

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
11.3	Reduction Pattern	375
11.4	Scan Pattern	379
12	Heterogeneous Computing - DSLs and HLS	383
12.1	Introduction to Heterogeneous Computing	383
12.2	Heterogeneous parallel programming	384
12.3	DSLs and Halide	385
	Index	384

12 Heterogeneous Computing - DSLs and HLS

12.1 Introduction to Heterogeneous Computing

Heterogeneous Computing (or **Heterogeneous Processing**) refers to systems that use multiple types of processors or accelerators to handle different workloads more efficiently.

- In contrast to traditional homogeneous systems (which only use CPUs), heterogeneous systems combine different processing units such as CPUs, GPUs, DSPs, and FPGAs.
- The goal is to match the right processor to the right task, achieving higher performance and energy efficiency.

Example 1: Heterogeneous Processing

A self-driving car requires CPUs for decision-making, GPUs for image recognition, and FPGAs for real-time sensor fusion.

🔌 Energy-Efficient Computing Strategies

When designing a heterogeneous system, performance isn't the only goal; **energy efficiency is just as critical**. Given a fixed power budget, simply **increasing performance without considering power constraints is inefficient**. Specialized hardware (e.g., FPGAs, ASICs) achieves better performance per watt than general-purpose processors.

There are two main strategies for improving energy efficiency:

1. **Use Specialized Processors.** CPUs are not energy-efficient due to instruction decoding, branch handling, and pipeline management overhead. Specialized hardware (FPGAs, ASICs) reduces overhead, leading to more computations per joule.

$$\text{Power} = \frac{\text{Op}}{\text{second}} \times \frac{\text{Joules}}{\text{Op}}$$

2. **Minimize Data Movement.** Memory access consumes more energy than computation! Optimizing data locality reduces power consumption. For example, moving computation closer to memory (e.g., using tensor core inside GPUs) significantly reduces energy cost.

12.2 Heterogeneous parallel programming

⚠ Challenges of Writing Portable and Efficient Parallel Code

Writing parallel programs for heterogeneous systems is difficult due to the following reasons:

1. **Diverse Hardware Architectures.** A CPU, GPU, and FPGA all have different programming models. **Code written for one hardware type may not perform well on another.**
2. **Performance vs. Productivity Trade-offs.**
 - **Performance:** Low-level programming (e.g., **CUDA, OpenCL, Verilog**) allows fine-tuned optimizations but **is hard to program.**
 - **Productivity:** High-level abstractions (e.g., **OpenMP, DSLs**) improve productivity but **may introduce performance overhead.**
3. **Memory Management.** Different memory models (shared vs. distributed) require different optimizations. Data movement between CPU and GPU memory can be costly if not handled efficiently.
4. **Scalability Issues.** Some **programs scale well on GPUs but poorly on CPUs** due to synchronization and memory bandwidth limitations.

✔ The Ideal Parallel Programming Language

An ideal parallel programming model should provide a balance of:

- ✔ **Performance.** Optimized execution across different hardware.
- ✔ **Productivity.** Easy to use and develop.
- ✔ **Generality.** Works across different architectures.

However, **most existing languages optimize only one or two** of these factors, leading to trade-offs.

Approach	Performance	Productivity	Generality
CUDA/OpenCL	✔ High	✗ Low	✗ Low
OpenMP (CPU)	✔ High	✔ Medium	✗ Low
MPI (Distributed)	✔ High	✗ Low	✔ High
FPGA/Verilog/VHDL	✔ Very High	✗ Very Low	✗ Low
High-Level Synthesis	✔ High	✔ Medium	✗ Low

❓ Why is this important?

If we want **portable parallel programs**, we need **new high-level abstractions** like Domain-Specific Languages (DSLs), which will be covered in the next section.

12.3 DSLs and Halide

🔗 What are Domain-Specific Languages (DSLs)?

A **Domain-Specific Language (DSL)** is a **specialized programming language** designed for a **specific application domain**. The main characteristics of DSLs are:

- **Restricted expressiveness** (focused on a single domain)
- **High-level, declarative syntax** (easier than general purpose languages)
- **Optimized performance** for the target domain
- **May be standalone or embedded** in another language

DSL Name	Target Domain	Key Benefits
Halide	Image Processing	Separates algorithm from scheduling for optimized execution.
TensorFlow	Machine Learning	Optimized computation graphs for AI workloads.
SQL	Databases	Declarative queries for efficient data retrieval.
Verilog/VHDL	Hardware Design	Describes digital circuits for synthesis.

Table 11: Examples of DSLs.

⚖️ Embedded vs. External DSLs

DSLs can be classified as:

- **External** DSLs:
 - 📖 Have **their own** *syntax* and *compiler/interpreter*.
 - 🔗 **Example:** SQL, Halide, Verilog.
 - ✅ **Advantages:** can be **more optimized** but require custom compilers.
- **Embedded** DSLs:
 - 📖 **Built inside another general-purpose language.**
 - 🔗 **Example:** TensorFlow (embedded in Python).
 - ✅ **Advantages:** benefit from **integration** with the host language

⚡ DSL Use Case: Halide for Image Processing

Halide is a Domain-Specific Language (DSL) for high-performance image processing.

❓ Why does image processing need DSLs?

- 🔗 Image processing is **data-intensive** and **requires high performance**.
- 🗨️ **Traditional solutions** (C++, CUDA, OpenCV) **require manual optimizations**.
- 😞 Optimizing code for **parallelism and memory efficiency** is difficult.

❓ Why Halide?

- ✂ Separates “*what*” is computed from “*how*” it is executed.
- 🔗 Expresses **computations at a high level**, leaving optimizations to the compiler.
- 🌐 **Portable** across CPUs, GPUs, and FPGAs.

✂ How Halide Works: Separating Algorithm from Schedule

In Halide, a **key feature is the separation of what** a program computes (*computation/algorithm*) from **how** it executes (*schedule*). This means:

- The **algorithm** specifies **what operations should be performed**. In other words, specifies **what to compute** (like a mathematical formula).
- The **schedule** defines **how those operations should be executed efficiently** on the hardware. In other words, specifies **how to execute the computation** (parallelism, memory layout, vectorization).

Now we see the difference between the traditional approach we have always used and the halide approach:

- **Traditional Approach (C++, CUDA, OpenCV)**. In traditional programming languages (e.g., C++, OpenCV, CUDA), the **algorithm and execution strategy are mixed together**. This means that if we want to **change parallelization or memory access optimizations**, we must **rewrite parts of the algorithm itself**. This makes it *hard to experiment* with different optimizations.

⚠ Problems with the Traditional Approach

1. **If we want to optimize** for vectorization, parallel execution, or memory layout, we **must modify the algorithm itself**.
2. The **same code cannot easily be reused** for different architectures (e.g., CPU, GPU, FPGA).

Example 2: Problems with the Traditional Approach

```

1 void box_blur(const Image &in, Image &out) {
2     for (int y = 1; y < in.height() - 1; y++) {
3         for (int x = 1; x < in.width() - 1; x++) {
4             out(x, y) = (
5                 in(x-1, y) + in(x, y) + in(x+1, y)
6             ) / 3;
7         }
8     }
9 }

```

The problem here is that we need to specify the “*what*”, i.e. what operations should be performed, but also “*how*” these operations should be performed.

- **Halide’s Approach: Separate Algorithm from Execution.** Halide splits the computation into two parts:

1. **Computation** (Algorithm) - *What to Compute*
 - Defines the mathematical computation.
 - **Remains unchanged across different hardware targets.**
2. **Schedule** - *How to Execute Efficiently*
 - Controls memory layout, parallelization, and optimization.
 - **Can be changed without modifying the algorithm.**

Example 3: Box Blur in Halide

The computation part is separated from the scheduling! So a change in the algorithm can be made without affecting the control of the execution. Therefore, we define simply:

1. Computation (Algorithm), stays the same:

```

1 Var x, y;
2 Func blurx, blurry;
3
4 // First pass: horizontal blur
5 blurx(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))
6               / 3;
7
8 // Second pass: vertical blur
9 blurry(x, y) = (blurx(x, y-1) + blurx(x, y) + blurx(
10                x, y+1)) / 3;

```

This part **only describes the math**, not how it should run.

2. Schedule, controls execution (can be changed easily):

```

1 blurry.tile(x, y, xi, yi, 256, 32)
2     // Vectorized execution for SIMD
3     .vectorize(xi, 8)
4     // Parallel execution over y-dimension
5     .parallel(y);

```

```
6
7 blurx.compute_at(blury, x) // Compute blurx only
   when needed by blury
8   .vectorize(x, 8);
```

This part **controls execution strategy** but does **not modify the algorithm**. The same algorithm can now **run efficiently on different hardware architectures** just by changing the schedule.

✔ Why DSLs Matter for Performance and Productivity: Advantages

- ✔ **Performance Optimization**. A Halide program can be better than **hand-optimization C++ code**. Scheduling decisions affect **parallel execution, memory locality, and vectorization**.
- ✔ **Productivity**. Instead of manually optimizing, **Halide allows rapid exploration of different schedules**. Easier to **port to different architectures** (CPU, GPU, FPGA).

In conclusion, DSLs like Halide **automate low-level optimizations**, enabling *faster* and *more efficient* code for specialized domains.