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.