MP3 Report

Nachuan Wang(nachuan3)
Liangyu Zhou(liangyu5)

Part A: Matrix Multiply

Assumptions:

Assume that we can decide the factor number of unrolling and in which loop to place the FIFO pragma.

Description:

Version 1-1: Original 3-loop matrix multiplication.

Version 1-2: Put pipeline pragma to the most inner loop to pipeline the entire process.

Version 1-3: Put unroll pragma to the most inner loop to parallelize this loop.

Version 2-1: Replace the memory-mapped array interface in version 1-1 with FIFO by placing pragmas to first loop (for port C only).

Version 2-2: Replace the memory-mapped array interface in version 1-2 with FIFO by placing pragmas to first loop (for port C only).

Version 2-3: Replace the memory-mapped array interface in version 1-3 with FIFO by placing pragmas to first loop (for port C only).

Changes of code:

All changes of code are described in previous section.

Limitation/Modification:

No limitation in Part A. No need to modify algorithms.

Total resource utilization of all versions:

Version	BRAM_18K	DSP48E	FF	LUT	URAM
(Memory) Unoptimized	0	3	257	376	0
(Memory) Loop pipelining	0	4	283	572	0
(Memory) Loop unrolling	0	7	426	709	0
(FIFO) Unoptimized	0	3	257	376	0
(FIFO) Loop pipelining	0	4	242	438	0
(FIFO) Loop unrolling	0	7	426	709	0

Estimated latency for all versions:

Version	Latency		
	Min	Avg	Max
(Memory) Unoptimized	2020201	2323608	4020201
(Memory) Loop pipelining	2000004	2000004	2000004
(Memory) Loop unrolling	1270201	1573608	3270201
(FIFO) Unoptimized	2020201	2323608	4020201
(FIFO) Loop pipelining	2000501	2000501	2000501
(FIFO) Loop unrolling	1270201	1573608	3270201

Modification:

We did not modify algorithm. We also did not do anything in particular to leverage interface or optimization directive.

Version that is not possible:

No version is impossible if we use pragmas in appropriate way.

Beneficial Application/Scenario:

Version 1-1: Hardware resources are limited.

Version 1-2: Abundant hardware resources (for registers) but limited memory bandwidth.

Version 1-3: Abundant hardware resources, loop iterations are independent of each other (so that they can be parallelized), large memory bandwidth.

Version 2-1: Limited hardware resource and sequential memory access.

Version 2-2: Abundant hardware resources (for registers) and sequential memory access, but limited memory bandwidth.

Version 2-3: Abundant hardware resources, sequential memory access and large memory bandwidth, especially when seeking for high performance/throughput.

Why latency varies:

Version 1-1: It has longest latency since no optimization is applied in this case.

Version 1-2: It has smaller latency and uses slightly more hardware resources than version 1-1 because a pipeline is implemented.

Version 1-3: It has much smaller latency and uses much more hardware resources than version 1-1 because the inner loop is parallelized by factor of 4.

Version 2-1, 2-2, 2-3: They have roughly same latency with their correspondence in version 1, but all of them use less hardware resources because a FIFO interface is implemented.

Latency Calculation:

Version 1-1: In the unoptimized version the latency is calculated by summing the memory access time and time for each loop iteration to complete (one add plus one multiplication); we can infer the "average memory access time + time for one add and one multiplication"; we can estimate the latency only if we know the memory access time and time to perform add/multiply.

Version 1-2: In the pipelined version the latency is calculated by summing, memory access time, the time to fill up the pipeline and number of items in C; we can infer the average memory access time; we can estimate the latency by adding the memory access time, the depth of pipeline and number of items in C.

Version 1-3: In the unrolled version the latency is calculated by memory access time and time for all loop iteration to complete divided by unrolling factor; we can infer nothing from the latency if do not know the unrolling factor ahead; we can estimate the latency if we know how memory bandwidth compares to the through put of computing circuit.

Version 2-1: In this case FIFO interface does not add to latency, so all statements for version 1-1 still hold for version 2-1.

Version 2-2: In this case FIFO even adds some extra latency, so beside latencies mentioned in version 1-2 we also need to know the latency for FIFO.

Version 2-3: In this case FIFO interface does not add to latency, so all statements for version 1-3 still hold for version 2-3.

Difficulties/Bugs:

We first declared FIFO port outside all loops, but we soon found that the code cannot pass simulation, we then put it inside the first loop to resolve the problem.

What you learned:

We learned how each pragma means to the hardware implementation of C code. We also learned how they effect the performance of hardware in various ways.

Vivado Version:

We used Vivado HLS 2019.1 on our personal computers with Windows.

Part B: Matrix Multiply with internal buffers

Assumptions:

In this part, we assume that we should have a balance between latency and utilization, which means we should not pursue latency regardless of the utilization.

Description:

For the unoptimized version, we add an internal buffer to read and write back the data onto Part A. The entire process contains a read loop, calculation loop, and write loop. And we also regulate the A, B, and C ports to be FIFO interfaces. And this version is the base version that we used in this part. Each version below is optimized based on this version.

For Pipelined read and write version, we pipelined the read input and write back output loop to reduce latency. It is noteworthy that we tried to change the calculation loop into pipelined but the performance was reduced a lot.

For the loop unrolling version, we unroll the calculation loop with a factor of 4, which we think is the perfect balance between latency and utilization.

For the optimized version, we add all loop unrolling and pipelining optimization method into the base version to pursue the minimum latency.

Changes of code:

For the unoptimized version, we change the code into 3 for loops from part A. And we also regulate the A, B, and C ports to be FIFO interfaces by changing the "directive" tag.

For pipelined read and write version, we add HLS PIPELINE pragmas inside the read loop and write loop.

For the loop unrolling version, we add HLS UNROLL pragma into the calculation loop, whose unroll factor is 4.

For the optimized version, we combined all the changes to finalize our MP.

Limitation/Modification:

The limitation of the optimization is the utilization. We could not unroll the loop as much as we want. We did not modify the basic algorithm. To improve deeper of the circuit, we should have more resources to both read and calculate the data.

Total resource utilization of all versions:

Version	BRAM_18K	DSP48E	FF	LUT	URAM

Unoptimized	96	3	370	800	0
Pipelined read and write	96	3	379	996	0
Loop unrolling	96	7	539	1111	0
Optimized	96	7	548	1306	0

Estimated latency for all versions:

Version	Latency	
	Min	Max
Unoptimized	2050603	4050603
Pipelined read and write	2040206	4040206
Loop unrolling	1300603	3300603
Optimized	1290206	3290206

Application:

For the FIFO interface optimization, we think we can use this optimization when the resource is limited.

For pipelined read and write, this optimization can consume more resources but lower latency, we can use this optimization when reading the data from memory.

For loop unrolling, this optimization has a significant influence when we calculate data. we should use this method when calculating consisting the most portion of the latency.

For internal buffer, we can use this method when the data reuse rate is high in a specific algorithm.

Why latency varies:

Using the FIFO interface does not influence in terms of latency.

Using pipelined read can reduce the latency of the read loop from 10200 to 10000 because we overlap the input and output cycle to some extent.

Using loop unrolling we reduce the latency of the calculation loop significantly because we use more resources in this optimization.

Latency calculation:

For the basic version, in the reading loop, each read takes 2 cycles. For the inner loop, we read 100 data at a time, so the total latency is 102 cycles (the FIFO interface is inherently pipelined). And we have 100 times outer loop. So, we have a latency of 10200 for the read loop. If we pipeline the read loop, the performance won’t improve very much because FIFO is inherently pipelined. For the adder loop, we have 3 cycles to do the multiply and 1 cycle to do the add, which is 4 cycles in total. We need to have 100*100*100 times calculation. We have

4 *100*100*100 cycles to implement the calculation part. If we unroll the calculation loop by a factor of 4, theoretically we should have 4 times improvement compared to the unchanged version. But if we do the unrolling, the calculation cost will get higher a little bit, so the actual result will be more than 100*100*100. We need 2 cycles to write back each result for the writing back loop. So, the total latency is 2*100*100 cycles. If we pipeline the writing back loop, we can have a better latency of 100*100 + 2 cycles.

Difficulties:

We found that we shouldn't use an undefined loop limit for loop in C code HLS. We should use a fixed loop limit instead.

What learned:

We learned how to use HLS tools to design hardware in software programming.

Vivado version:

We used Vivado HLS 2019.1 on our personal computers with Windows.