

# Contents

<b>1</b>	<b>PRAM</b>	<b>6</b>
1.1	Prerequisites . . . . .	6
1.2	Definition . . . . .	6
1.3	How it works . . . . .	7
1.3.1	Computation . . . . .	7
1.3.2	PRAM Classification . . . . .	7
1.3.3	Strengths of PRAM . . . . .	8
1.3.4	How to compare PRAM models . . . . .	8
1.4	MVM algorithm . . . . .	10
1.5	SPMD sum . . . . .	12
1.6	MM algorithm . . . . .	16
1.7	PRAM variants and Lemmas . . . . .	17
1.8	PRAM implementation . . . . .	18
1.9	Amdahl's and Gustafson's Laws . . . . .	20
<b>2</b>	<b>Fundamentals of architecture</b>	<b>23</b>
2.1	Introduction . . . . .	23
2.1.1	Simplest processor . . . . .	23
2.1.2	Superscalar processor . . . . .	24
2.1.3	Single Instruction, Multiple Data (SIMD) processor . . . . .	25
2.1.4	Multi-Core Processor . . . . .	25
2.2	Accessing Memory . . . . .	26
2.2.1	What is a memory? . . . . .	26
2.2.2	How to reduce processor stalls . . . . .	28
2.2.2.1	Cache . . . . .	28
2.2.2.2	Multi-threading . . . . .	28
<b>3</b>	<b>Programming models</b>	<b>31</b>
3.1	Implicit SPMD Program Compiler (ISPC) . . . . .	31
3.2	Shared Address Space Model . . . . .	35
3.3	Message Passing model of communication . . . . .	36
3.4	Data-Parallel model . . . . .	37
<b>4</b>	<b>Parallel Programming Models and pthreads</b>	<b>39</b>
4.1	How to create parallel algorithms and programs . . . . .	39
4.2	Analyze parallel algorithms . . . . .	41
4.3	Technologies . . . . .	44
4.4	Threads . . . . .	47
4.4.1	Flynn's taxonomy . . . . .	47
4.4.2	Definition . . . . .	47
4.4.3	pthreads API . . . . .	49
4.4.3.1	Creation . . . . .	49
4.4.3.2	Termination . . . . .	50
4.4.3.3	Joining . . . . .	51
4.4.3.4	Detaching . . . . .	52
4.4.3.5	Joining through Barriers . . . . .	53
4.4.3.6	Mutexes . . . . .	54
4.4.3.7	Condition variables . . . . .	54

<b>5</b>	<b>OpenMP v5.2</b>	<b>55</b>
5.1	Introduction . . . . .	55
5.2	Basic syntax . . . . .	57
5.3	Work sharing . . . . .	60
5.3.1	For . . . . .	60
5.3.1.1	Reduction . . . . .	65
5.3.2	Sections . . . . .	67
5.3.3	Single/Master . . . . .	68
5.3.4	Tasks . . . . .	69
5.3.4.1	Task dependences . . . . .	72
5.4	Synchronization . . . . .	76
5.5	Data environment . . . . .	79
5.6	Memory model . . . . .	87
5.7	Nested Parallelism . . . . .	90
5.8	Cancellation . . . . .	94
5.9	SIMD Vectorization . . . . .	97
<b>6</b>	<b>GPU Architecture</b>	<b>100</b>
6.1	Introduction . . . . .	100
6.2	GPU compute mode . . . . .	101
6.3	CUDA . . . . .	103
6.3.1	Basics of CUDA . . . . .	103
6.3.2	Memory model . . . . .	107
6.3.3	NVIDIA V100 Streaming Multiprocessor (SM) . . . . .	109
6.3.4	Running a CUDA program on a GPU . . . . .	112
6.3.5	Implementation of CUDA abstractions . . . . .	118
6.3.6	Advanced thread scheduling . . . . .	121
6.3.7	Memory and Data Locality in Depth . . . . .	126
6.3.8	Tiling Technique . . . . .	135
6.3.8.1	Tiled Matrix Multiplication . . . . .	138
6.3.8.2	Implementation Tiled Matrix Multiplication . . . . .	143
6.3.8.3	Any size matrix handling . . . . .	148
6.3.9	Optimizing Memory Coalescing . . . . .	153
<b>7</b>	<b>CUDA</b>	<b>162</b>
7.1	Introduction . . . . .	162
7.2	CUDA Basics . . . . .	166
7.2.1	GPGPU Best Practices . . . . .	168
7.2.2	Compilation . . . . .	170
7.2.3	Debugging . . . . .	172
7.2.4	CUDA Kernel . . . . .	175
7.3	Execution Model . . . . .	178
7.4	Querying Device Properties . . . . .	180
7.5	Thread hierarchy . . . . .	182
7.6	Memory hierarchy . . . . .	185
7.7	Streams . . . . .	193
7.8	CUDA and OpenMP or MPI . . . . .	197
7.8.1	Motivations . . . . .	197
7.8.2	CUDA API for Multi-GPUs . . . . .	202
7.8.3	Memory Management with Multiple GPUs . . . . .	205

7.8.4	Batch Processing and Cooperative Patterns with OpenMP	211
7.8.5	OpenMP for heterogeneous architectures . . . . .	213
7.8.6	MPI-CUDA applications . . . . .	216
<b>8</b>	<b>Memory Consistency</b>	<b>220</b>
8.1	Coherence vs Consistency . . . . .	220
8.2	Definition . . . . .	223
8.3	Sequential Consistency Model . . . . .	225
8.4	Memory Models with Relaxed Ordering . . . . .	229
8.4.1	Allowing Reads to Move Ahead of Writes . . . . .	230
8.4.2	Allowing writes to be reordered . . . . .	232
8.4.3	Allowing all reorderings . . . . .	234
8.5	Languages Need Memory Models Too . . . . .	236
8.6	Implementing Locks . . . . .	238
8.6.1	Introduction . . . . .	238
8.6.2	Test-and-Set based lock . . . . .	240
8.6.3	Test-and-Test-and-Set lock . . . . .	244
<b>9</b>	<b>Heterogeneous Processing</b>	<b>248</b>
9.1	Energy Constrained Computing . . . . .	250
9.2	Compute Specialization . . . . .	251
9.3	Challenges of heterogeneous designs . . . . .	265
9.4	Reducing energy consumption . . . . .	268
<b>10</b>	<b>Patterns</b>	<b>271</b>
10.1	Dependencies . . . . .	271
10.2	Parallel Patterns . . . . .	280
10.2.1	Nesting Pattern . . . . .	281
10.2.2	Serial Control Patterns . . . . .	282
10.2.3	Parallel Control Patterns . . . . .	284
10.2.4	Serial Data Management Patterns . . . . .	289
10.2.5	Parallel Data Management Patterns . . . . .	292
10.2.6	Other Parallel Patterns . . . . .	295
10.3	Map Pattern . . . . .	297
10.3.1	What is a Map? . . . . .	297
10.3.2	Optimizations . . . . .	299
10.3.2.1	Sequences of Maps . . . . .	299
10.3.2.2	Code Fusion . . . . .	300
10.3.2.3	Cache Fusion . . . . .	301
10.3.3	Related Patterns . . . . .	302
10.3.4	Scaled Vector Addition (SAXPY) . . . . .	304
10.4	Collectives operations . . . . .	307
10.4.1	Reduce (or Reduction) Pattern . . . . .	308
10.4.2	Scan Pattern . . . . .	314
10.5	Gather Pattern . . . . .	325
10.5.1	What is a Gather? . . . . .	325
10.5.2	Shift . . . . .	329
10.5.3	Zip . . . . .	331
10.5.4	Unzip . . . . .	332
10.6	Scatter Pattern . . . . .	333

10.6.1	What is a Scatter? . . . . .	333
10.6.2	Avoid race conditions . . . . .	335
10.6.2.1	Atomic Scatter . . . . .	335
<b>Index</b>		<b>330</b>

---

## 10.6.2 Avoid race conditions

### 10.6.2.1 Atomic Scatter

**Atomic Scatter** is a **collision resolution strategy** used when multiple threads attempt to write to the same memory location simultaneously. It is based on one of the most common and famous topic: **atomic operations**.

#### ? How does it work?

- ✓ **Atomic Writes.** Each write is **atomic**, meaning it either **completes fully or doesn't happen at all**. No partial writes.
- ✗ **Non-Deterministic Outcome.** Since it relies solely on atomic writes, this doesn't mean that there is a mechanism to check which thread should write before another. **When a collision occurs, the outcome depends on which write completes first**. There's no predefined rule for determining which value to store.

#### Example 49: Atomic Scatter

Consider this:

- Source Data:

`A = [A, B, C, D, E, F]`

- Index Array (write locations):

`idx = [1, 5, 0, 2, 2, 4]`

Notice that each thread is assigned to every position of the `idx` array. This means that threads #4 and #5 must write to the same position on the output array. However, we adopt the atomic strategies so that what happens depends on the order of writing.

A possible scenario:

1. Thread #4 asks to do an atomic write to position 2 of the output array. The operation is marked atomic and can be done safely. The value written to `A[2]` is D.
2. Then thread #5 performs the same steps and overwrites the value written by thread #4 and writes the value E to `A[2]`.

This is non-deterministic, so this is a possible scenario, but it could also be the other way around and the final value could be D.

---

### ✓ Advantages

- ✓ **Simple to implement.** No need for complex locking mechanisms.
- ✓ **Fast.** Atomic **operations are optimized in hardware**, making them efficient for parallel execution.

### ✗ Disadvantages

- ✗ **Unpredictable results.** Non-determinism can be **problematic for algorithms that require consistent outputs**.
- ✗ **Data loss.** **One of the colliding values will be lost** without any mechanism to combine or preserve both.

### ? When to use Atomic Scatter?

Atomic scatter is suitable when:

- **Collisions are rare:** so the occasional data loss or non-determinism is acceptable.
- **We don't care which value is kept:** for example, in random sampling or approximate algorithms.