Lecture 20
Parallel Processing

Parallel Processing Overview

- What is parallel processing?
- Why parallel processing:
  - System constraints:
    - Slowing down of Moore's Law
  - Application requirements:
    - Apps process large datasets within/across machines
    - Cloud services, IoT applications, ML models
- Granularities & frameworks for parallel processing:
  - ILP → Superscalar execution
  - DLP → SIMD
  - TLP → OpenMP, pthreads, etc.
  - Etc.

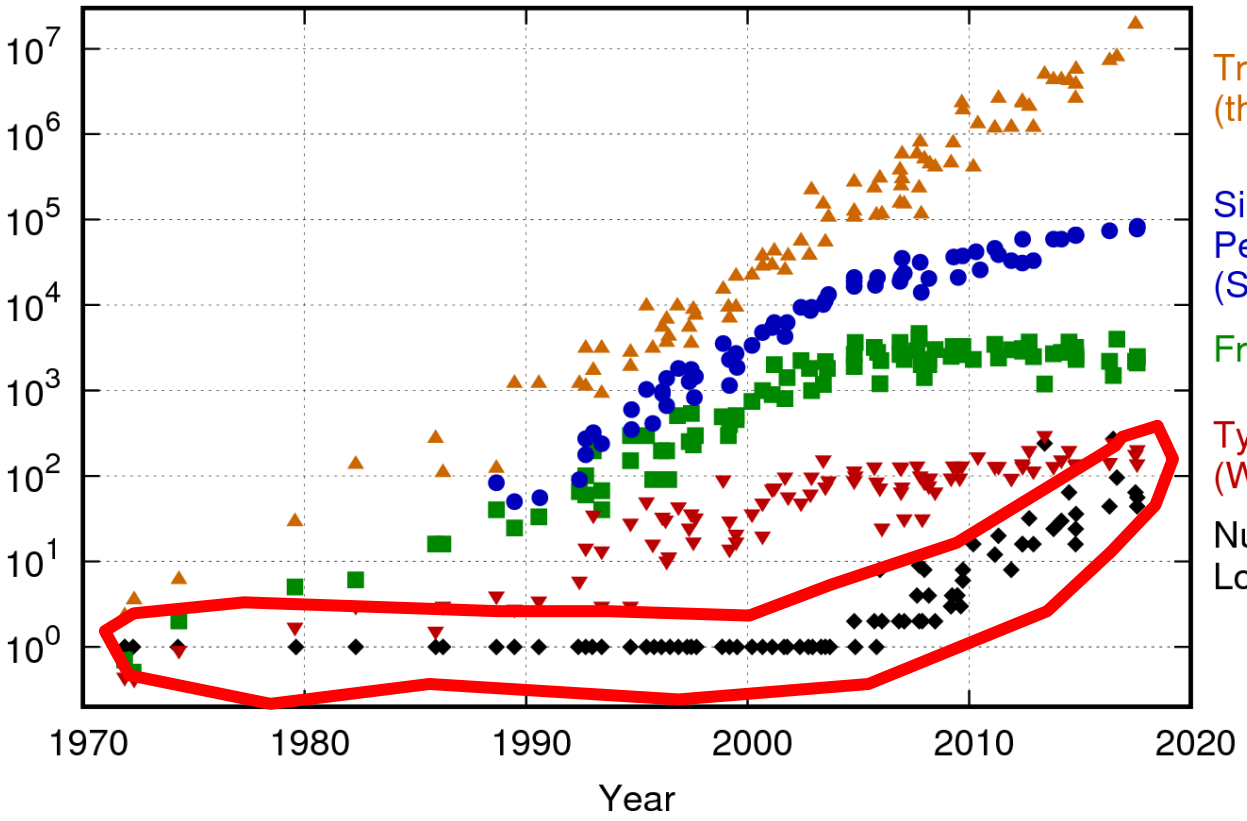
What is Parallel Processing?

- A computational approach that divides a problem into smaller tasks which can then be processed in parallel across multiple processing units (ALUs, CPUs, GPUs, servers)

- Very general concept that can be applied across different granularities
  - Instruction level parallelism
  - Data level parallelism
  - Thread level parallelism
  - Task/Request level parallelism

- Benefit: Enables solving larger problems, maximizes resource utilization, improves performance

Reminder: Moore's Law



Need frame-works that effectively use multiple cores

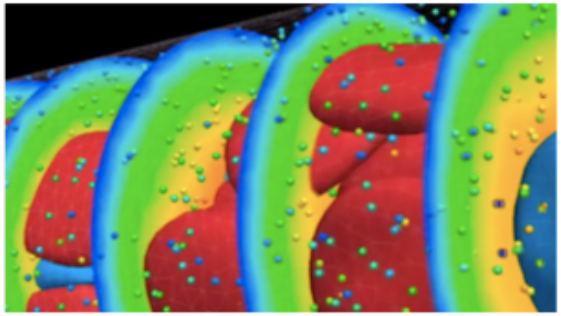
Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp [https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/]

- But Moore's Law is waning; reaching physical limits!
- Has motivated multicore processor designs

Application Requirements Increased

- Programs have evolved from simple computations to complex, large-scale applications

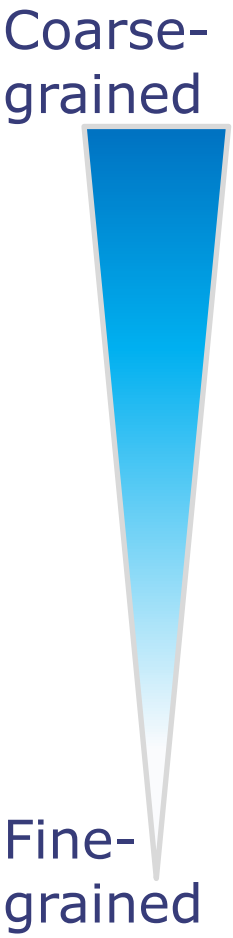
- Scientific simulations (climate modeling, molecular dynamics)
- ML and generative AI (training deep neural networks, ChatGPT)
- Distributed cloud applications (websearch, social networks, databases, etc.)
- Image and video processing (rendering, compression, etc.)



"10 Award-Winning Scientific Simulation Videos" [Link]; "YOLOv5 Is Here" [Link]; "Artificial Intelligence in Healthcare" [Link]

Parallel Processing Granularities

- Different granularities depending on the unit that is processed in parallel



Task/Request Level Parallelism (TLP/RLP)	Independent tasks or user requests
Thread Level Parallelism (TLP)	Same functionality across threads
Data Level Parallelism (DLP)	Same operation across data chunks
Instruction Level Parallelism (ILP)	Independent instructions

Parallel Processing Techniques

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

- Superscalar execution → ILP
- SIMD (single instruction, multiple data) programming → DLP
- Multithreading, CUDA programming → TLP
- Distributed memory programming
- Distributed data analytics frameworks
- Online interactive services

} TLP/RLP

Parallel Processing Techniques

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

- Superscalar execution → ILP
- SIMD (single instruction, multiple data) programming → DLP
- Multithreading, CUDA programming → TLP
- Distributed memory programming
- Distributed data analytics frameworks
- Online interactive services

} TLP/RLP

What are Locality and Branching?

- Locality: reuse of same or neighboring addresses by a program
  - Temporal locality: reuse of the same memory location (instructions or data)
  - Spatial locality: use of neighboring memory locations (instructions or data)
  - Loop reordering, loop tiling
- Branching: mispredicted branches are very expensive in performance and resources → worse in modern processors
  - Compiler optimizations: reorder code to minimize branches
  - Loop unrolling

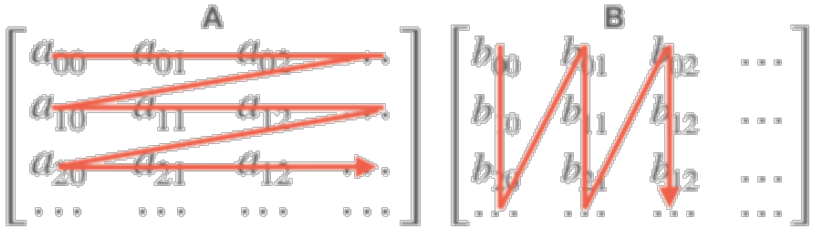
Loop Reordering

- Caches designed to take advantage of data locality
  - Cache lines > 1 word (amortize cost of main memory access)
- However, the code is not always structured to take advantage of the program's inherent locality
- Loop Reordering: change the order of loop iterations to leverage cache's spatial locality → Reduce cache misses
  - e.g., i, j, k -> i, k, j

```
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]
```

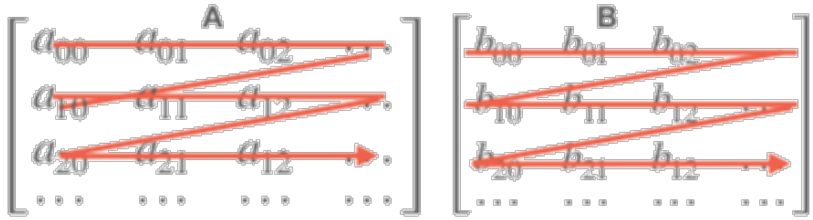
Reorder (i, j, k) as (i, k, j)

```
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]
```



Good data locality Poor data locality!

* Assume stored in row-major order



Good data locality

* Assume stored in row-major order

Loop Reordering

- Matrix multiplication before and after loop reordering

```
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;
    }
}
```

Without Loop reordering:

```
naive_mat_mul: 24296 ms
```

```
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;
    }
}
```

With Loop reordering:

```
mat_mul_reordering: 1979 ms
```

12x speed up

*Results are measured on Intel Xeon 4114

Loop Tiling 1D

- What if the data is much larger than the cache size?
  - Data in cache will be evicted before reuse -> cache miss ↑
  - e.g., B is much larger than cache size
- Loop tiling: Partition the loop iteration space
  - Fit elements accessed in the loop into cache
  - 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]
```

$$\begin{bmatrix} b_{00} & b_{01} & \cdots & \cdots \\ b_{10} & b_{11} & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \end{bmatrix}$$

Accessed elements in B = N^2

Tile the loop of j

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

N

$$\begin{bmatrix} b_{00} & b_{01} & \cdots & \cdots \\ b_{10} & b_{11} & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \end{bmatrix}$$

TILE_SIZE

Accessed elements in B: N x TILE_SIZE

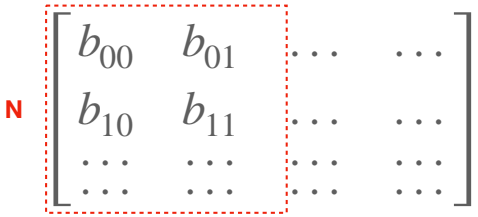
Loop Tiling 2D

```
T_j = TILE_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] * B[k][j]
```

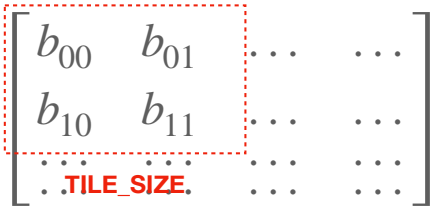
Tile the loop of k

```
T_j = T_k = 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] * B[k][j]
```

Accessed elements in B

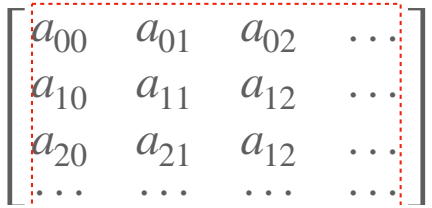


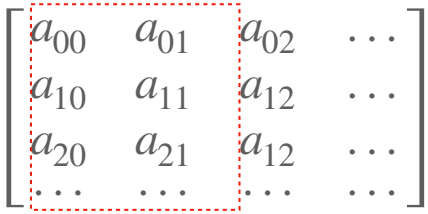
TILE_SIZE



N x TILE_SIZE → TILE_SIZE²

Accessed elements in A





N² → N x TILE_SIZE

Loop Tiling 2D

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

Tile the loop of i

```
Tj = Tk = Ti = TILE_SIZE
→ for i_t in range(0, N, Ti):
    for k_t in range(0, N, Tk):
        for j_t in range(0, N, Tj):
        → for i in range(i_t, i_t + Ti):
            for k in range(k_t, k_t + Tk):
                for j in range(j_t, j_t + Tj):
                    C[i][j] += A[i][k] * B[k][j]
```

Accessed elements in A

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & \cdots \\ a_{10} & a_{11} & a_{12} & \cdots \\ a_{20} & a_{21} & a_{12} & \cdots \\ \cdots & \cdots & \cdots & \cdots \end{bmatrix}$$

TILE_SIZE

TILE_SIZE

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & \cdots \\ a_{10} & a_{11} & a_{12} & \cdots \\ a_{20} & a_{21} & a_{12} & \cdots \\ \cdots & \cdots & \cdots & \cdots \end{bmatrix}$$

N x TILE_SIZE → TILE_SIZE²

Loop Tiling 3D

```
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]
```

$T_j = T_k = T_i = TILE\_SIZE$

```
for i_t in range(0, N, Ti):
    for k_t in range(0, N, Tk):
        for j_t in range(0, N, Tj):
            for i in range(i_t, i_t + Ti):
                for k in range(k_t, k_t + Tk):
                    for j in range(j_t, j_t + Tj):
                        C[i][j] += A[i][k] * B[k][j]
```

Accessed elements in A: $N^2 \rightarrow TILE\_SIZE^2$

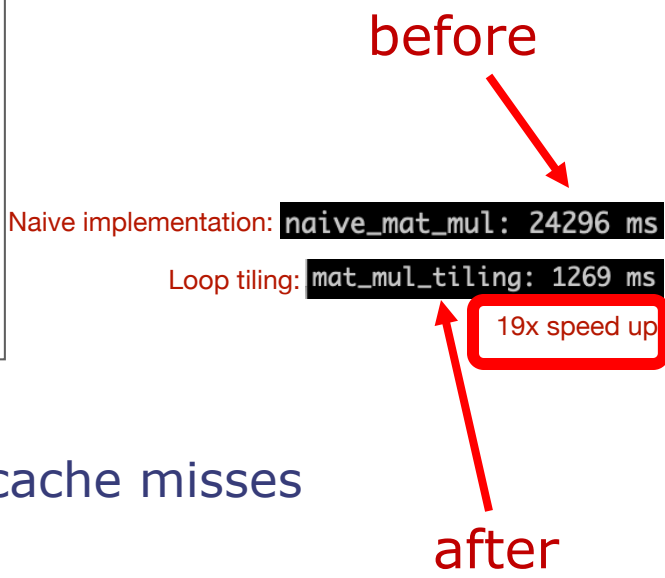
Accessed elements in B: $N^2 \rightarrow TILE\_SIZE^2$

Accessed elements in C: $N^2 \rightarrow TILE\_SIZE^2$

- The tile size is a function of the cache size
- Tiling improves data reuse → reduces data cache misses

Loop Tiling 3D

```
#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];
            }
          }
        }
      }
    }
  }
}
```



- The tile size is a function of the cache size
- Tiling improves data reuse → reduces data cache misses

Loop Unrolling: Reduce branching

- Branch mispredictions → poor performance, wasted resources
- Loops have several overheads:
  - Pointer bumps (increment i, j, k)
  - End of loop test (e.g., k < N)
  - Branch prediction
- Loop unrolling reduces these overheads:
  - Execute more iterations per loop
  - Fewer branch predictions → fewer mispredictions
  - Trade-off between binary size and loop overheads

```
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]
```

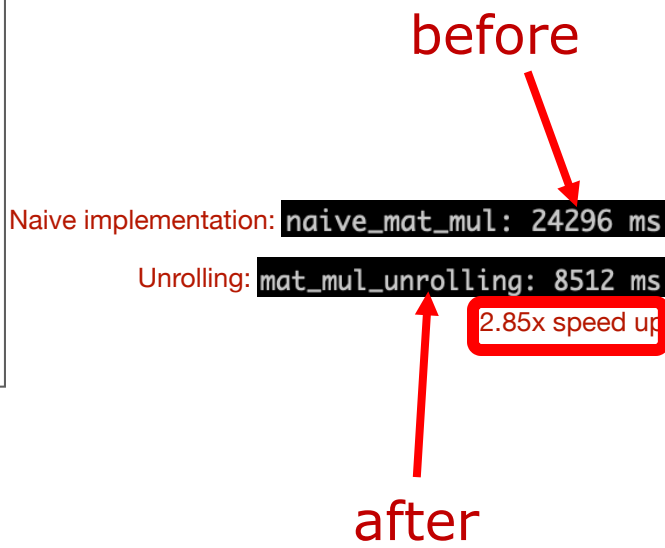
e.g., unroll by 4

```
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]
```

- Arithmetic operations for pointers: N^3 -> $1/4N^3$
- Number of loop tests: N^3 -> $1/4N^3$
- Code size of the most inner loop: 1 -> 4

Loop Unrolling: Reduce branching

```
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+2] * B[k+2][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

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

- **Superscalar execution → ILP**
- SIMD (single instruction, multiple data) programming → DLP
- Multithreading, CUDA programming → TLP
- Distributed memory programming
- Distributed data analytics frameworks
- Online interactive services

} TLP/RLP

Superscalar Execution (ILP)

- Compilers can reorder instructions to increase ILP
- Processors we studied so far → CPI >=1 (even with perfect pipelining)
- Superscalar processors enable CPI < 1 by executing multiple instructions in parallel → wider pipeline
- Duplicate parts of the pipeline:
  - Fetch multiple instructions per cycle
  - Decode multiple instructions per cycle
  - Execute multiple instructions per cycle
  - Writeback multiple instructions per cycle

Execution in Superscalar Processor

```
addi x11, x10, 2
lw x13, 8(x14)
sub x15, x16, x17
xor x19, x20, x21
add x22, x23, x24
addi x25, x26, 1
```

Cycles →

	1	2	3	4	5	6	7
IF	addi lw	sub xor	add addi				
DEC		addi lw	sub xor	add addi			
EXE			addi lw	sub xor	add addi		
MEM				addi lw	sub xor	add addi	
WB					addi lw	sub xor	add addi

Superscalar Execution (ILP)

- Downsides of superscalar execution

- Pipeline hazards are much more expensive:
  - More data hazards
  - Branch mispredictions are most costly → annul more instructions in flight

- Hardware support (bypassing & branch prediction) and compiler techniques to address them
  - Bypassing also gets more expensive
  - 6.590: Advanced Computer Architecture

Parallel Processing Techniques

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

- Superscalar execution → ILP
- **SIMD (single instruction, multiple data) programming → DLP**
- Multithreading, CUDA programming → TLP
- Distributed memory programming
- Distributed data analytics frameworks
- Online interactive services

} TLP/RLP

Single Instruction Multiple Data (DLP)

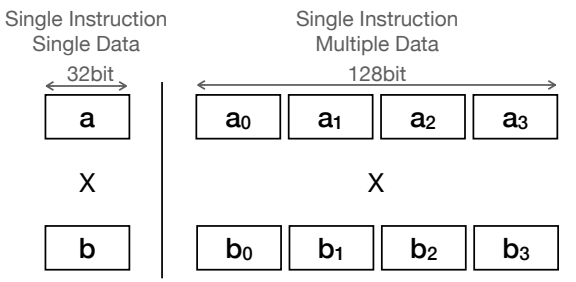
- So far in our ISA a single instruction manipulates a single piece of data (arithmetic, memory, etc.)
- Many application classes apply the same operations on different data chunks:
  - ML, image/video processing, graphics, gaming, etc.
- SIMD (Single instruction multiple data)
  - Apply the same operation on multiple data simultaneously (in parallel)
  - Common extension in modern processors

Single Instruction Multiple Data (DLP)

- SIMD (Single instruction multiple data)
  - Apply the same operation on multiple data elements simultaneously (in parallel)

- 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 instead of a single data element

- Improves throughput and latency

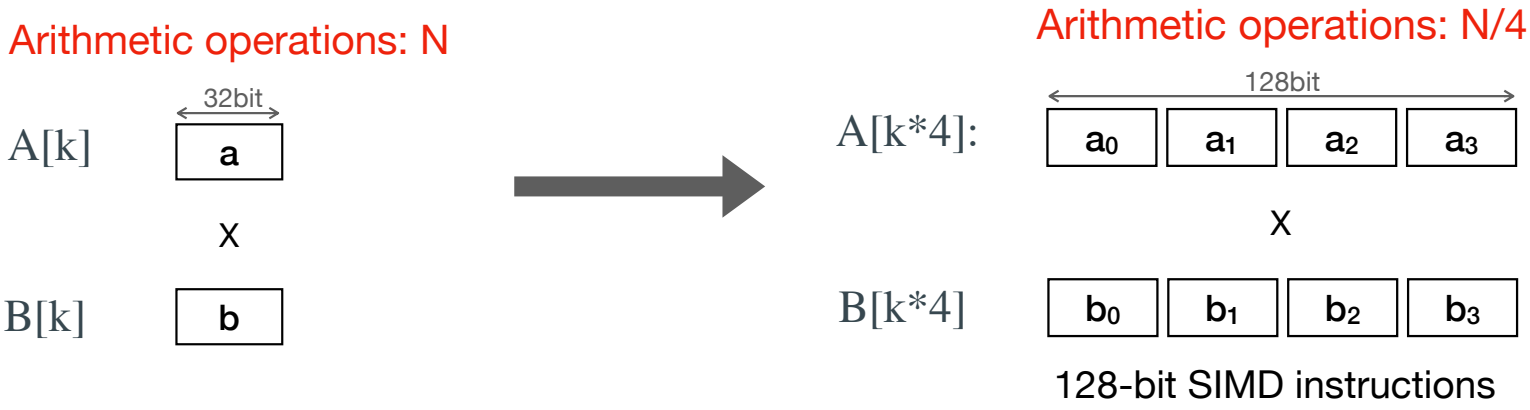


Difference between a conventional and SIMD instructions

Single Instruction Multiple Data (DLP)

- SSE extensions:
  - _mm_load_ps/_mm_mul_ps/_mm_add_ps
  - mm: multimedia
  - Load/mul/add: load/multiple/add
  - Ps: packed single-precision
- NEON: similar extension for ARM

```
// SISD programming
for k in range(0, N):
    C += A[k] * 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]))
```



Single Instruction Multiple Data (DLP)

```
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);
            C[i][j] = accumulators[0] + accumulators[1] + accumulators[2] + accumulators[3];
        }
}
```

before

Naive implementation: `naive_mat_mul: 24296 ms`

Loop tiling: `mat_mul_transpose_simd: 4484 ms`

5.4x speed up

after

Parallel Processing Techniques

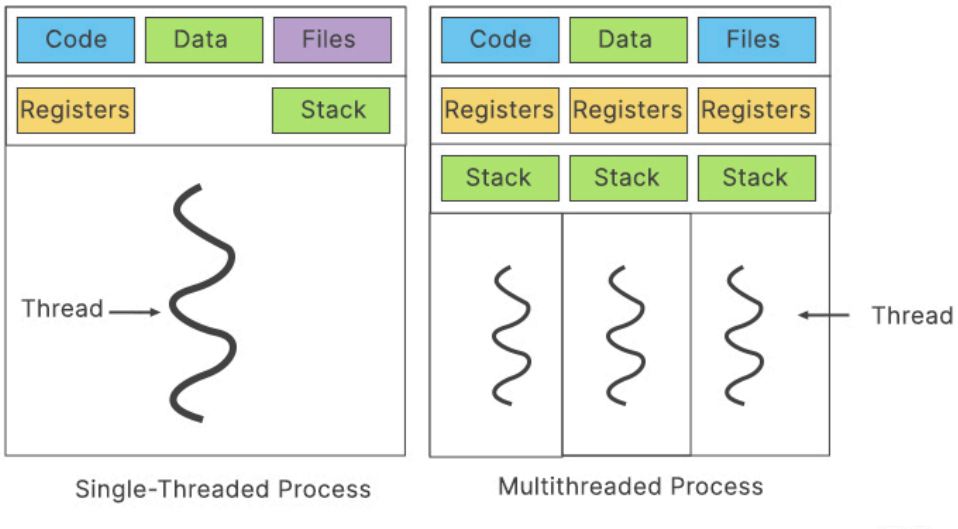
- Optimize locality and reduce branching overhead
  - Loop reordering
  - Loop tiling
  - Loop unrolling

- Superscalar execution → ILP
- SIMD (single instruction, multiple data) programming → DLP
- **Multithreading, CUDA programming → TLP**
- Distributed memory programming
- Distributed data analytics frameworks
- Online interactive services

} TLP/RLP

Multithreading (TLP)

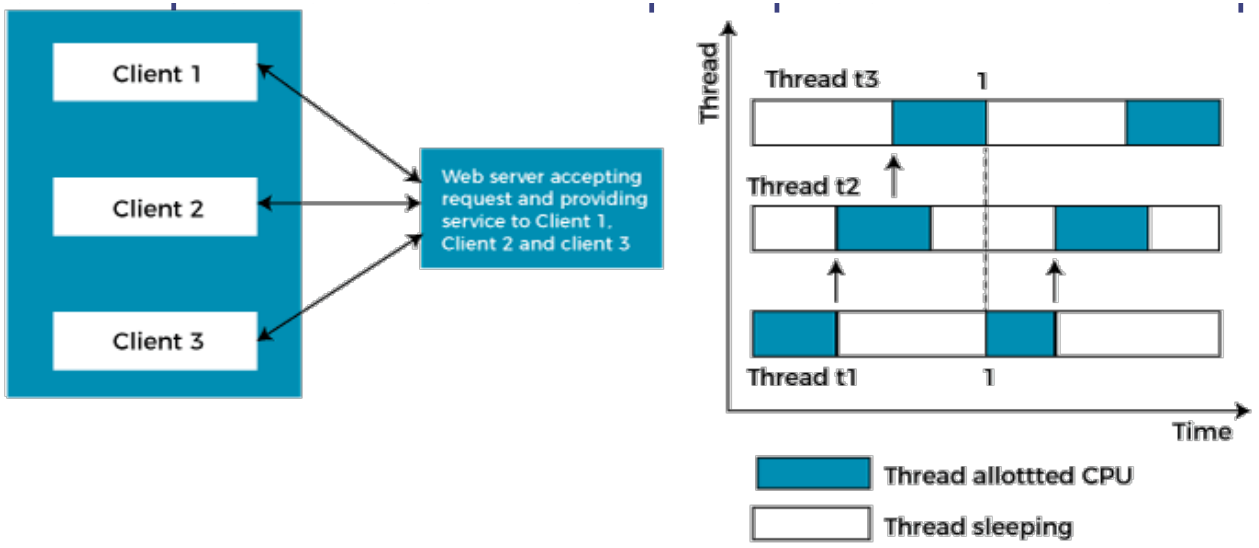
- Multithreading is the concurrent execution of multiple threads within the same process
- A thread is the smallest unit of execution in a program
- Threads share the same memory space and resources but have their own stack and program counter
- Different threads can run on different CPUs, which improves performance and better utilizes available resources



"What Is Multithreading In OS? Understanding The Details" [Link]

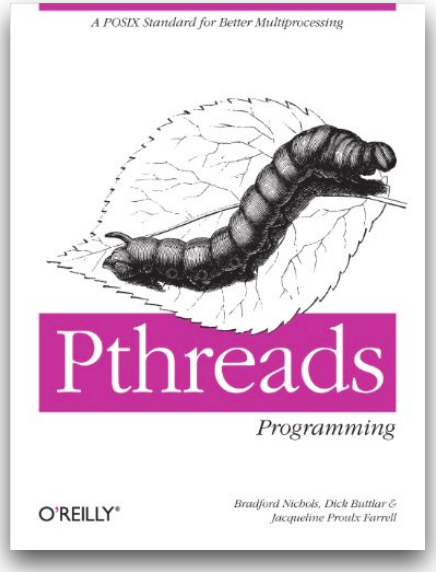
Benefits of Multithreading

- Performance: Independent operations are divided across threads which then work simultaneously
- Responsiveness: A program can remain responsive to user input without blocking until the current request is done
- Resource utilization: Threads share resources, reducing the overhead of creating multiple processes
- Simplified program structure: Multithreading breaks down complex problems into smaller tasks



Examples of Multithreading Frameworks

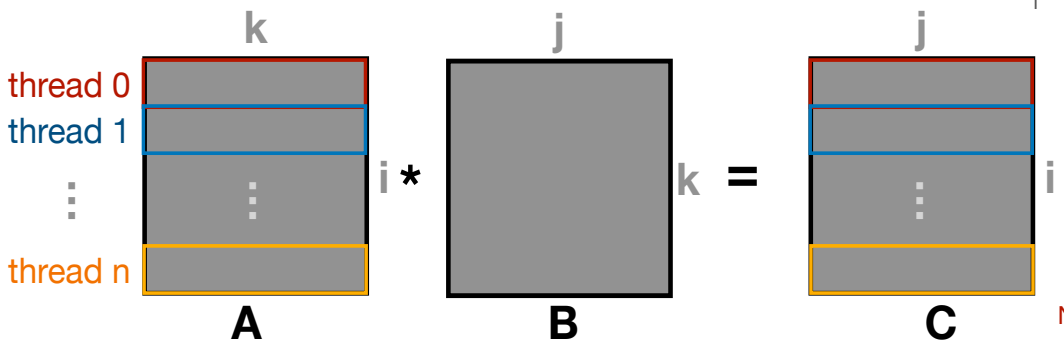
- Pthreads:
  - a C library for creating and managing POSIX threads
- OpenMP:
  - an API for C, C++, and Fortran to support parallel programming using a shared-memory model



Pthread example: Matrix Multiplication

```
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_multithr
eading,&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;
}
```

```
struct ThreadData {
    int thread_id;
};

// Specify the function that each thread needs to do
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
       threads
    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_multithreadin: 5864 ms

OpenMP

- Open Multi-Processing (OpenMP):
  - Shared-memory parallel programming model
  - API for C, C++, and Fortran
  - Portable across platforms and OS
  - Easy integration with existing code
  - Good scalability as #CPUs increases

- OpenMP compiler directives:
  - #pragma omp parallel: create a parallel region
  - #pragma omp for: parallelize a for loop
  - #pragma omp sections: define parallel sections

- Manage threads and synchronization:
  - #pragma omp single, critical, barrier
  - Omp_set_num_threads(), omp_get_num_threads(), etc.

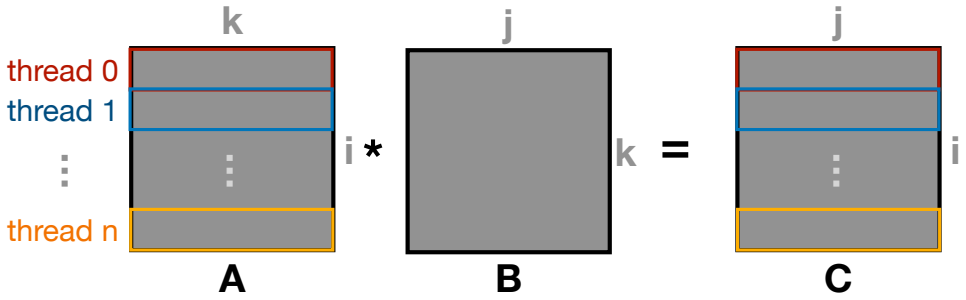
OpenMP

```
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)

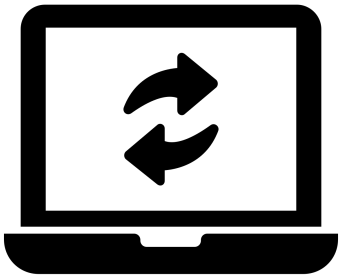
Indicate the number of threads

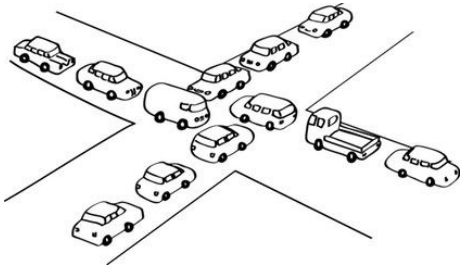
In OpenMP, we use #pragma to indicate where to parallelize

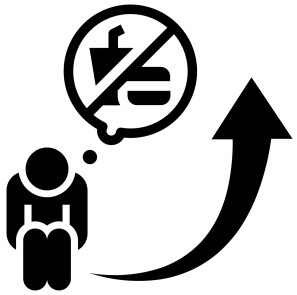


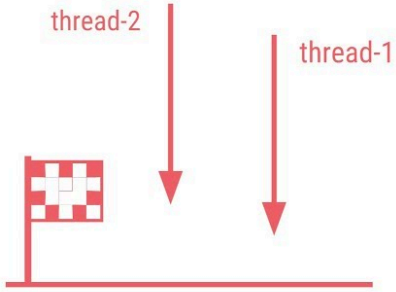
OpenMP Issues

- Thread synchronization:
  - Ensuring threads access shared resources in a coordinated manner to avoid conflicts
- Deadlocks:
  - Threads are blocked, waiting for resources held by other blocked threads
- Starvation:
  - A thread is unable to obtain resources due to other threads holding on to them
- Race conditions:
  - Undesirable situation when the program behavior depends on the relative timing of events
- Load imbalance:
  - Situation where the work is not evenly divided among threads









"Lock it, Block it, but Don't Deadlock it" [Link]; "Race Condition vs Data Races in Java" [Link]

Multithreading on GPUs

- pThreads and OpenMP primarily used on CPUs
- GPUs have their own parallel programming languages
- CUDA from NVIDIA (introduced in 2006) is the most popular parallel programming framework for GPUs
  - C like language to express computation mapped on GPUs

Parallel Processing Techniques

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

- Superscalar execution → ILP
- SIMD (single instruction, multiple data) programming → DLP
- Multithreading, CUDA programming → TLP
- **Distributed memory programming**
- Distributed data analytics frameworks
- Online interactive services

} TLP/RLP

Distributed Memory Programming

- Distributed Memory Programming allows each process to have its own private memory address space, and data is shared explicitly through messages

- Advantages:
  - Scalability: easily applicable to large-scale systems
  - Flexibility: supports complex and irregular data structures

- Challenges:
  - Communication overhead: can limit performance is not managed correctly
  - Load balancing: need to ensure fair distribution of work among processes

- Common applications: high-performance computing (HPC), large-scale simulations (e.g., climate forecasting), big data analytics

Summary

- Loop reordering, tiling, and loop unrolling improve cache locality, and reduce the overhead of branch mispredictions and loop management
- Parallelism allows computation to execute simultaneously improving performance
- Different granularities of parallelism (ILP, DLP, TLP, TLP/RLP)
- Different techniques/programming frameworks to take advantage of each
  - Superscalar execution (ILP)
  - SIMD (DLP)
  - Multithreading (TLP)
  - Distributed memory programming/Data analytics (TLP/RLP)

Thank you!

Next Lecture: Synchronization