

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
12.4	Scheduling & Performance Optimization in Halide	389
12.5	Introduction to HLS	392
12.6	HLS Workflow	394
	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.

12.4 Scheduling & Performance Optimization in Halide

This section focuses on **how different scheduling strategies in Halide affect performance** and how to **choose the best schedule for different hardware architectures**.

🔍 Why Scheduling Matters

Scheduling is the **key to optimizing performance in Halide**.

- A poorly scheduled program can be 10× slower than an optimized one.
- Memory access, cache locality, parallelism, and vectorization all **depend on scheduling**.

As we have already seen, in traditional languages (C++, CUDA, OpenMP), scheduling decisions must be hard-coded into the algorithm. In Halide, the schedule is separate and can be changed without changing the algorithm.

🔍 How Scheduling Affects Performance

Different **scheduling strategies** impact how the computation is executed on hardware:

Scheduling Strategy	Impact
Serial Execution	Simple, but slow. No parallelism.
Parallel Execution	Uses multiple CPU cores. Good for multi-core CPUs.
Vectorization (SIMD)	Uses wide registers for efficiency (e.g., AVX).
Tiling	Improves cache locality by processing data in chunks.
Compute-at	Controls when intermediate results are computed.
Store-at	Controls where intermediate results are stored.


☰ Scheduling Strategies in Halide

Let's take the Box Blur Algorithm as an example. The **Box Blur** is a simple **image processing technique used for smoothing or blurring an image**. It performs two main steps:

1. **Each pixel** in the output image is computed as the **average of its neighboring pixels**.
2. It applies a **moving average filter** over a **small window** (e.g., 3×3 or 5×5 pixels).

It is used to reduce noise in images, to create a smooth, blurry effect and, above all, because it is fast and efficient, since it involves only two operations: adding and dividing.

However, we will now analyze the algorithm implemented in Halide using different strategies:




 **Default Serial Execution (Slow).** Without scheduling, Halide will execute in **serial order** (one pixel at a time).


```

1 Var x, y;
2 Func blurx, blurry;
3
4 blurx(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y)) / 3;
5 blurry(x, y) = (blurx(x, y-1) + blurx(x, y) + blurx(x, y+1)) /
    3;

```

Problems

-  No parallelism or vectorization.
-  Poor memory access patterns.
-  Slow execution.



 **Parallel Execution.** We can **add parallelism** to use multiple CPU cores:


```

1 blurry.parallel(y);

```

Advantages

-  Halide automatically splits the work across CPU cores.
-  Useful for multi-threaded execution on CPUs.



 **Vectorization (SIMD).** Modern CPUs support **SIMD instructions** (e.g., AVX, NEON) to process multiple pixels at once:


```

1 blurry.vectorize(x, 8);

```

Advantages

-  Uses **SIMD registers** for faster execution.
-  Works best for **data-parallel workloads** like image processing.


 **Tiling for Better Cache Performance.** Instead of processing the whole image at once, we **divide it into smaller tiles**:

```

1 blurry.tile(x, y, xi, yi, 256, 32);

```

Advantages

-  Each tile fits better in cache, **reducing memory latency**.

✓ Improves locality of reference (less cache thrashing).

🔧 **Optimized Schedule: Combining Techniques.** We can combine multiple scheduling strategies for maximum performance:

```

1 // Process in 256x32 tiles
2 blurry.tile(x, y, xi, yi, 256, 32)
3     // Vectorized execution
4     .vectorize(xi, 8)
5     // Parallel execution across CPU cores
6     .parallel(y);
7
8 // Compute blurx only when needed
9 blurx.compute_at(blurry, x)
10    .vectorize(x, 8);

```

✓ Advantages

- ✓ Breaks image into tiles for cache efficiency.
- ✓ Uses SIMD vectorization for fast execution.
- ✓ Runs in parallel on multiple CPU cores.
- ✓ Intermediate results (blurx) are computed only when needed.

⚖️ Trade-offs in Scheduling

But *how do we choose the right scheduling?* Well, we need to find a good trade-off. Different scheduling choices affect **performance trade-offs**:

Scheduling Strategy	Performance Impact
Parallel Execution	Increases throughput, uses multiple cores.
Vectorization (SIMD)	Improves performance on CPUs/GPUs.
Tiling	Improves cache locality, reduces memory overhead.
Compute-at	Avoids redundant computations.
Store-at	Reduces memory footprint but increases re-computation.

Table 12: Performance trade-offs in Scheduling.

✓ Conclusion

In conclusion, the key takeaways are:

- **Scheduling** is the key to performance in Halide.
- **Parallel execution, vectorization, and tiling** significantly improve performance.
- **Halide's flexibility** allows quick experimentation with different schedules.
- The **right schedule depends on hardware constraints** (CPU, GPU, FPGA).

12.5 Introduction to HLS

🧐 What is High-Level Synthesis (HLS)?

High-Level Synthesis (HLS) is a process that converts high-level software code (C/C++/Python) into hardware designs (Verilog/VHDL). It automates hardware generation, allowing developers to describe behavior in software-like code while the tool generates optimized circuits.

📋 Traditional FPGA/ASIC Design Flow (Without HLS)

The traditional FPGA or ASIC design flow is divided into three main steps:

1. **Write Register-Transfer Level (RTL) code** in Verilog/VHDL.
2. Manually optimize for **timing, area, power**.
3. Run **logic synthesis, place & route**, and **fabrication**.

The main **problems** with this approach are:

- ✗ **Time-consuming** (designing hardware manually takes months).
- ✗ **Error-prone** (low-level bugs are hard to debug).
- ✗ **Difficult to modify** (small changes require rewriting RTL code).

Aspect	Traditional RTL	High-Level Synthesis
Design Level	Low-level: gates, registers	High-level (C++, Python)
Productivity	✗ Time-consuming	✓ Faster development
Optimizations	Manual pipeline and control logic	Automated scheduling
Reusability	✗ Difficult to modify	✓ Easily reusable code
Learning Curve	Steep (hardware expertise needed)	Easier (similar to software programming)

Table 13: 🔄 Differences Between HLS and RTL-Based Hardware Design.

We can conclude that **HLS allows software engineers to efficiently design hardware without deep knowledge of Verilog/VHDL**.

✔ Why use HLS instead of traditional RTL design? Benefits of HLS

- ✔ **Faster Development Cycle.** Designers can write C++/Python instead of Verilog, reducing design time.
- ✔ **Automated Optimizations.** HLS compilers automatically optimize **parallelism**, **pipelining**, and **resource allocation**.
- ✔ **Easier HW/SW Co-Design.** Enables rapid prototyping of hardware accelerators.
- ✔ **Technology Independence.** The same C++ code can be compiled for different FPGA/ASIC platforms.

✖ Challenges of HLS

- Not all software code can be efficiently translated into hardware.
- HLS tools must explore a huge design space (parallelism, pipelining, resource constraints).
- Requires careful optimization of memory access and data flow.

✔ Conclusion

In conclusion, the key takeaways are:

- HLS translates **high-level software** (C/C++/Python) into hardware designs (Verilog/VHDL).
- It automates the hardware design process, **reducing time and complexity**.
- HLS enables software engineers to **design hardware accelerators without deep RTL expertise**.
- Challenges include **optimizing memory access, parallelism, and scheduling**.

12.6 HLS Workflow

This section explains *how* **HLS converts high-level code** (C/C++/Python) **into hardware** (Verilog/VHDL) and the key steps in the HLS design flow.

Inputs to an HLS Compiler

To generate hardware from high-level code, an **HLS tool requires three main inputs**:

1. **High-Level Code** (C, C++, or Python)
 - Describes the **algorithm's behavior**.
 - Written similarly to software but **optimized for hardware**.
2. **Library of Characterized Modules**
 - **Predefined hardware** building blocks (e.g., adders, multipliers, memory units).
 - Helps the compiler understand available resources.
3. **Constraints & Optimization Directives**
 - Designer-defined constraints such as:
 - **Area constraints** (how much hardware can be used).
 - **Timing constraints** (desired clock speed).
 - **Memory hierarchy** (external vs. on-chip memory).
 - Optional **HLS pragmas** (e.g., loop unrolling, pipelining) to fine-tune performance.

HLS Objectives

The goal of **HLS is to generate an efficient hardware design** based on the following objectives:

- ✓ **Minimize Area**: Uses fewer functional units, registers, and interconnects.
- ✓ **Maximize Speed**: Reduces the number of clock cycles (latency) and increases throughput.
- ✓ **Optimize Power**: Reduces energy consumption for embedded systems.

However, the main trade-offs must be:

- **Optimizing for speed** \Rightarrow may increase hardware area.
- **Reducing hardware area** \Rightarrow may increase execution time.

HLS Compilation Flow

HLS follows **three main steps**:

- **Front-End (Parsing and IR Generation)**. Converts C++/Python code into an **Intermediate Representation (IR)**. IR is **similar to software compiler representations** (e.g., LLVM IR). In other words, it breaks the program down into basic operations.
- **Middle-End (Optimization and Scheduling)**. At this point, the HLS tools performed three important operations:
 1. It **analyzes data dependencies** and **determines execution order**.
 2. It performs **scheduling** (decides when each operation runs).
 3. It allocates **hardware resources** (multipliers, memory, registers).

Example 4: Scheduling Choices

Scheduling Strategy	Impact
As Soon As Possible	Minimizes latency, but may use more hardware.
As Late As Possible	Reduces hardware usage, but may increase delay.
Loop Unrolling	Increases parallelism, but requires more area.
Pipelining	Allows overlapping computations to improve throughput.

- **Back-End (Hardware Generation)**. Finally, the tools made two steps:
 1. **Converts optimized IR into Verilog/VHDL**.
 2. **Generates a Finite State Machine with Datapath (FSMD) representation**.

The result is a **synthesizable hardware description** ready for FPGA/ASIC implementation.

Therefore, a **final output** of HLS tools is:

- Verilog/VHDL code (for FPGA/ASIC synthesis).
- Datapath and Controller Design.
- Simulation files for verification.

✓ Conclusion

In conclusion, the key takeaways are:

- **HLS automates hardware design** from high-level code (C/C++/Python).
- The workflow involves **parsing, optimization, scheduling, and hardware generation**.
- **Performance tuning** requires adjusting **scheduling, pipelining, and parallelism**.
- **HLS outputs Verilog/VHDL**, which can be synthesized on FPGAs/ASICs.