Dataflow Graph Generation Using LLVM: A Tool for Guiding High-Level Synthesis and Hardware Optimization

Yen-Chieh Huang
r13921038@ntu.edu.tw

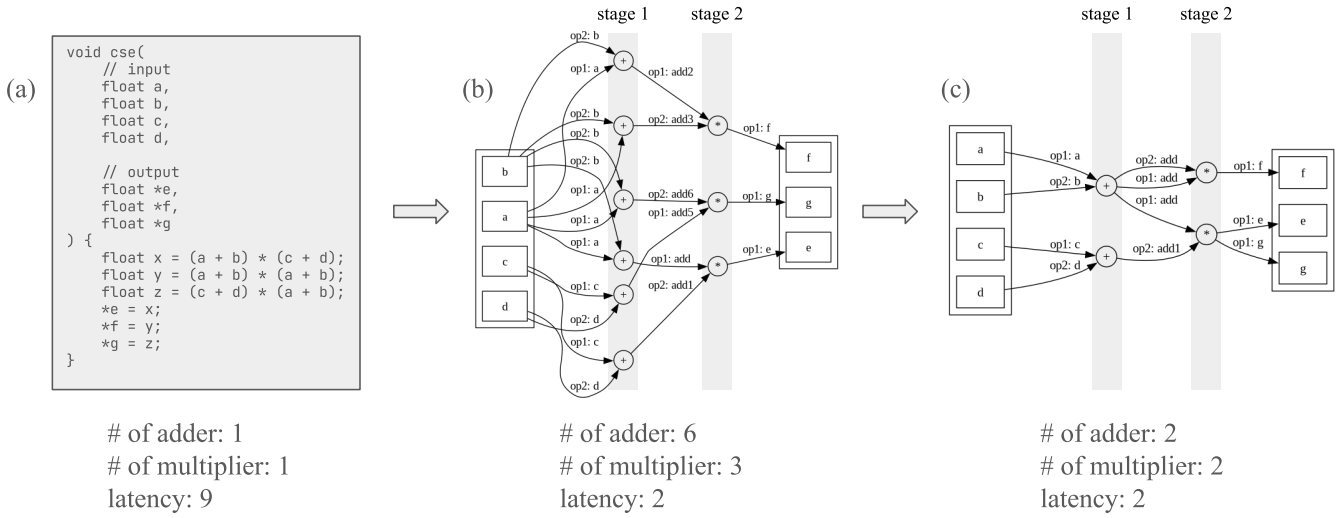

Figure 1: A motivating example for the `DataflowGenerator` LLVM pass: (a) The original C++ program runs sequentially, with each instruction executed one after another, resulting in a total latency of 9 cycles under the assumption that each operator takes exactly one cycle. (b) The dataflow graph generated by the LLVM pass represents all computation dependencies and executes with 6 adders and 3 multipliers in parallel, divided into multiple pipeline stages as highlighted in grey, reducing the latency to 2 cycles. (c) The optimized dataflow graph, after applying common subexpression elimination (CSE), minimizes redundancy, requiring only 2 adders and 2 multipliers, and achieving the same latency of 2 cycles with execution again divided into two pipeline stages. This progression demonstrates the pass's ability to uncover optimization opportunities and guide HLS tools and RTL engineers in designing efficient hardware.

1 Introduction

High-Level Synthesis (HLS) bridges the gap between high-level programming languages and register-transfer level (RTL) hardware representations. As the complexity of modern hardware accelerators increases, tools that analyze high-level code and provide actionable insights for hardware implementation have become indispensable. This report presents the design and implementation of a custom LLVM pass, `DataflowGenerator`, which generates dataflow graphs from C++ functions. Unlike traditional optimization passes, the primary purpose of this pass is not to optimize the original program but to produce dataflow graphs that guide HLS tools and RTL engineers in optimizing hardware implementations.

The `DataflowGenerator` pass leverages the LLVM framework to translate C++ code into dataflow graphs that capture computational dependencies. These graphs reveal optimization opportunities, such as parallelism and redundancy elimination, that are critical for hardware efficiency. For example, Figure 1 demonstrates the transformation from sequential execution to pipelined dataflow designs, showcasing the utility of the pass in hardware synthesis workflows.

Beyond graph generation, the pass integrates with LLVM's built-in optimization passes, such as dead code elimination (DCE) and common subexpression elimination (CSE). These transformations refine the generated graphs, improving their utility for hardware design by reducing resource usage and execution latency. A diverse set of test cases is used to validate the functionality of the pass, covering basic operations (e.g., copy propagation) and complex algorithms (e.g., Sobel edge detection, matrix multiplication).

In addition, a custom verifier ensures that the generated dataflow graphs are semantically equivalent to the original programs, confirming their correctness. This report details the design, implementation, and evaluation of the `DataflowGenerator` pass, emphasizing its potential to bridge the gap between software-level representations and hardware-optimized implementations. Future directions include integrating domain-specific optimizations, such as energy-efficient hardware transformations, to extend its applicability further.

2 Implementation Details

The `DataflowGenerator` LLVM pass is designed to analyze C++ functions and generate dataflow graphs that represent computational dependencies. This section details the methodology, key components, and challenges in the pass's implementation.

2.1 Graph Generation Process

The `DataflowGenerator` is implemented as an LLVM function pass using the PassPlugin infrastructure. It processes each function independently, traversing its basic blocks and mapping operations to nodes in a dataflow graph. Key steps include:

- **Node Identification:** Each input, operation, and output is assigned a unique node. Input nodes are labeled `in#`, while output nodes are labeled `out#`.
- **Edge Construction:** Edges represent dependencies between nodes. For instance, operands of binary operations connect to their corresponding operation node, while results connect to subsequent instructions or output nodes.
- **Graph Output:** The pass outputs the generated dataflow graph in DOT format, suitable for visualization tools like Graphviz.

The process is automated using the provided script `tests/run.sh`, which compiles input programs, applies LLVM passes (e.g., DCE, CSE), and generates the corresponding unoptimized and optimized dataflow graphs. This workflow streamlines the validation of test cases and enables quick iterations during development.

2.2 Handling LLVM Instructions

The pass focuses on binary operations and store instructions as the primary computational constructs. For each operation:

- Binary operations (`add`, `sub`, `mul`, `div`) are mapped to circular nodes labeled with the corresponding operator symbol (`+`, `-`, `*`, `/`).
- Store instructions are represented as edges connecting computational results to output nodes.
- Other instructions, such as `return`, are ignored as they do not contribute directly to the dataflow representation.

2.3 Unoptimized vs. Optimized Dataflow Graphs

To demonstrate the capability of the `DataflowGenerator` pass, Figure 2 and Figure 3 show the unoptimized and optimized dataflow graphs generated for `dce.cpp`, respectively.

In Figure 2, the graph represents all computations in the original program, including unnecessary or "dead" computations such as the multiplication which does not contribute to the final output. This unoptimized graph requires additional hardware resources and longer execution time.

In Figure 3, the redundant computations have been removed using LLVM's built-in Dead Code Elimination (DCE) pass, leaving only the essential computations required to produce the output. This optimization reduces the number of nodes and edges, decreasing hardware costs and execution cycles.

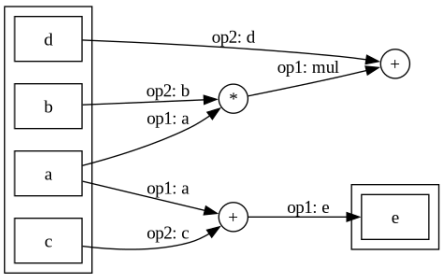

Figure 2: Unoptimized dataflow graph for `dce.cpp`. Redundant operations such as the multiplication are present.

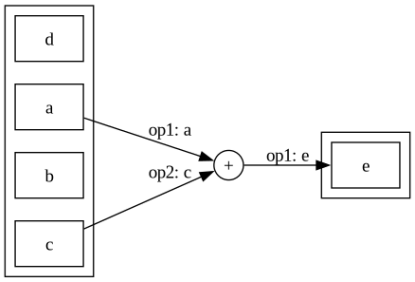

Figure 3: Optimized dataflow graph for `dce.cpp`. Redundant operations are removed using LLVM's built-in DCE pass.

2.4 Integration with LLVM Pipeline

The pass integrates seamlessly with the LLVM pipeline using the PassPlugin API, allowing it to be invoked via the -passes="dataflow-generator" command-line option. Additionally, it supports chaining with other LLVM passes to produce optimized dataflow graphs.

2.5 Challenges and Solutions

Several challenges were addressed during development:

- **Mapping LLVM Values to Graph Nodes:** The SSA (Static Single Assignment) nature of LLVM complicates the mapping of intermediate values. This was resolved by maintaining a dense map between LLVM IR values and graph node identifiers.
- **Unsupported Operations:** Operations not contributing directly to dataflow (e.g., `return`) are safely ignored to ensure compatibility with diverse inputs.
- **Readability of Generated Graphs:** To improve clarity, inputs and outputs are grouped into clusters, and operand names are used to label edges.

3 Correctness Proof

The correctness of the `DataflowGenerator` LLVM pass is critical to ensure that the generated dataflow graphs accurately represent the semantics of the original C++ functions. This section provides a formal and practical validation of correctness through invariant preservation, edge case analysis, and a dedicated verification tool.

3.1 Invariant Preservation

The following invariants are preserved by the `DataflowGenerator` pass:

- **Semantic Equivalence:** Each node in the generated graph corresponds to an instruction, input, or output in the original function, and each edge represents a true data dependency. No computational operation or data dependency is omitted or incorrectly represented.
- **Dependency Preservation:** The edges in the graph strictly adhere to the operand relationships in the LLVM IR. For example, the operands of a binary operation connect to the operation node, and the result connects to its subsequent use.
- **Data Integrity:** Input values are correctly propagated through the graph, ensuring the output values computed from the graph match those produced by the original function.

3.2 Edge Case Analysis

The pass handles edge cases to ensure robustness:

- **Unsupported Instructions:** Instructions that do not contribute directly to the dataflow (e.g., `return` instructions) are safely ignored without impacting graph accuracy.
- **Empty Functions:** Functions with no computational instructions result in graphs with only input and output nodes, maintaining structural correctness.

3.3 Verification Tool

A custom verification tool, `src/verifier.cpp`, was implemented to validate the correctness of the generated dataflow graphs. The tool performs the following steps:

(1) **Graph Parsing:** The tool parses the DOT representation of the graph to reconstruct its structure.
(2) **Input Generation:** Randomized input values are generated and assigned to the input nodes.
(3) **Graph Evaluation:** The graph is evaluated node by node, propagating values through the edges and computing results based on operation nodes.
(4) **Output Comparison:** The computed outputs from the graph are compared against the outputs of the original C++ function, executed with the same randomized inputs.

3.4 Validation Results

The verification tool was executed across all test cases, including both basic optimizations (`copy_prop.cpp`, `cse.cpp`, `dce.cpp`) and complex algorithms (`lagrange.cpp`, `matrix.cpp`, `ray_trig.cpp`, `rotation.cpp`, `sobel.cpp`). The tool validated the correctness of the generated graphs for all test cases, with no mismatches detected between the graph outputs and the original program outputs.

The results confirm that the dataflow graphs produced by the `DataflowGenerator` pass are semantically equivalent to the original C++ functions. This guarantees the reliability of the graphs as a foundation for guiding hardware optimizations during high-level synthesis. The correctness proof further demonstrates that the `DataflowGenerator` LLVM pass faithfully represents the computations of the original programs in its generated dataflow graphs, ensuring its utility as a reliable tool for hardware designers and HLS frameworks.

4 Discussion

While the `DataflowGenerator` LLVM pass effectively demonstrates the generation of dataflow graphs from C++ programs, it has certain limitations that should be acknowledged:

4.1 Operation Limitations

The current implementation supports a limited set of operations:

- Only basic arithmetic operations (`+`, `-`, `*`, `/`) are supported.
- Branching constructs, such as conditional statements and loops, are not currently handled.
- Constants and other advanced operations (e.g., bitwise operations) are not represented in the dataflow graph.

These limitations are primarily due to the scope of this implementation, which aims to demonstrate the feasibility of generating dataflow graphs using LLVM passes. Extending the pass to handle branching, constants, and additional operations is largely an engineering task, requiring further development but no fundamental changes to the approach.

4.2 Key Distinction from AST-Based Graph Generation

A significant advantage of generating dataflow graphs from LLVM IR, rather than directly analyzing the abstract syntax tree (AST), lies in the insights provided by the intermediate representation. Unlike AST-based approaches, which operate on high-level language constructs, LLVM IR represents the program at a lower level, capturing detailed computational dependencies and enabling more advanced optimizations.

For example, the teaser image (Figure 1) demonstrates how high-level C++ code transitions through LLVM IR to a dataflow graph. This process reveals opportunities for optimization by explicitly representing computational dependencies and allowing redundant operations to be identified and removed. Similarly, the unoptimized and optimized dataflow graphs for `dce.cpp` (Figures 2 and 3) illustrate the power of LLVM IR-based graph generation. The unoptimized graph includes all computations, including dead code, while the optimized graph, produced after applying LLVM's built-in DCE pass, retains only the necessary computations. These examples highlight how LLVM IR facilitates graph-based optimization that would be challenging to achieve at the AST level.

Furthermore, LLVM IR abstracts away high-level language features, providing a representation closer to machine-level instructions. This lower-level abstraction ensures more accurate modeling of dependencies, parallelism opportunities, and control flows. Such precision is critical for hardware design, making LLVM IR-based graph generation particularly well-suited for guiding HLS tools and hardware optimizations.

The distinction is also evident in the ability of LLVM IR to model real-world constraints such as hardware resource utilization and execution timing, which are often missing in AST-based approaches. Additionally, one can integrate custom hardware-aware LLVM passes—optimized for lower area, energy consumption, or

other specific metrics—to further refine the generated dataflow graphs. These hardware-aware passes could introduce domain-specific transformations that adapt the graph for more efficient implementation on specialized architectures, extending the utility of the `DataflowGenerator` pass in hardware design workflows.

5 Experimental Evaluation

The `DataflowGenerator` LLVM pass was evaluated using diverse test cases to analyze its impact on execution latency and hardware resource usage. By generating dataflow graphs, this pass enables RTL engineers to implement pipelined hardware designs, which can achieve significant performance gains compared to sequential execution.

5.1 Methodology

To evaluate the effectiveness of the generated dataflow graphs, three implementation strategies were compared:

- **Sequential Execution:** The latency is calculated by analyzing the unoptimized LLVM IR, assuming each operation takes one cycle. This corresponds to a non-pipelined execution model.
- **Dataflow Execution:** The latency corresponds to the number of pipeline stages in the generated dataflow graph. Fully pipelined execution allows a throughput of one result per cycle.
- **Optimized Dataflow Execution:** Optimizations such as dead code elimination (DCE) and common subexpression elimination (CSE) are applied to refine the dataflow graph, reducing latency and hardware usage.

Consider a function with n independent operations. In sequential execution, the latency is n cycles, assuming one operation per cycle. In a fully pipelined dataflow design, the latency is determined by the number of pipeline stages s, where $s \leq n$. Optimization further reduces s by eliminating redundant operations or dependencies, as demonstrated in `dce.cpp`, where the latency decreases from 2 to 1 cycle.

For each test case, latency and hardware usage were measured for all three strategies. The results are summarized in Tables 1 and 2.

5.2 Test Case Descriptions

The test cases demonstrate various computational scenarios:

- `cpy` (Copy Propagation): Simplifies assignments by eliminating redundant variables.
- `cse` (Common Subexpression Elimination): Eliminates repeated computations of identical expressions.
- `dce` (Dead Code Elimination): Removes computations with no effect on program output.
- `lag` (Lagrange Interpolation): Involves polynomial computations with multiple divisions and multiplications.
- `mat` (Matrix Multiplication): Performs matrix operations, showcasing parallel computation paths.
- `ray` (Ray-Triangle Intersection): Executes geometric computations requiring multiple cross-products.
- `rot` (Quaternion Rotation): Rotates vectors using quaternion-based transformations.
- `sob` (Sobel Operator): Detects edges in images by computing gradients.

5.3 Results and Analysis

Latency Analysis: Sequential execution exhibits the highest latency as each operation is executed in a cycle-by-cycle manner. Dataflow execution significantly reduces latency by exploiting parallelism, with fully pipelined designs achieving minimal latency. Optimized dataflow execution further reduces latency by removing redundant operations, as seen in `dce.cpp`, where latency drops from 2 to 1 cycle. The latency results are presented in Table 1.

Table 1: Latency Comparison Across Test Cases

Test Case	cpy	cse	dce	lag	mat	ray	rot	sob
Sequential	1	9	3	26	12	45	46	17
Dataflow	1	2	2	6	2	7	7	6
Optimized	1	2	1	6	2	7	7	6

Hardware Usage Analysis: Dataflow execution incurs significantly higher hardware usage than sequential execution, as each operation in the graph maps to a dedicated hardware unit. Optimized dataflow execution reduces this by eliminating redundant operations, as seen in `cse.cpp`, where hardware requirements decrease from 9 to 4 units. Table 2 summarizes the hardware resource usage.

Table 2: Hardware Resource Usage Across Test Cases

Test Case	cpy	cse	dce	lag	mat	ray	rot	sob
Sequential	1	2	2	4	2	4	3	3
Dataflow	1	9	3	26	12	45	46	17
Optimized	1	4	1	23	12	45	45	17

While fully pipelined dataflow execution achieves minimal latency, it comes at the cost of high hardware resource usage. To address this trade-off, multi-cycle implementations of dataflow graphs can be employed. In a multi-cycle design, the same hardware units are reused over multiple cycles to perform different operations. This approach reduces the overall hardware requirement significantly while still maintaining relatively low latency compared to sequential execution. For instance, instead of allocating 6 adders and 3 multipliers as in Figure 1(b), a multi-cycle design could reuse one adder and one multiplier across multiple cycles, achieving a balance between resource usage and performance.

5.4 Discussion

The results highlight the trade-offs in dataflow execution:

- **Latency Benefits:** Fully pipelined designs achieve minimal latency, making them suitable for real-time applications such as image processing (`sob.cpp`) and geometric computations (`ray.cpp`).
- **Hardware Costs:** Pipelined designs require higher hardware usage due to dedicated units for each operation. Optimized dataflow execution reduces these costs by eliminating redundant operations.

- **Multi-Cycle Designs:** To address the resource usage challenge, multi-cycle implementations of dataflow graphs can be employed. By reusing hardware units across cycles, resource usage is reduced, offering a practical trade-off between latency and hardware efficiency.

As seen in Figure 1, the progression from sequential execution to pipelined designs and further optimization underscores the utility of the `DataflowGenerator` pass in hardware design workflows.

6 Conclusion

The `DataflowGenerator` LLVM pass demonstrates its utility in bridging high-level programming constructs and hardware design by generating accurate and insightful dataflow graphs. These graphs provide a clear representation of computational dependencies, enabling RTL engineers and HLS tools to design hardware that achieves high performance and efficiency.

The experimental evaluation highlights the benefits of transitioning from sequential to dataflow execution. By leveraging pipelined designs, dataflow execution minimizes latency and achieves high throughput, making it suitable for performance-critical applications. Furthermore, the integration of LLVM's built-in optimization passes, such as dead code elimination and common subexpression elimination, enhances the generated dataflow graphs by reducing redundant operations, further improving resource efficiency.

Despite the advantages of pipelined execution, the increased hardware resource usage presents a trade-off. However, alternative implementation strategies, such as multi-cycle designs, offer a viable solution by reusing resources across cycles to balance performance and cost. These flexible design options underscore the versatility of the `DataflowGenerator` pass in catering to diverse hardware requirements.

By accurately representing optimized computational dependencies, the `DataflowGenerator` pass serves as a valuable tool for guiding the synthesis and optimization of hardware designs. This work demonstrates the potential of LLVM-based tools to streamline the hardware design process and lays the groundwork for future advancements in dataflow analysis and optimization for hardware applications.

References

[1] Adrian Sampson. 2023. *sampsyo/llvm-pass-skeleton.* https://github.com/sampsyo/llvm-pass-skeleton
[2] Andrzej Warzyński. 2024. *banach-space/llvm-tutor.* https://github.com/banach-space/llvm-tutor