Contents

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

```cpp
void box_blur(const Image &in, Image &out) {
    for (int y = 1; y < in.height() - 1; y++) {
        for (int x = 1; x < in.width() - 1; x++) {
            out(x, y) = (
                in(x-1, y) + in(x, y) + in(x+1, y)
            ) / 3;
        }
    }
}
```

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:

```cpp
Var x, y;
Func blurx, blury;

// First pass: horizontal blur
blurx(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))
    / 3;

// Second pass: vertical blur
blury(x, y) = (blurx(x, y-1) + blurx(x, y) + blurx(
    x, y+1)) / 3;
```

This part **only describes the math**, **<u>not how</u>** it should run.

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

```cpp
blury.tile(x, y, xi, yi, 256, 32)
    // Vectorized execution for SIMD
    .vectorize(xi, 8)
    // Parallel execution over y-dimension
    .parallel(y);
```

```

blurx.compute_at(blury, x)  // Compute blurx only
    when needed by blury
    .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.