Contents

11 Parallel Patterns in OpenMP and CUDA

11.1 OpenMP

OpenMP provides efficient implementations of many parallel programming patterns. Here are some of the key patterns that we have already discussed:

1. **Map Pattern on OpenMP**

   Characteristics

   - **No dependencies** between operations.
   - Simple and **ideal for SIMD** (Single Instruction, Multiple Data) execution.
   - Executes independently on different data elements.

   Example 1: Map Pattern on OpenMP

   ```
   #pragma omp parallel for
   for (int i = 0; i < N; i++) {
       result[i] = operation(data[i]);
   }
   ```

2. **Reduction Pattern on OpenMP**

   Characteristics

   - **Combines results from multiple threads into a single value** using a specified operation (e.g., sum, min, max).
   - **Supported natively** in OpenMP (see 5.3.1.1, page 66). It is an easy integration for operations that aggregate data across threads.

   Example 2: Reduction Pattern on OpenMP

   ```
   #pragma omp parallel for reduction(+:sum)
   for (int i = 0; i < N; i++) {
       sum += array[i];
   }
   ```

3. **Workpile Pattern on OpenMP**

   Characteristics

   - **Handles irregular work distribution that may change dynamically** at runtime.
   - Assumes **all tasks are independent**.
   - Commonly used in applications like **tree searches** or **recursive computations**.
   - Leverages OpenMP's tasking feature for dynamically generated workloads.

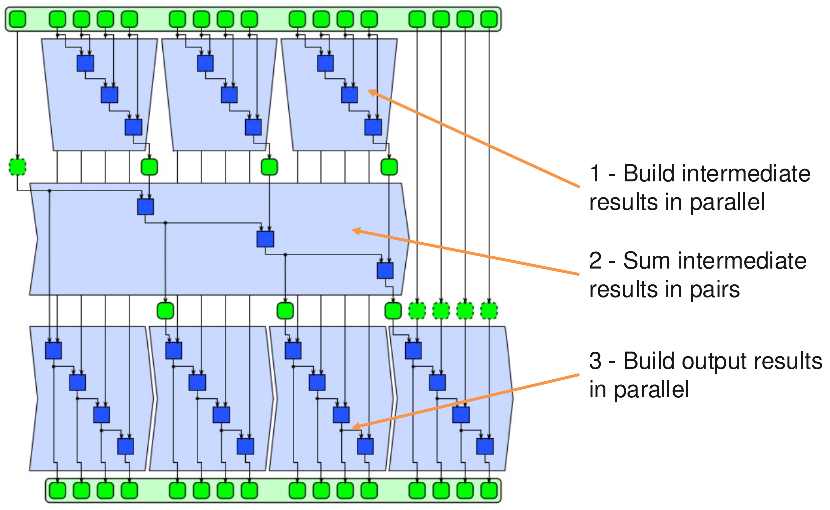
Example 3: Workpile Pattern on OpenMP

```
#pragma omp task
void process_task() {
    // Perform a unit of work
}
```

4. **Scan Pattern on OpenMP**

Characteristics

- **Efficiently performs prefix-sum** or similar operations over an array.
- Widely **used for cumulative computations** and **inclusive/exclusive scans**.
- OpenMP 5.0 introduced **dedicated support** for this pattern.
- Three-Phase Approach:
  (a) **Build intermediate results** in parallel.
  (b) **Combine intermediate results** in pairs.
  (c) **Build final output** in parallel.


1 - Build intermediate results in parallel
2 - Sum intermediate results in pairs
3 - Build output results in parallel

Example 4: Scan Pattern on OpenMP

Example with a SIMD reduction clause:

```
#pragma omp simd reduction(inscan, +:scan_a)
for (int i = 0; i < N; i++) {
    simd_scan[i] = scan_a;
    #pragma omp scan exclusive(scan_a)
    scan_a += array[i];
}
```

11.2 Histogram Pattern

The **Histogram Pattern** is a fundamental and widely used computational method for **analyzing large datasets by aggregating values into predefined bins**. It is used in applications such as feature extraction, fraud detection, and speech recognition.

Characteristics

- **Definition**. **For each data element, a specific *bin counter* is identified and incremented**.
- **Applications**
  - **Feature extraction**: Identifying key characteristics in images or data.
  - **Fraud detection**: Analyzing transactional data for anomalies.
  - **Speech recognition**: Identifying patterns in audio signals.
- **Main Challenges**
  - **Output Interference**: **Avoiding concurrent writes** to the same bin counter in a parallel implementation.
  - **Memory Access Efficiency**: **Ensuring memory coalescence** for better bandwidth utilization.

Example 5: Histogram for Letter Count in Strings

Given an input string, the goal is to count the frequency of letters grouped into bins. For instance: the string “`programming massively parallel processors`” could be grouped into 4-letter bins like `{a-d, e-h, i-l, ...}`.
The output is:

```
a-d: 5
e-h: 5
i-l: 6
m-p: 10
q-t: 10
u-x: 1
y-z: 1
```

Sequential Implementation in C

```c
#define ALPHABET_LENGTH 26

sequential_Histogram(char *data, int length, int *histo) {
    for (int i = 0; i < length; ++i) {
        int alphabet_position = data[i] - 'a';
        if (alphabet_position >= 0 &&
            alphabet_position < ALPHABET_LENGTH) {
            histo[alphabet_position / 4]++;
        }
    }
}
```

Parallel Implementation

To parallelize, we must:

1. **Partition Input**: Divide the input dataset into sections.
2. **Thread Processing**: Assign each thread a section to process independently.

So we need to choose how to split the input and which thread to assign to each piece of data.

- **Simplest parallel version**. The simplest parallel implementation divides the input into sections. The **number of sections created is equal to the number of threads available**. For example, if the input are 28 words long and we have 4 threads, each section length will be $28 \div 4 = 7$ words per thread. In parallel, **each thread iterates through a section**.

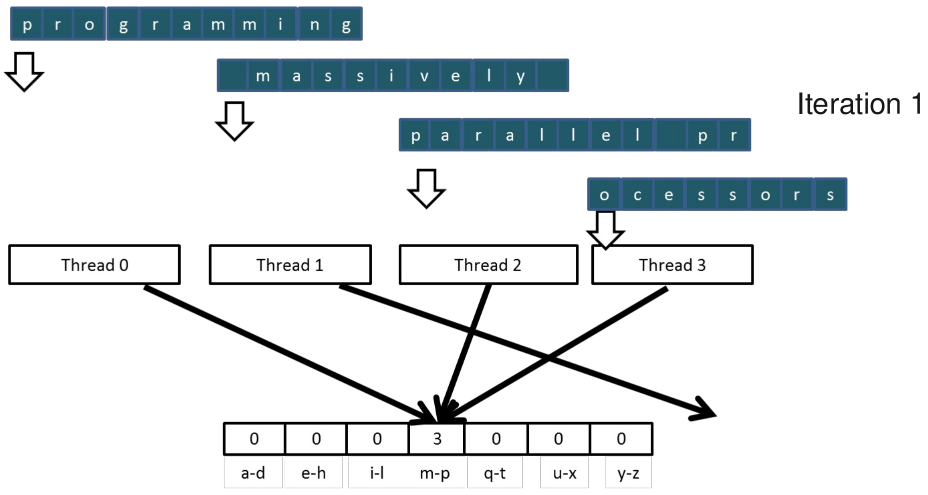


Figure 58: The simplest parallel implementation of the histogram pattern.

Simplest parallel version: Inefficient memory access

Each thread works on its assigned contiguous section of the input. And this might be good, but we don't consider that **memory accesses across threads are not contiguous**! So at each iteration, each thread requests memory locations not contiguous on memory.

- ✗ Memory Access Efficiency. **Adjacent threads work on non-adjacent memory sections**. While each thread accesses contiguous memory, the threads themselves are **not accessing contiguous memory as a group**.
- ✗ DRAM Bandwidth. Accesses from different threads are not coalesced, meaning **memory requests are spread out across the memory space**, leading to inefficient use of bandwidth. If the memory addresses are in the same location, the memory requests are grouped together.

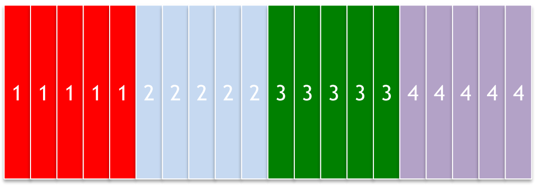


Figure 59: Memory allocation between threads on the simplest parallel version.

- **Parallel version with interleaved partitioning**. The interleaved partitioning, **threads work on interleaved indices**, so their **access are distributed across contiguous memory locations**.

Interleaved partitioning: the best memory option

In this case, **memory accesses from different threads are closer together in memory**, which aligns better with how memory is organized in hardware (cache lines and DRAM bursts). As a result:

- ✓ Memory Coalescing. **Threads access memory in a way that aligns with cache lines and DRAM bursts**.
- ✓ Bandwidth Utilization. **Memory requests are grouped together**, maximizing the DRAM bandwidth and improving overall performance.

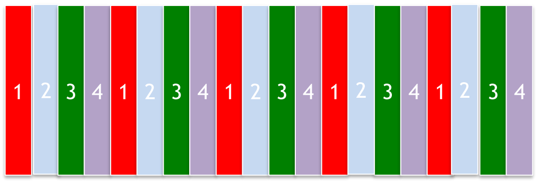


Figure 60: Memory allocation between threads on the interleaved partitioning version.

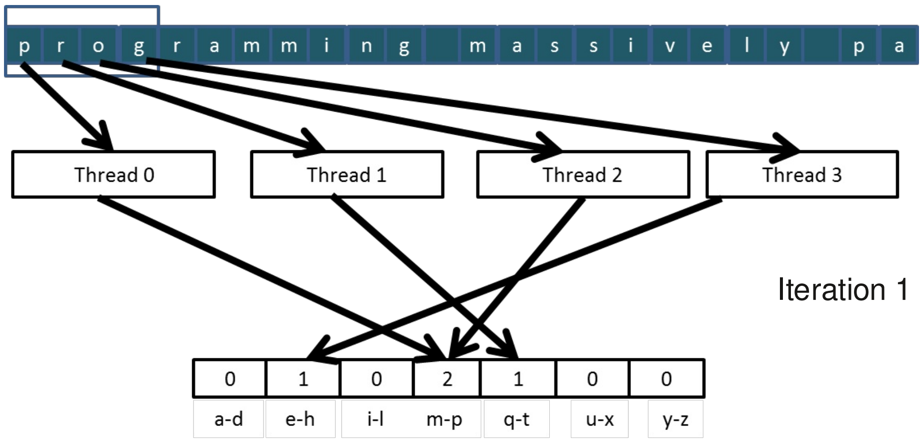

p r o g r a m m i n g m a s s i v e l y p a
Thread 0
Thread 1
Thread 2
Thread 3
Iteration 1
0 1 0 2 1 0 0
a-d e-h i-l m-p q-t u-x y-z

Figure 61: The parallel version with interleaved partitioning.

Why Interleaved Partitioning is Better

In the simplest version, the threads work independently on large contiguous chunks. While this avoids conflicts between threads, it leads to **scattered memory access** patterns at a hardware level. **DRAM modules are optimized** for coalesced accesses, which occur **when requests are close together**.

In interleaved partitioning, threads' accesses are distributed such that memory requests from different threads are **closer together in memory**. This aligns with how caches and DRAM are designed to handle access patterns efficiently, improving speed and reducing memory latency.

Implementations suffer from race condition

What happened? In this parallel implementation, **multiple threads update a shared histogram simultaneously**.

For example, in the figure 59 (page 368), we can see that threads #0, #2, and #3 are trying to increment the same bin at the same time. The correct result is 3, but this can **lead to data corruption because these operations are not atomic**. So we got lucky.

Why this happens. Incrementing a histogram bin is typically a three-step process:

1. Read the current value of the bin.
2. Increment the value.
3. Write the new value back to the bin.

In parallel, **these steps can interleave among threads, causing incorrect results**.

How to avoid race conditions

Exists two technique to use to avoid race conditions on the histogram pattern:

- **Atomic Operations**. We already discussed what atomic operations are and how they can solve (not the best way we can) the race conditions on page ??.

  But here we are talking about implementation. In CUDA, **atomic operations are implemented as hardware-supported functions** that perform a **read-modify-write operation as a single instruction** on a memory address. For the histogram pattern, atomic operations help resolve race conditions by ensuring only one thread modifies a memory location at any given time.

  Some of the most common APIs provided by CUDA include:

  - `atomicAdd`: **Adds a value** to a memory address atomically. Doc.
  - `atomicSub`: **Subtract** operation (atomically). Doc.
  - `atomicMin`: **Find minimum** operation (atomically). Doc.
  - `atomicMax`: **Find maximum** operation (atomically). Doc.
  - `atomicCAS` (**Compare-And-Swap**): Compares a value at an address and swaps it conditionally. Doc.

  #### Implementation of Atomic Operations in CUDA

  In this implementation, atomic operations ensure that multiple threads updating the same histogram bin don't cause data corruption.

```
__global__
void histo_kernel(
    unsigned char *buffer,
    long size,
    unsigned int *histo
) {
    // Unique thread ID
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    // Stride for processing chunks
    int stride = blockDim.x * gridDim.x;

    for (unsigned int i = tid; i < size; i += stride) {
        // Calculate bin index
        int alphabet_position = buffer[i] - 'a';
        if (alphabet_position >= 0 && alphabet_position < 26)
        {
            // Atomic update to avoid race conditions
            atomicAdd(&(histo[alphabet_position / 4]), 1);
        }
    }
}
```

- `tid`: Thread's unique ID, ensures threads process distinct elements (explained on page 0).
- `stride`: Ensures each thread processes non-overlapping data chunks.
- `atomicAdd`: Performs the addition operation atomically, preventing race conditions when multiple threads update the same histogram bin.

Performance Considerations & Recommendations

- **Memory Type**:
  - **Global memory (DRAM)**: High latency (over 1000 cycles per atomic operation).
  - **L2 Cache**: Approximately 1/10th the latency of DRAM.
  - **Shared Memory**: Lowest latency and is private to each thread block.
- **Throughput Impact**:
  * **Atomic operations significantly reduce throughput** when multiple threads access the same location because threads are serialized.
  * Access patterns and contention have a major impact on overall performance.

The **recommendations** are:

✓ **Use Shared Memory**: **Reduce global memory access latency** by leveraging shared memory for intermediate computation within thread blocks.

✓ **Reduce Contention**:
  * **Partition data** so threads update different bins.
  * **Minimize the number of atomic updates** to the same location (every atomic operation introduces potential overhead).

- **Privatization**. **Privatization** is a technique used to avoid race conditions in parallel computing by **providing private copies of shared resources to threads or blocks**.

  What it does

  Privatization **creates private copies of shared resources** (e.g., histograms) **for each thread or block**. Instead of multiple threads contending to access and modify a shared resource, each thread or block updates its own private copy independently. The **operation performed on the data must be associative and commutative** (e.g., addition for histograms).

  Benefits

  ✓ **Reduces Contention**. Threads do not compete to access shared memory resources during intermediate computations, significantly reducing contention.

  ✓ **Improves Throughput**. Since atomic operations on shared memory are faster than on global memory, privatization improves performance, especially when using shared memory for private copies.

  Challenges

  - **Overhead**. Allocating and initializing private copies adds computational overhead. Combining or reducing private copies into a final shared copy at the end introduces additional steps.
  - **Memory Fit**. The private data must fit into shared memory, which limits the size of privatized resources.

  How it works in combination with a histogram pattern

  1. **Private Copies Initialization**. Each block of threads initializes a local histogram in shared memory (e.g., `histo_s[]`).
  2. **Local Updates**. Threads within a block update their private histogram **using atomic operations** in shared memory. This step ensures no inter-block contention since the operations are confined to the shared memory of each block.
  3. **Reduction Step**. At the end of processing, private histograms are merged into a final shared histogram in global memory **using atomic operations**.

  Why Private Copies in Shared Memory?

  - Each block gets its own private copy of the histogram in **shared memory**, which is **fast and local to the block**.
  - **Threads** within the same block can **update their shared histogram without interfering with threads in other blocks** because each block **works on its own local copy**.

- Key Benefit: **no contention between blocks**, as there is no shared resource across blocks.

Why no inter-block contention?

- Since blocks do not access the same global memory histogram during local computations, there is **no need for atomic operations between blocks during this stage**.
- Threads within a block may still use **atomic operations**, but these are **limited to shared memory**, which is **much faster than global memory**.

The reduction phase is not so slow?

Only after all blocks have finished updating their private histograms do they write their results back to the global histogram. At this stage, atomic operations are required, but **the volume of these operations is much smaller**. In fact, instead of all threads updating the global histogram during each update, there is **only one write per bin per block**.

Example 6: Histogram Patterns Privatization Analogy

Image we have 10 groups of workers (*blocks*), each responsible for counting objects (*bins*) in their own room (*shared memory*).

- ✗ **Without privatization**: All workers from all groups try to update a single global counter in a central office (*global memory*). They must wait in line to update the counter.
- ✓ **With privatization**: Each group has its own private counter in their room. They work independently without competing. At the end, a manager (*reduction step*) collects and sums up the results from each room.

Implementation of Privatization in CUDA

```
__global__
void histogram_privatized_kernel(
    unsigned char* input,
    unsigned int* bins,
    unsigned int num_elements,
    unsigned int num_bins
) {
    // 1. Initialization
    unsigned int tid = blockIdx.x * blockDim.x + threadIdx.x;
    extern __shared__ unsigned int histo_s[];

    for(
        unsigned int binIdx = threadIdx.x;
        binIdx < num_bins;
        binIdx += blockDim.x
    ) {
        // Initialize private histogram
        histo_s[binIdx] = 0u;
    }
    __syncthreads();


    // 2. Local Histogram Updates
    for (
        unsigned int i = tid;
        i < num_elements;
        i += blockDim.x * gridDim.x
    ) {
        int alphabet_position = buffer[i] - 'a';
        if (alphabet_position >= 0 && alphabet_position < 26)
    {
            atomicAdd(&(histo_s[alphabet_position]), 1);
        }
    }
    __syncthreads();


    // 3. Reduction to Global Memory
    for (
        unsigned int binIdx = threadIdx.x;
        bindIdx < num_bins;
        binIdx += blockDim.x
    ) {
        atomicAdd(&(histo[binIdx]), histo_s[binIdx]);
    }
}
```