

Contents

1	PRAM	7
1.1	Prerequisites	7
1.2	Definition	7
1.3	How it works	8
1.3.1	Computation	8
1.3.2	PRAM Classification	8
1.3.3	Strengths of PRAM	9
1.3.4	How to compare PRAM models	9
1.4	MVM algorithm	11
1.5	SPMD sum	13
1.6	MM algorithm	17
1.7	PRAM variants and Lemmas	18
1.8	PRAM implementation	19
1.9	Amdahl's and Gustafson's Laws	21
2	Fundamentals of architecture	24
2.1	Introduction	24
2.1.1	Simplest processor	24
2.1.2	Superscalar processor	25
2.1.3	Single Instruction, Multiple Data (SIMD) processor	26
2.1.4	Multi-Core Processor	26
2.2	Accessing Memory	27
2.2.1	What is a memory?	27
2.2.2	How to reduce processor stalls	29
2.2.2.1	Cache	29
2.2.2.2	Multi-threading	29
3	Programming models	32
3.1	Implicit SPMD Program Compiler (ISPC)	32
3.2	Shared Address Space Model	36
3.3	Message Passing model of communication	37
3.4	Data-Parallel model	38
4	Parallel Programming Models and pthreads	40
4.1	How to create parallel algorithms and programs	40
4.2	Analyze parallel algorithms	42
4.3	Technologies	45
4.4	Threads	48
4.4.1	Flynn's taxonomy	48
4.4.2	Definition	48
4.4.3	pthreads API	50
4.4.3.1	Creation	50
4.4.3.2	Termination	51
4.4.3.3	Joining	52
4.4.3.4	Detaching	53
4.4.3.5	Joining through Barriers	54
4.4.3.6	Mutexes	55
4.4.3.7	Condition variables	55

5	OpenMP v5.2	56
5.1	Introduction	56
5.2	Basic syntax	58
5.3	Work sharing	61
5.3.1	For	61
5.3.1.1	Reduction	66
5.3.2	Sections	68
5.3.3	Single/Master	69
5.3.4	Tasks	70
5.3.4.1	Task dependences	73
5.4	Synchronization	77
5.5	Data environment	80
5.6	Memory model	88
5.7	Nested Parallelism	91
5.8	Cancellation	95
5.9	SIMD Vectorization	98
6	GPU Architecture	101
6.1	Introduction	101
6.2	GPU compute mode	102
6.3	CUDA	104
6.3.1	Basics of CUDA	104
6.3.2	Memory model	108
6.3.3	NVIDIA V100 Streaming Multiprocessor (SM)	110
6.3.4	Running a CUDA program on a GPU	113
6.3.5	Implementation of CUDA abstractions	119
6.3.6	Advanced thread scheduling	122
6.3.7	Memory and Data Locality in Depth	127
6.3.8	Tiling Technique	136
6.3.8.1	Tiled Matrix Multiplication	139
6.3.8.2	Implementation Tiled Matrix Multiplication	144
6.3.8.3	Any size matrix handling	149
6.3.9	Optimizing Memory Coalescing	154
7	CUDA	163
7.1	Introduction	163
7.2	CUDA Basics	167
7.2.1	GPGPU Best Practices	169
7.2.2	Compilation	171
7.2.3	Debugging	173
7.2.4	CUDA Kernel	176
7.3	Execution Model	179
7.4	Querying Device Properties	181
7.5	Thread hierarchy	183
7.6	Memory hierarchy	186
7.7	Streams	194
7.8	CUDA and OpenMP or MPI	198
7.8.1	Motivations	198
7.8.2	CUDA API for Multi-GPUs	203
7.8.3	Memory Management with Multiple GPUs	206

7.8.4	Batch Processing and Cooperative Patterns with OpenMP	212
7.8.5	OpenMP for heterogeneous architectures	214
7.8.6	MPI-CUDA applications	217
8	Memory Consistency	221
8.1	Coherence vs Consistency	221
8.2	Definition	224
8.3	Sequential Consistency Model	226
8.4	Memory Models with Relaxed Ordering	230
8.4.1	Allowing Reads to Move Ahead of Writes	231
8.4.2	Allowing writes to be reordered	233
8.4.3	Allowing all reorderings	235
8.5	Languages Need Memory Models Too	237
8.6	Implementing Locks	239
8.6.1	Introduction	239
8.6.2	Test-and-Set based lock	241
8.6.3	Test-and-Test-and-Set lock	245
9	Heterogeneous Processing	249
9.1	Energy Constrained Computing	251
9.2	Compute Specialization	252
9.3	Challenges of heterogeneous designs	266
9.4	Reducing energy consumption	269
10	Patterns	272
10.1	Dependencies	272
10.2	Parallel Patterns	281
10.2.1	Nesting Pattern	282
10.2.2	Serial Control Patterns	283
10.2.3	Parallel Control Patterns	285
10.2.4	Serial Data Management Patterns	290
10.2.5	Parallel Data Management Patterns	293
10.2.6	Other Parallel Patterns	296
10.3	Map Pattern	298
10.3.1	What is a Map?	298
10.3.2	Optimizations	300
10.3.2.1	Sequences of Maps	300
10.3.2.2	Code Fusion	301
10.3.2.3	Cache Fusion	302
10.3.3	Related Patterns	303
10.3.4	Scaled Vector Addition (SAXPY)	305
10.4	Collectives operations	308
10.4.1	Reduce (or Reduction) Pattern	309
10.4.2	Scan Pattern	315
10.5	Gather Pattern	326
10.5.1	What is a Gather?	326
10.5.2	Shift	330
10.5.3	Zip	332
10.5.4	Unzip	333
10.6	Scatter Pattern	334

10.6.1	What is a Scatter?	334
10.6.2	Avoid race conditions	337
10.6.2.1	Atomic Scatter	337
10.6.2.2	Permutation Scatter	339
10.6.2.3	Merge Scatter	341
10.6.2.4	Priority Scatter	343
10.7	Pack Pattern	344
10.7.1	What is a Pack?	344
10.7.2	Split	347
10.7.3	Unsplit	348
10.7.4	Bin	349
10.7.5	Expand	350
10.8	Partitioning Data	351
10.9	AoS vs. SoA	352
10.10	Stencil Pattern	357
10.10.1	What is a Stencil?	357
10.10.2	Implementing stencil with shift	358
10.10.3	Cache optimizations	360
10.10.4	Communication optimizations	362
11	Parallel Patterns in OpenMP and CUDA	364
11.1	OpenMP	364
11.2	Histogram Pattern	366
	Index	365

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

```
1 #pragma omp parallel for
2 for (int i = 0; i < N; i++) {
3     result[i] = operation(data[i]);
4 }
```

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

```
1 #pragma omp parallel for reduction(+:sum)
2 for (int i = 0; i < N; i++) {
3     sum += array[i];
4 }
```

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

```

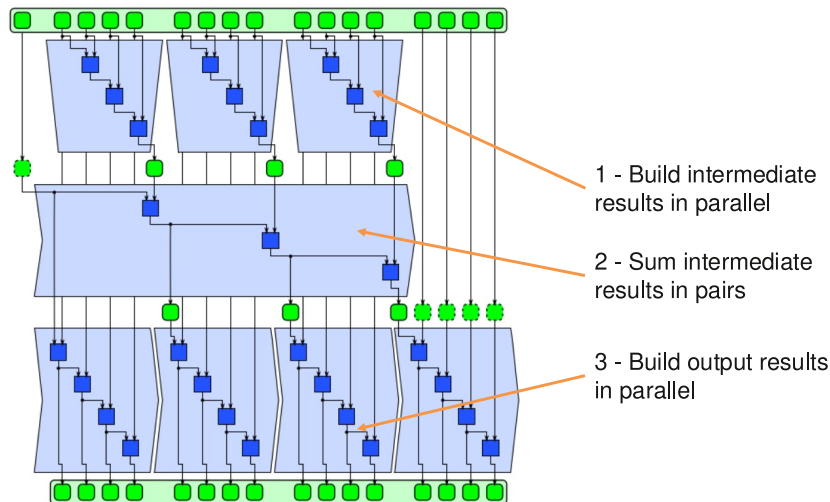
1 #pragma omp task
2 void process_task() {
3     // Perform a unit of work
4 }

```

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.



Example 4: Scan Pattern on OpenMP

Example with a SIMD reduction clause:

```

1 #pragma omp simd reduction(inscan, +:scan_a)
2 for (int i = 0; i < N; i++) {
3     simd_scan[i] = scan_a;
4     #pragma omp scan exclusive(scan_a)
5     scan_a += array[i];
6 }

```

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:

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

✂ Sequential Implementation in C

```

1 #define ALPHABET_LENGTH 26
2
3 sequential_Histogram(char *data, int length, int *histo) {
4     for (int i = 0; i < length; ++i) {
5         int alphabet_position = data[i] - 'a';
6         if (alphabet_position >= 0 &&
7             alphabet_position < ALPHABET_LENGTH) {
8             histo[alphabet_position / 4]++;
9         }
10    }
11 }

```

🧑‍💻 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**.

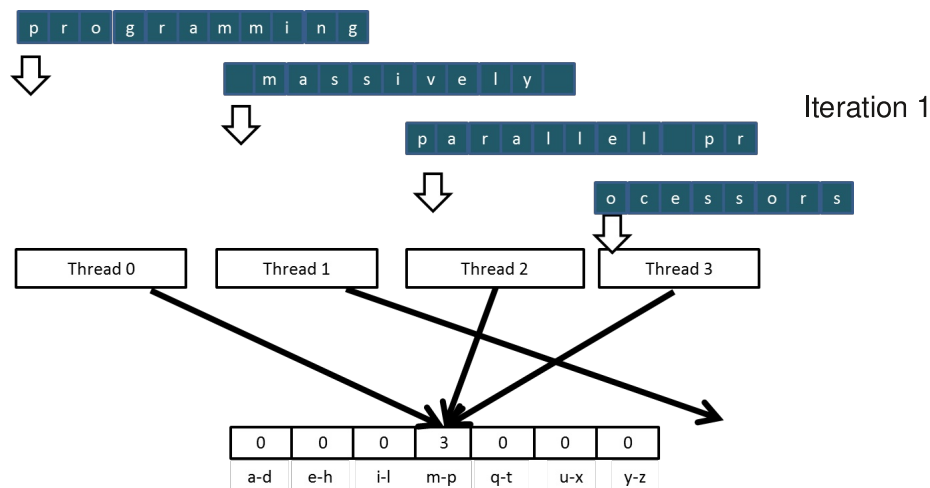


Figure 58: The simplest parallel implementation of the histogram pattern.

⚡ 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.

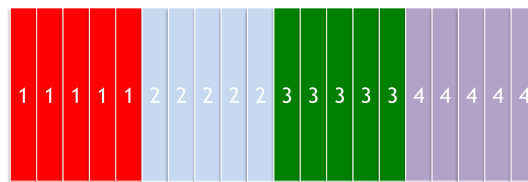


Figure 59: Memory allocation between threads on the simplest parallel version.

- **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.

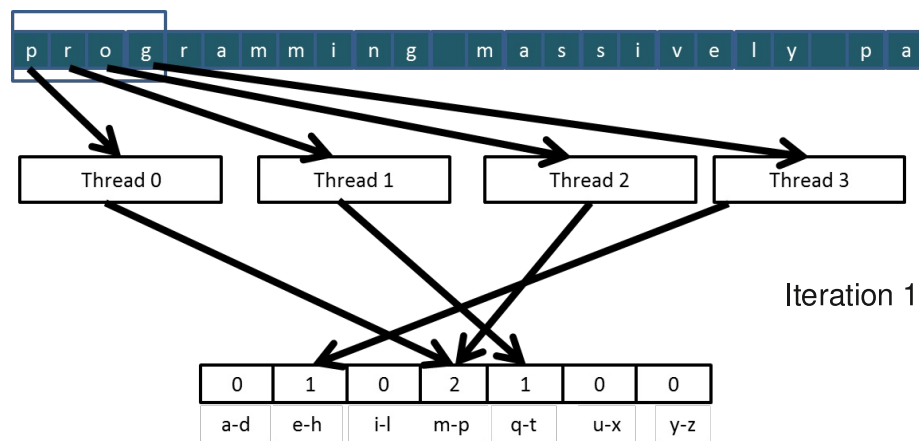


Figure 61: The parallel version with interleaved partitioning.

🧠 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.

```

1 __global__
2 void histo_kernel(
3     unsigned char *buffer,
4     long size,
5     unsigned int *histo
6 ) {
7     // Unique thread ID
8     int tid = threadIdx.x + blockIdx.x * blockDim.x;
9     // Stride for processing chunks
10    int stride = blockDim.x * gridDim.x;
11
12    for (unsigned int i = tid; i < size; i += stride) {
13        // Calculate bin index
14        int alphabet_position = buffer[i] - 'a';
15        if (alphabet_position >= 0 && alphabet_position < 26)
16        {
17            // Atomic update to avoid race conditions
18            atomicAdd(&(histo[alphabet_position / 4]), 1);
19        }
20    }
21 }

```

- **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

Imagine 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

```

1  __global__
2  void histogram_privatized_kernel(
3      unsigned char* input,
4      unsigned int* bins,
5      unsigned int num_elements,
6      unsigned int num_bins
7  ) {
8      // 1. Initialization
9      unsigned int tid = blockIdx.x * blockDim.x + threadIdx.x;
10     extern __shared__ unsigned int histo_s[];
11
12     for(
13         unsigned int binIdx = threadIdx.x;
14         binIdx < num_bins;
15         binIdx += blockDim.x
16     ) {
17         // Initialize private histogram
18         histo_s[binIdx] = 0u;
19     }
20     __syncthreads();
21
22
23     // 2. Local Histogram Updates
24     for (
25         unsigned int i = tid;
26         i < num_elements;
27         i += blockDim.x * gridDim.x
28     ) {
29         int alphabet_position = buffer[i] - 'a';
30         if (alphabet_position >= 0 && alphabet_position < 26)
31         {
32             atomicAdd(&(histo_s[alphabet_position]), 1);
33         }
34     }
35     __syncthreads();
36
37     // 3. Reduction to Global Memory
38     for (
39         unsigned int binIdx = threadIdx.x;
40         binIdx < num_bins;
41         binIdx += blockDim.x
42     ) {
43         atomicAdd(&(histo[binIdx]), histo_s[binIdx]);
44     }
45 }

```
