrix A among the processes in the order of their rank
and MPI_Bcast to broadcast matrix B to all processes

MPI_Gather to collect the local_C matrices from all processes and combine them into the full C matrix on the root process.
Summary

- **Loop tiling/reordering/unrolling**
  - Optimize locality and reduce branching overhead
- **SIMD (single instruction, multiple data) programming**
  - Data parallelism that allows a single instruction to multiple data elements simultaneously.
- **Multithreading**
  - Threads share the same memory space but have their own stack and PC.
  - Improve performance and resource utilization with intra-process concurrency.
- **CUDA programming**
  - Threads and Blocks
  - Private memory (per-thread), shared memory (per-block), global memory
- **Distributed Memory Programming**
  - Message passing between processes
  - Challenging communication and synchronization overhead
Thank you!

Next Lecture: Synchronization