POLITECNICO
MILANO 1863

DIGITAL SYSTEM DESIGN & METHODOLOGIES II

A.Y. 2020/2021

Optional project report

High Level Synthesis – Optimization of a hardware accelerator

Gioele Mombelli

Personal Code: 10520552
gioele.mombelli@mail.polimi.it

0 – Introduction and goals

I have developed this project as a part of the course "Digital System Design & Methodologies" at Politecnico di Milano. The goal of the project is to learn the fundamentals of High Level Synthesis and its tools, with a particular focus on Vitis HLS by Xilinx (the new version of the well-known Vivado HLS) and the features that it provides.

This work is not meant to be more than an academic exercise. I will briefly explain in this report the way of reasoning behind the project, the motivations that driven each solution and the issues faced during the development.

Sources and study material

- *Lectures of the course "Digital System Design & Methodologies" by Professor Pilato and Professor Soldavini (AY 2020/2011);*
- *Vivado Design Suite User Guide – High Level Synthesis - UG902 (v2017.4) February 2, 2018 by Xilinx;*
- *Lecture "Introduction to HLS" by Simone Bologna – University of Bristol;*
- *Vivado HLS: debug guide for investigating C/RTL co-simulation issues by Xilinx;*
- *Designing High-Quality Hardware on a Development Effort Budget: A Study of the Current State of High-Level Synthesis by Sun, Zuo, Campbell, Rupnow, Gurumani, Doucet, Chen.*

During the whole project I referred multiple times to the lectures provided by Professor Pilato and Professor Soldavini during the course. While the lectures had a fundamental role in my understanding of the dynamics behind HLS, I also had to search for additional material in order to have a deeper understanding of the processes involved. Last but not the least, some advice and guidance by the teachers themselves have been crucial for the development of the project.

1 – Project overview

The focus of the project was to synthesize and optimize a hardware accelerator. The original code was provided by Professor Pilato and professor Soldavini (file: kernel_gemm.c).

High Level Synthesis allows a designer to obtain a hardware implementation of the functionalities described by a program written in a high level language, such as C or C++, without having to directly write it in a hardware description language, such as VHDL or Verilog. While this makes the process easier for the designer, at least for the part related to bridging the gap between high level algorithms and their hardware implementation, it also requires some skills and knowledge in order fine-tune the synthesis tool and to obtain better performances. One of the biggest issues I faced during the development was indeed predicting the effects of a modification or a directive: some solutions that I thought wouldn't perform well got instead good results, while other solutions that I believed to be optimized worsened the performances of the component.

First analysis of the source code

I started by performing a manual inspection of the code, in order to check if the program was correctly synthesizable. The original source code didn't include any system calls (such as memory allocation functions like malloc or calloc), any recursive functions (which are not synthesizable) and its constructs had a finite or bounded dimension. This first inspection confirmed that the functionality could be implemented in hardware. A guidance and more information on the synthesizability of a generic C code and the properties that must be satisfied can be found in the Xilinx HLS guide.

The second analysis consisted in reading the code to get an understanding of the operations performed. Since the code relied heavily on operations between matrices and tensors, it was particularly prone to a hardware implementation in order to be able to exploit the implicit parallelism of the technology. The various loops executed in the code were the parts that could be exploited in order to obtain improvements, so one of the first step I did was to label each of them. I will briefly report here the original source code with the labels:

[file: src/vanilla/kernel_gemm.c]

```
1  /*inverse_helmholtz_gemm.c*/
2
3  /* Copyright (C) 2017-2020 TU Dresden */
4  /* All Rights Reserved                */
5  /*                                    */
6  /* Authors: Gerald Hempel             */
7  /*          Karl Friebel              */
8  #define X    11
9  #define Y    11
10 #define Z    11
11
12 /* general 2D matrix multiplication */
13 void gemm(
14   unsigned m1,
15   unsigned n1,
16   unsigned m2,
17   unsigned n2,
18   const double in1[m1][n1],
19   const double in2[m2][n2],
20   double out[m1][n2]
21 )
22 {
23   GEMM_LOOP1: for (unsigned i = 0; i < m1; i++)
24     GEMM_LOOP2: for (unsigned j = 0; j < n2; j++) {
25       out[i][j] = 0;
26       GEMM_LOOP3: for (unsigned k = 0; k < m2; k++) {
27         out[i][j] += in1[i][k] * in2[k][j];
28       }
29     }
30 }
31
32 /** Transposes a matrix. */
33 void matrix_transpose(const double in[X][Y], double out[Y][X])
34 {
35   MTRANS_LOOP1: for (unsigned i = 0; i < X; ++i)
36     MTRANS_LOOP2: for (unsigned j = 0; j < Y; ++j)
37       out[j][i] = in[i][j];
38 }
39 /** Reshapes and transposes a tensor [X, Y, Z] into [Z, X*Y]. */
40 void reshape_transpose(const double in[X][Y][Z], double out[Z][X*Y])
41 {
42   RES_LOOP1: for (unsigned i = 0; i < X; ++i)
43     RES_LOOP2: for (unsigned j = 0; j < Y; ++j)
44       RES_LOOP3: for (unsigned k = 0; k < Z; ++k)
45         out[k][i*Y + j] = in[i][j][k];
46 }
47 /** Transposes and reshapes a tensor [Z, X*Y] into [X, Y, Z]. */
48 void transpose_reshape(const double in[Z][X*Y], double out[X][Y][Z])
49 {
50   TRANSRES_LOOP1: for (unsigned i = 0; i < X; ++i)
```

```
51      TRANSRES_LOOP2: for (unsigned j = 0; j < Y; ++j)
52        TRANSRES_LOOP3: for (unsigned k = 0; k < Y; ++k)
53          out[i][j][k] = in[k][i*Y + j];
54  }
55  /** Transposes a tensor [X, Y*Z] into [Z, X*Y]. */
56  void tensor_transpose(const double in[X][Y*Z], double out[Z][X*Y])
57  {
58    TTRANS_LOOP1: for (unsigned i = 0; i < X; ++i)
59      TTRANS_LOOP2: for (unsigned j = 0; j < Y; ++j)
60        TTRANS_LOOP3: for (unsigned k = 0; k < Z; ++k)
61          out[k][i*Y + j] = in[i][j*Y + k];
62  }
63
64  void kernel_body(
65    const double S[X][Y],
66    const double D[X][Y][Z],
67    const double u[X][Y][Z],
68    double r[X][Y][Z],
69    double S_T[Y][X],
70    double t1[Z][X*Y],
71    double t2[Z][X*Y]
72  )
73  {
74    reshape_transpose(u, t1);
75
76    gemm(X, Y, Z, X*Y, S, t1, t2);
77    tensor_transpose(t2, t1);
78    gemm(X, Y, Z, X*Y, S, t1, t2);
79    tensor_transpose(t2, t1);
80    gemm(X, Y, Z, X*Y, S, t1, t2);
81
82    matrix_transpose(S, S_T);
83
84    MULT_LOOP1: for (unsigned i = 0; i < X; ++i)
85      MULT_LOOP2: for (unsigned j = 0; j < Y; ++j)
86        MULT_LOOP3: for (unsigned k = 0; k < Z; ++k)
87          t2[i][j*Y + k] *= D[i][j][k];
88
89    gemm(X, Y,  X, Y*Z, S_T, t2, t1);
90    tensor_transpose(t1, t2);
91    gemm(X, Y,  X, Y*Z, S_T, t2, t1);
92    tensor_transpose(t1, t2);
93    gemm(X, Y,  X, Y*Z, S_T, t2, t1);
94
95    transpose_reshape(t1, r);
96  }
97
98
99  void kernel(
100   const double S[11][11],
101   const double D[11][11][11],
102   const double u[11][11][11],
103   double r[11][11][11]
104 )
105 {
106   double S_T[Y][X];
107   double t1[Z][X*Y];
108   double t2[Z][X*Y];
109
110   kernel_body(S, D, u, r, S_T, t1, t2);
111 }
112
```

Once the code has been analysed and the environment has been set up, the following step was about obtaining a test bench for the code.

2 – Test bench generation

Some time has been dedicated to the development and the generation of a test bench that could be used to test the correctness of the code through the process. The goal of the test bench was *not to test the correctness of the original code*, but to have a reliable input/output association obtained *from the original code* that could be used to check that the behaviour of the application remained constant through the whole process of synthesis and optimization. The test files, called “input” and “output”, were generated by the following code, which will be explained later:

[file: test/dataset_generator.c]

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include "kernel_gemm.c"

#define RANGE_MIN 0.0
#define RANGE_MAX 10.0

#define X 11
#define Y 11
#define Z 11

double random_double(){

double range = (RANGE_MAX - RANGE_MIN);
double div = RAND_MAX / range;
return RANGE_MIN + (rand() / div);

}

void write_input(const char* file){

FILE* input;
input = fopen(file, "w");
int i = 0;
int j = 0;
int k = 0;
srand(17);

//Section 1: writing of S matrix
for(i=0; i<X; i++){
    for(j=0; j<Y; j++){
    fprintf(input, "%.15lf\n", random_double());
    }
}


//Section 2: writing of D tensor:
for(i=0; i<X; i++){
    for(j=0; j<Y; j++){
            for(k=0; k<Z; k++){
                fprintf(input, "%.15lf\n", random_double());
            }
    }
}



//Section 2: writing of u tensor:
for(i=0; i<X; i++){
    for(j=0; j<Y; j++){
            for(k=0; k<Z; k++){
                fprintf(input, "%.15lf\n", random_double());
            }
    }
}
```

```
56
57  fclose(input);
58
59  }
60
61  void load_input(const char* file, double S[X][Y], double D[X][Y][Z], double u[X][Y][Z]){
62
63  FILE* input;
64  input = fopen(file, "r");
65  double number = 1.0;
66  int i = 0;
67  int j = 0;
68  int k = 0;
69
70  //Section 1: reading of S matrix
71  printf("S matrix:\n");
72  for(i=0; i<X; i++){
73      for(j=0; j<Y; j++){
74          fscanf(input, "%lf[^\n]", &number);
75          printf("S[%d][%d] = %.15lf\n", i, j, number);
76          S[i][j] = number;
77      }
78  }
79
80  //Section 2: reading of D tensor
81  printf("D tensor:\n");
82  for(i=0; i<X; i++){
83      for(j=0; j<Y; j++){
84              for(k=0; k<Z; k++){
85                  fscanf(input, "%lf[^\n]", &number);
86                  printf("D[%d][%d][%d] = %.15lf\n", i, j, k, number);
87                  D[i][j][k] = number;
88              }
89      }
90  }
91
92  //Section 3: reading of u tensor
93  printf("u tensor:\n");
94  for(i=0; i<X; i++){
95      for(j=0; j<Y; j++){
96              for(k=0; k<Z; k++){
97                  fscanf(input, "%lf[^\n]", &number);
98                  printf("u[%d][%d][%d] = %.15lf\n", i, j, k, number);
99                  u[i][j][k] = number;
100             }
101     }
102 }
103
104
105
106 }
107
108 void write_output(const char* file, double r[X][Y][Z]){
109
110 FILE* output;
111 output = fopen(file, "w");
112
113 printf("r tensor:\n");
114 for(int i=0; i<X; i++){
115     for(int j=0; j<Y; j++){
116             for(int k=0; k<Z; k++){
117                 printf("r[%d][%d][%d] = %.15lf\n", i, j, k, r[i][j][k]);
118                 fprintf(output, "%.15lf\n", r[i][j][k]);
119             }
120     }
121 }
122
123 fclose(output);
124
125 }
126
127 int main(){
128
129 double S[X][Y];
130 double D[X][Y][Z];
131 double u[X][Y][Z];
132 double r[X][Y][Z];
133
```

```
134  write_input("C:/Users/gio_m/Desktop/ProgettoDSDM/src/input");
135  printf("The following input test case has been generated:\n");
136  load_input("C:/Users/gio_m/Desktop/ProgettoDSDM/src/input", S, D, u);
137
138  printf("Start execution of original source code kernel_gemm...\n");
139
140  kernel(S, D, u, r);
141
142  printf("The following output has been generated:\n");
143
144  write_output("C:/Users/gio_m/Desktop/ProgettoDSDM/src/output", r);
145
146  printf("Test generation complete. You can find the results in the input file and in the output
file.\n");
147
148  return 0;
149  }
```

This code simply generates random floating point values between 0 and 10 and writes them to a file. There are just enough values to fill the inputs of the original source code: S, D, u and r. The values are then read from the file and used as inputs for an execution of the “kernel” function, the one to be synthesized. The output of the function is finally written to another file, called “output”. In this way I have obtained an input file and its associated output from the original source code. They can be found in the attached folder “test”.

The next step was to write the actual test bench that the HLS tool would have used. This is the code of the test bench:

[file: test/kernel_gemm_tb.c]

```
 1  #include <stdio.h>
 2  #include <stdlib.h>
 3  #include <math.h>
 4
 5  #define X 11
 6  #define Y 11
 7  #define Z 11
 8
 9  #define DOUBLE_EPSILON 1E-10
10
11  #define INPUT "C:/Users/gio_m/Desktop/ProgettoDSDM/src/input"
12  #define OUTPUT "C:/Users/gio_m/Desktop/ProgettoDSDM/src/output"
13
14  int almost_equal(double a, double b, double epsilon) {
15      printf("Epsilon: %.15lf\n", fabs(a-b));
16      return fabs(a-b) <= epsilon;
17  }
18
19  void load_input(const char* file, double S[X][Y], double D[X][Y][Z], double u[X][Y][Z]){
20
21  FILE* input;
22  input = fopen(file, "r");
23  double number = 1.0;
24  int i = 0;
25  int j = 0;
26  int k = 0;
27
28  //Loading 1: reading of S matrix
29  for(i=0; i<X; i++){
30      for(j=0; j<Y; j++){
31          fscanf(input, "%lf[^\n]", &number);
32          S[i][j] = number;
33      }
34  }
35
36  //Loading 2: reading of D tensor
37  for(i=0; i<X; i++){
38      for(j=0; j<Y; j++){
39              for(k=0; k<Z; k++){
40                  fscanf(input, "%lf[^\n]", &number);
41                  D[i][j][k] = number;
```

```
 42              }
 43      }
 44  }
 45
 46  //Loading 3: reading of u tensor
 47  for(i=0; i<X; i++){
 48      for(j=0; j<Y; j++){
 49              for(k=0; k<Z; k++){
 50                  fscanf(input, "%lf[^\n]", &number);
 51                  u[i][j][k] = number;
 52              }
 53      }
 54  }
 55
 56
 57
 58  }
 59
 60  int compare(const char* file, double r[X][Y][Z]){
 61
 62  FILE* output;
 63  output = fopen(file, "r");
 64  double current_value = -1.0;
 65
 66  for(int i=0; i<X; i++){
 67      for(int j=0; j<Y; j++){
 68              for(int k=0; k<Z; k++){
 69                  fscanf(output, "%lf[^\n]", &current_value);
 70                  printf("Current value: %.15lf - Test value: %.15lf\n", current_value,
r[i][j][k]);
 71
 72                  if(!almost_equal(current_value, r[i][j][k], DOUBLE_EPSILON)){
 73                      fclose(output);
 74                      return(-1);
 75                  }
 76              }
 77      }
 78  }
 79
 80  fclose(output);
 81  return(0);
 82
 83  }
 84
 85  int main(){
 86
 87  double S[X][Y];
 88  double D[X][Y][Z];
 89  double u[X][Y][Z];
 90  double r[X][Y][Z];
 91
 92  load_input(INPUT, S, D, u);
 93
 94  kernel(S, D, u, r);
 95
 96  int test_result = compare(OUTPUT, r);
 97
 98  if(test_result == 0){
 99      printf("The test was successful!\n");
100      return 0;
101  }
102
103  if(test_result == -1){
104      printf("Test failed!\n");
105      return -1;
106  }
107
108  printf("Unexpected error.");
109  return 1;
110
111  }
```

The most useful resource in writing the test bench was the Xilinx HLS user guide, which provided me a complete explanation of the best practices that should be adopted when writing a test bench, such as the required return values. One of the issues I faced was the comparison of two floating

point numbers: the function at line 14 performs the comparison by stating that two values are equal if their difference is less than a chosen epsilon, in this case 10^{-10}. This seemed to me a reasonably small value in order to consider two numbers as equal, since in real application such a small difference could be neglectable most of the times.

3 – Solutions evaluation

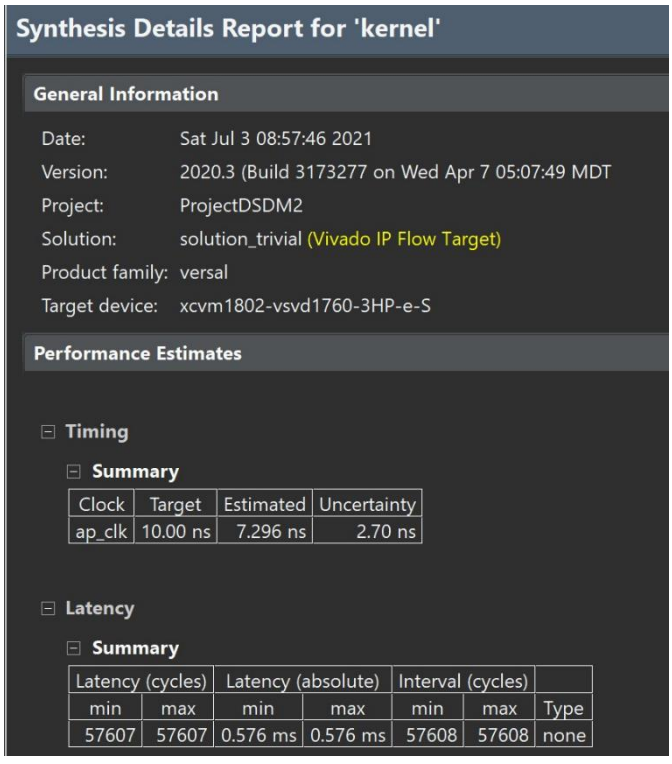
Between all the solutions I have tried, some of them turned out to be improvements while others didn't affect at all the overall performance, or even worsened it. Others brought an improvement but failed the Co-Simulation. Here I will report them in chronological order, providing explanations and the reasoning behind each one. The first solution I implemented was the trivial one, in order to have a reference value to evaluate the improvement.

Note: all the solutions provided here are obtained by running, in order, C simulation, C synthesis and C/RTL co-simulation. Every solution provided passes C simulation correctly.
The target clock is set as 10.0ns and the target device is the same for each solution presented here.

Solution 1: Trivial (with default pipeline)

Source file: *src/vanilla/kernel_gemm.c*
User defined directives*: none*
Latency: *57607 cycles*
Co-simulation result: *pass*

This solution is obtained by running HLS on the vanilla version of the code, without any optimization or directive applied. Actually, Vitis HLS tried to apply some default optimization like loop pipelining, but I decided to consider this as the starting point anyway. The synthesis required a few minutes and the following results were obtained:



Synthesis Details Report for 'kernel'

General Information

Date:	Sat Jul 3 08:57:46 2021
Version:	2020.3 (Build 3173277 on Wed Apr 7 05:07:49 MDT
Project:	ProjectDSDM2
Solution:	solution_trivial (Vivado IP Flow Target)
Product family:	versal
Target device:	xcvm1802-vsvd1760-3HP-e-S

Performance Estimates

Timing

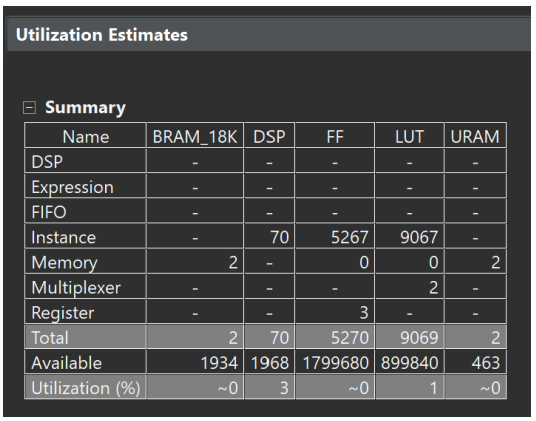
Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00 ns	7.296 ns	2.70 ns

Latency

Summary

Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
57607	57607	0.576 ms	0.576 ms	57608	57608	none



Utilization Estimates

Summary

Name	BRAM_18K	DSP	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	-	-	-
FIFO	-	-	-	-	-
Instance	-	70	5267	9067	-
Memory	2	-	0	0	2
Multiplexer	-	-	-	2	-
Register	-	-	3	-	-
Total	2	70	5270	9069	2
Available	1934	1968	1799680	899840	463
Utilization (%)	~0	3	~0	1	~0

Cosimulation Report for 'kernel'

General Information

Date:	Sat Jul 3 09:09:54 2021	Solution:	solution_trivial (Vivado IP Flow
Version:	2020.3 (Build 3173277 on Wed Apr 7 05:07:49 MDT	Product	versal
Project:	ProjectDSDM2	Target device:	xcvm1802-vsvd1760-3HP-e-S
Status:	Pass		

Cosim Options

Tool: Vivado XSIM RTL: Verilog

Performance Estimates

Modules	Avg II	Max II	Min II	Avg Latency	Max Latency	Min Latency
kernel				57607	57607	57607
kernel_body_1				57606	57606	57606

As we can see from the utilization estimates, the resource utilization is practically zero, but the number of clock cycles needed to complete the computation is quite high. This is an exhibition of the classic area/latency trade-off, so I decided to focus my project on improving the latency of the component since there is plenty of resource available. In order to do this, I relied on the insight given to me by Vitis, in particular the section that highlighted the various cycles needed to each loop:

Performance & Resource Estimates

Modules Loops

Modules && Loops	Issue Type	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined
kernel		-	57607	5,760E5	-	57608	-	no
kernel_body_1		-	57606	5,760E5	-	57606	-	no
gemm_1	II Violation	-	8020	8,020E4	-	8020	-	no
GEMM_LOOP1_GEMM_LOOP2	II Violation	-	8018	8,018E4	39	6	1331	yes
gemm	II Violation	-	8020	8,020E4	-	8020	-	no
GEMM_LOOP1_GEMM_LOOP2	II Violation	-	8018	8,018E4	39	6	1331	yes
RES_LOOP1_RES_LOOP2_RES_LOOP3		-	1334	1,334E4	5	1	1331	yes
TTRANS_LOOP1_TTRANS_LOOP2_TTRANS_LOOP3		-	1334	1,334E4	5	1	1331	yes
TTRANS_LOOP1_TTRANS_LOOP2_TTRANS_LOOP3		-	1334	1,334E4	5	1	1331	yes
MTRANS_LOOP1_MTRANS_LOOP2		-	124	1,240E3	5	1	121	yes
MULT_LOOP1_MULT_LOOP2_MULT_LOOP3		-	1336	1,336E4	7	1	1331	yes
TTRANS_LOOP1_TTRANS_LOOP2_TTRANS_LOOP3		-	1334	1,334E4	5	1	1331	yes
TTRANS_LOOP1_TTRANS_LOOP2_TTRANS_LOOP3		-	1334	1,334E4	5	1	1331	yes
TRANSRES_LOOP1_TRANSRES_LOOP2_TRANSRES_LOOP3		-	1334	1,334E4	5	1	1331	yes

As we can see, Vitis is trying to pipeline all the loops of the code with a II value of 1, but some of them result in a II Violation, and the initiation interval is longer. This gave me the idea of trying to partition the variables involved in the heaviest computations. But first, I tried out of curiosity to give manually the Pipeline directive, as seen in the next solution.

Solution 2: Pipelined

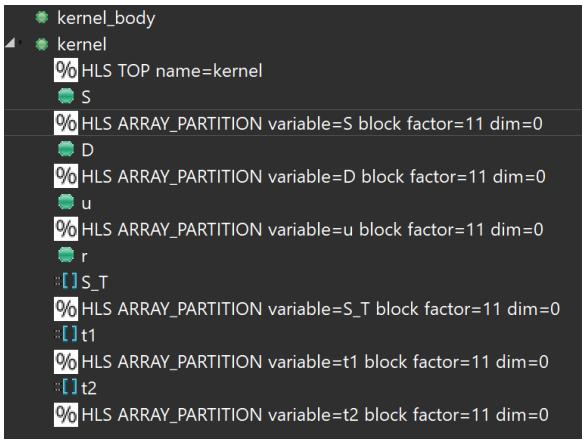
Source file: *src/vanilla/kernel_gemm.c*
User defined directives*: pipeline*
Latency: *57607 cycles*
Co-simulation result: *pass*

This solution will only be reported briefly since it didn't bring better performances. The only thing done here has been to manually apply the PIPELINE directive on the innermost loop of "gemm" function (GEMM_LOOP3). The fact that nothing changed in the results confirmed that Vitis already tried to pipeline the loops.

Solution 3: Pipelined and Partitioned

Source file: *src/vanilla/kernel_gemm.c*
User defined directives*: pipeline and partition*
Latency: *50974 cycles*
Co-simulation result*: fail*

This solution was obtained from the former one by applying also the ARRAY_PARTITION directive to all the variables involved in the code, with a block factor of 11 on each dimension (this was a brute-force approach that basically decomposed each element of the array into a single value assigned to a register). The reasoning behind this has been the following: since Vitis notified that there weren't enough memory ports to manage the data during the computation given the optimizations applied, the placement of the data into register structures should have led to an improvement.



Directives implemented in this solution.

Here the results of this implementation:

Synthesis Details Report for 'kernel'

General Information

Date:	Tue Jun 22 23:45:59 2021
Version:	2020.3 (Build 3173277 on Wed Apr 7 05:07:49 MDT
Project:	ProjectDSDM2
Solution:	solution_pipelined_partition_all (Vivado IP Flow Target)
Product family:	versal
Target device:	xcvm1802-vsvd1760-3HP-e-S

Performance Estimates

Timing

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00 ns	7.296 ns	2.70 ns

Latency

Summary

Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
50973	50973	0.510 ms	0.510 ms	50974	50974	none

Utilization Estimates

Summary

Name	BRAM_18K	DSP	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	-	-	-
FIFO	-	-	-	-	-
Instance	-	85	101034	241127	-
Memory	0	-	15488	7986	0
Multiplexer	-	-	-	2	-
Register	-	-	3	-	-
Total	0	85	116525	249115	0
Available	1934	1968	1799680	899840	463
Utilization (%)	0	4	6	27	0

Performance & Resource Estimates

Modules Loops

Modules && Loops	Issue Type	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
kernel		-	50973	5,100E5	-	50974	-	no	0	85	116525	249115	0
kernel_body_1	II Violation	-	50972	5,100E5	-	50972	-	no	0	85	101034	241127	0
gemm_12_13_1	II Violation	-	8030	8,030E4	-	8030	-	no	0	33	4996	12080	0
GEMM_LOOP1_GEMM_LOOP2	II Violation	-	8028	8,028E4	49	6	1331	yes	-	-	-	-	-
tensor_transpose		-	138	1,380E3	-	138	-	no	0	0	86054	34521	0
TTRANS_LOOP1_TTRANS_LOOP2		-	130	1,300E3	11	1	121	yes	-	-	-	-	-
gemm_clone	II Violation	-	8030	8,030E4	-	8030	-	no	0	33	4979	13576	0
GEMM_LOOP1_GEMM_LOOP2	II Violation	-	8028	8,028E4	49	6	1331	yes	-	-	-	-	-
RES_LOOP1_RES_LOOP2		-	133	1,330E3	14	1	121	yes	-	-	-	-	-
MTRANS_LOOP1		-	11	110.000	1	1	11	yes	-	-	-	-	-
MULT_LOOP1_MULT_LOOP2	II Violation	-	1331	1,331E4	11	11	121	yes	-	-	-	-	-
TRANSRES_LOOP1_TRANSRES_LOOP2		-	739	7,390E3	20	6	121	yes	-	-	-	-	-

As we can see from the report, there has been indeed an improvement of almost 7000 cycles, paired obviously with an increase of the resources exploited. The synthesis of this kind of implementation took a huge amount of time, probably because the tool had to build a large multiplexer to manage the access to each single data, and the results aren't as good as I was expecting. Moreover, this solution failed the C Co-Simulation. I decided to include it anyway in order to provide feedback on the work done, but I have noticed that a lot of solutions that used Partitioning ended up failing Co-Simulation. The debugging of this issue has been tough and even after some work I couldn't figure out what was going wrong, so I decided to try other approaches, such as unrolling loops.

Solution 4: Loop Unrolling

Source file: *src/vanilla/kernel_gemm.c*
User defined directives*: unrolling of GEMM_LOOP3 and TTRANS_LOOP2*
Latency: *18098 cycles*
Co-simulation result*: pass*

By analyzing the code and the report given by Vitis, it can be seen that the loop executed by the functions “gemm” and “tensor_transpose” are executed multiple times in the component, contributing for a large part in the overall count of clock cyles. I decided to start applying unrolling only to those two loops in order to check if the improvement was greater than before, and eventually progressing with the unrolling of every loop. Here the results obtained:

Synthesis Details Report for 'kernel'

General Information

Date:	Sat Jul 3 00:42:04 2021
Version:	2020.3 (Build 3173277 on Wed Apr 7 05:07:49 MDT
Project:	ProjectDSDM2
Solution:	solution_unroll (Vivado IP Flow Target)
Product family:	versal
Target device:	xcvm1802-vsvd1760-3HP-e-S

Performance Estimates

Timing

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00 ns	7.296 ns	2.70 ns

Latency

Summary

Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
18098	18098	0.181 ms	0.181 ms	18099	18099	none

Utilization Estimates

Summary

Name	BRAM_18K	DSP	FF	LUT	URAM
DSP	-	8	-	-	-
Expression	-	-	0	361	-
FIFO	-	-	-	-	-
Instance	-	514	246608	191282	-
Memory	2	-	0	0	2
Multiplexer	-	-	-	498	-
Register	-	-	452	0	-
Total	2	522	247060	192141	2
Available	1934	1968	1799680	899840	463
Utilization (%)	~0	26	13	21	~0

Cosimulation Report for 'kernel'

General Information

Date:	Sun Jul 11 12:34:59 2021	Solution:	solution_unroll (Vivado IP Flow
Version:	2020.3 (Build 3173277 on Wed Apr 7 05:07:49 MDT	Product	versal
Project:	ProjectDSDM2	Target device:	xcvm1802-vsvd1760-3HP-e-S
Status:	Pass		

Cosim Options

Tool: Vivado XSIM RTL: Verilog

Performance Estimates

Modules	Avg II	Max II	Min II	Avg Latency	Max Latency	Min Latency
kernel				18098	18098	18098
gemm_1	2790	2790	2790	1400	1400	1400
tensor_transpose	3745	5655	2790	1386	1386	1386
gemm	2790	2790	2790	1400	1400	1400

As the metrics of Vitis show, the unrolling of the loops allowed to obtain the first significant improvement in the performance of the accelerator, by saving 39.509 clock cycles. This is an increase in performance of **68,5%** with respect to the trivial solution. The overall resource exploitation increased; however, the DSP consumption had an increase of just +22% and the flip flops of +7%, and the LUT utilization even decreased from 27% to 21%.

Here is the metric of each loop:

Performance & Resource Estimates

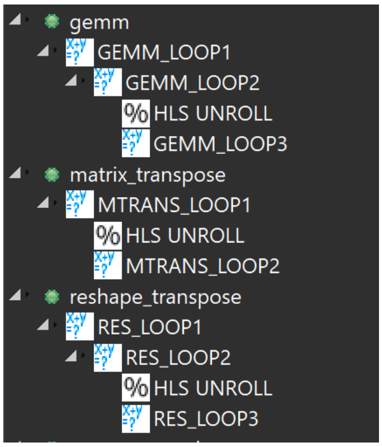
Modules && Loops	Issue Type	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	B
gemm_1	II Violation	-	1400	1,400E4	-	1400	-	no	
GEMM_LOOP1	II Violation	-	732	7,320E3	122	61	11	yes	
tensor_transpose		-	1386	1,386E4	-	1386	-	no	
TTRANS_LOOP2_TTRANS_LOOP3		-	124	1,240E3	5	1	121	yes	
TTRANS_LOOP2_TTRANS_LOOP3		-	124	1,240E3	5	1	121	yes	
TTRANS_LOOP2_TTRANS_LOOP3		-	124	1,240E3	5	1	121	yes	
TTRANS_LOOP2_TTRANS_LOOP3		-	124	1,240E3	5	1	121	yes	
TTRANS_LOOP2_TTRANS_LOOP3		-	124	1,240E3	5	1	121	yes	
TTRANS_LOOP2_TTRANS_LOOP3		-	124	1,240E3	5	1	121	yes	
TTRANS_LOOP2_TTRANS_LOOP3		-	124	1,240E3	5	1	121	yes	
TTRANS_LOOP2_TTRANS_LOOP3		-	124	1,240E3	5	1	121	yes	
TTRANS_LOOP2_TTRANS_LOOP3		-	124	1,240E3	5	1	121	yes	
TTRANS_LOOP2_TTRANS_LOOP3		-	124	1,240E3	5	1	121	yes	
TTRANS_LOOP2_TTRANS_LOOP3		-	124	1,240E3	5	1	121	yes	
gemm	II Violation	-	1400	1,400E4	-	1400	-	no	
GEMM_LOOP1	II Violation	-	732	7,320E3	122	61	11	yes	
RES_LOOP1_RES_LOOP2_RES_LOOP3		-	1334	1,334E4	5	1	1331	yes	
MTRANS_LOOP1_MTRANS_LOOP2		-	124	1,240E3	5	1	121	yes	
MULT_LOOP1_MULT_LOOP2_MULT_LOOP3		-	1336	1,336E4	7	1	1331	yes	
TRANSRES_LOOP1_TRANSRES_LOOP2_TRANSRES_LOOP3		-	1334	1,334E4	5	1	1331	yes	

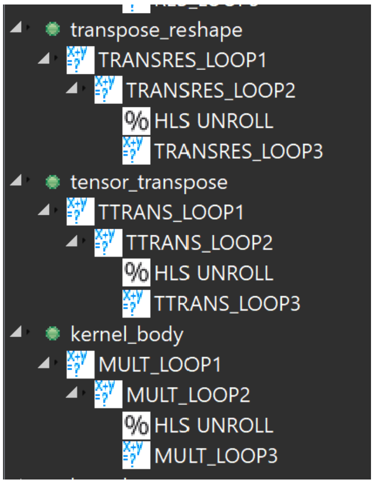
The next solution naturally followed from the results of this one: a full unrolling of each loop.

Solution 5: Every Loop Unrolling

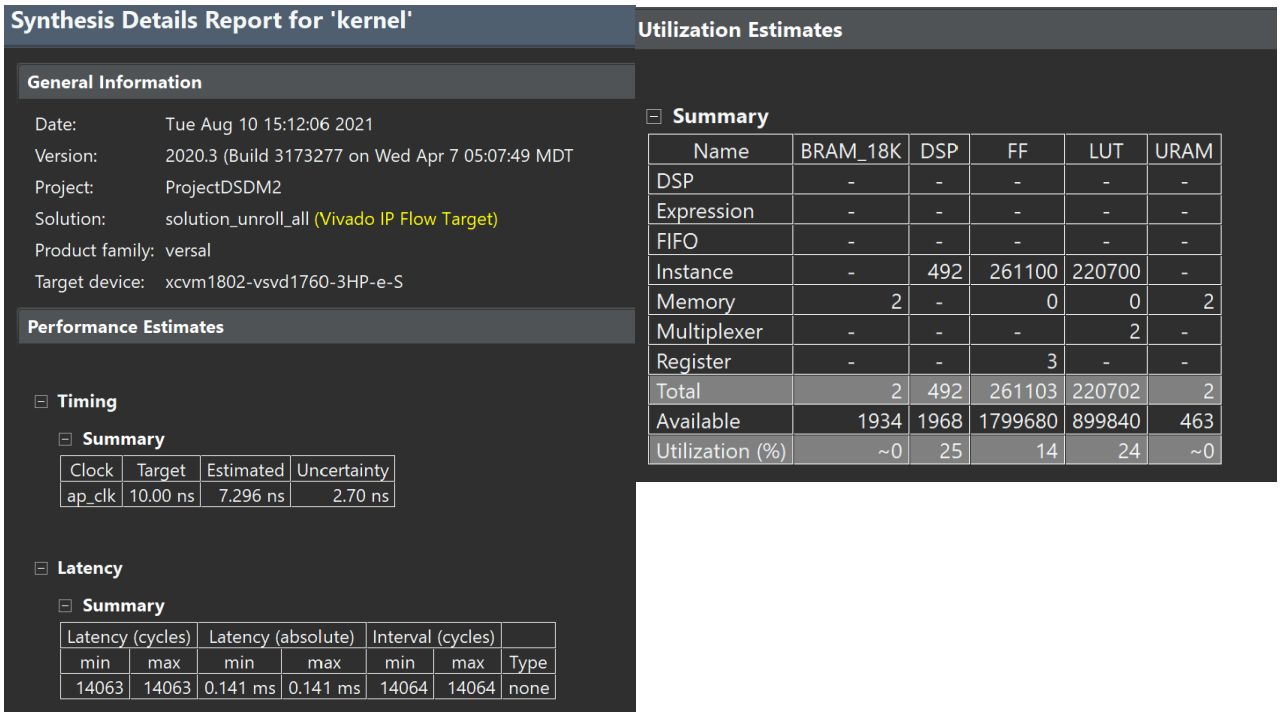
Source file: *src/vanilla/kernel_gemm.c*
User defined directives*: unrolling of every loop executed*
Latency: *14063 cycles*
Co-simulation result*: pass*

Very similar to the previous one but with slightly better performances. Here the optimizations applied:





And these are the results obtained:



Synthesis Details Report for 'kernel'

General Information

Date: Tue Aug 10 15:12:06 2021
Version: 2020.3 (Build 3173277 on Wed Apr 7 05:07:49 MDT
Project: ProjectDSDM2
Solution: solution_unroll_all (Vivado IP Flow Target)
Product family: versal
Target device: xcvm1802-vsvd1760-3HP-e-S

Performance Estimates

Timing

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00 ns	7.296 ns	2.70 ns

Latency

Summary

Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
14063	14063	0.141 ms	0.141 ms	14064	14064	none

Utilization Estimates

Summary

Name	BRAM_18K	DSP	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	-	-	-
FIFO	-	-	-	-	-
Instance	-	492	261100	220700	-
Memory	2	-	0	0	2
Multiplexer	-	-	-	2	-
Register	-	-	3	-	-
Total	2	492	261103	220702	2
Available	1934	1968	1799680	899840	463
Utilization (%)	~0	25	14	24	~0

Cosimulation Report for 'kernel'

General Information

Date: Tue Aug 10 15:24:18 2021
Version: 2020.3 (Build 3173277 on Wed Apr 7 05:07:49 MDT
Project: ProjectDSDM2
Status: Pass

Solution: solution_unroll_all (Vivado IP Flow
Product versal
Target device: xcvm1802-vsvd1760-3HP-e-S

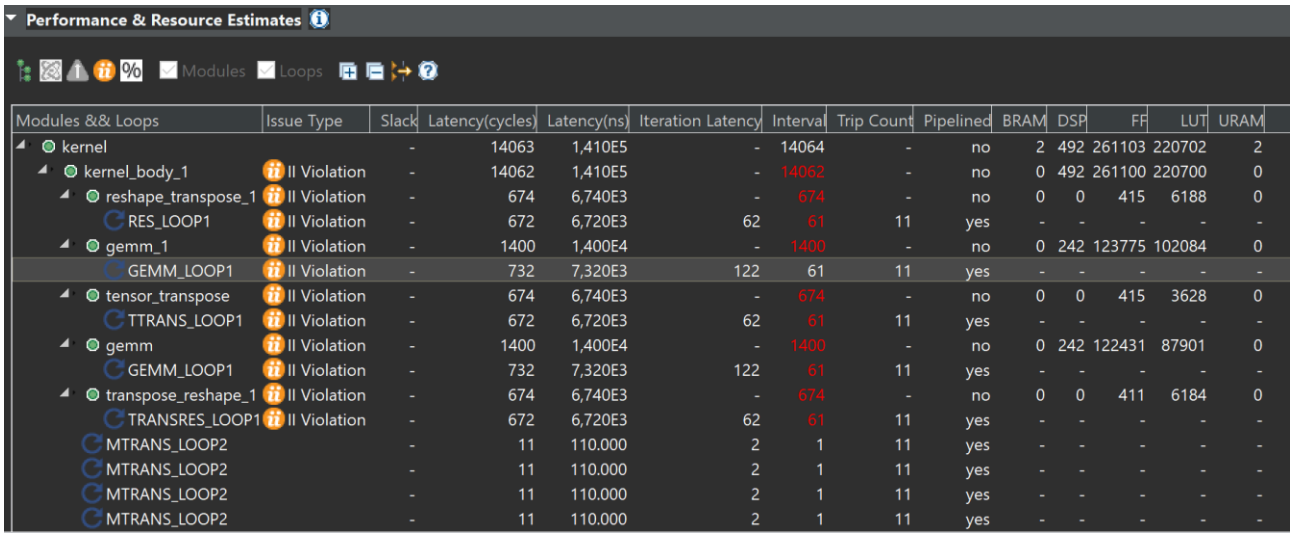
Cosim Options

Tool: Vivado XSIM
RTL: Verilog

Performance Estimates

Modules	Avg II	Max II	Min II	Avg Latency	Max Latency	Min Latency
kernel				14063	14063	14063
kernel_body_1				14062	14062	14062

This time the improvement was of more than 4000 cycles on the previous one value. This is far less than the amount of the previous solution but still a better result, especially if compared to the trivial approach. The utilization increased but each resource is occupied for less than 25%. It is important to notice that Vitis is still trying to automatically apply pipelining where possible:



Performance & Resource Estimates

Modules Loops

Modules && Loops	Issue Type	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
kernel		-	14063	1,410E5	-	14064	-	no	2	492	261103	220702	2
kernel_body_1	II Violation	-	14062	1,410E5	-	14062	-	no	0	492	261100	220700	0
reshape_transpose_1	II Violation	-	674	6,740E3	-	674	-	no	0	0	415	6188	0
RES_LOOP1	II Violation	-	672	6,720E3	62	61	11	yes	-	-	-	-	-
gemm_1	II Violation	-	1400	1,400E4	-	1400	-	no	0	242	123775	102084	0
GEMM_LOOP1	II Violation	-	732	7,320E3	122	61	11	yes	-	-	-	-	-
tensor_transpose	II Violation	-	674	6,740E3	-	674	-	no	0	0	415	3628	0
TTRANS_LOOP1	II Violation	-	672	6,720E3	62	61	11	yes	-	-	-	-	-
gemm	II Violation	-	1400	1,400E4	-	1400	-	no	0	242	122431	87901	0
GEMM_LOOP1	II Violation	-	732	7,320E3	122	61	11	yes	-	-	-	-	-
transpose_reshape_1	II Violation	-	674	6,740E3	-	674	-	no	0	0	411	6184	0
TRANSRES_LOOP1	II Violation	-	672	6,720E3	62	61	11	yes	-	-	-	-	-
MTRANS_LOOP2		-	11	110.000	2	1	11	yes	-	-	-	-	-
MTRANS_LOOP2		-	11	110.000	2	1	11	yes	-	-	-	-	-
MTRANS_LOOP2		-	11	110.000	2	1	11	yes	-	-	-	-	-
MTRANS_LOOP2		-	11	110.000	2	1	11	yes	-	-	-	-	-

Following this solution, several approaches were tried in order to obtain improvements. I tried to apply another time the partitioning, but even if applied only to one or two variables, the code **always failed co-simulation.** This issue remained during the whole process and in the end no solution was able to work if the partitioning was enabled.

After various attempts, no further directives (or combination of them) brought improvements to the project. I decided that it was time to start working on the source code, keeping the loops unrolled, in order to find a version of it that could perform better. The unsuccessful solutions are briefly reported, while the working ones are described in detail.

Solution 6: Gemm Modules Duplication

This solution will be briefly described just for keeping track of the work done. I tried to duplicate the various “gemm” functions by making copies of them and calling one and only one of them each time in the kernel function. I thought that this approach could force Vitis to implement a single module for each function and speed up the computation, but this didn’t bring any improvement.

Solution 7: Removing Ping-Pong Buffer

The kernel function uses two arrays, t1 and t2, as a ping pong buffer: the output of a computation is saved in one of them, in order to be used as an input on the next step and vice-versa. This solution tried to use dedicated arrays for each operation in order to have arrays always used as input and others always dedicated to the output. The solution however didn’t present improvements or different behaviours from the previous ones, so it is not reported here in detail. I will just mention how the different arrays have been implemented:

Original code:

```
 74    reshape_transpose(u, t1);
 75
 76    gemm(X, Y, Z, X*Y, S, t1, t2);
 77    tensor_transpose(t2, t1);
```

Edited code:

```
 74    reshape_transpose(u, t1);
 75    gemm(X, Y, Z, X*Y, S, t1, t2);
 76    double t3[Z][X*Y];
 77    tensor_transpose(t2, t3);
 78    double t4[Z][X*Y];
 79    gemm(X, Y, Z, X*Y, S, t3, t4);
and so on…
```

Solution 8: Transpose Deleted

Source file: *src/deleted_transpose/kernel_gemm.c*
User defined directives*: unrolling of every loop executed*
Latency: *11359 cycles*
Co-simulation result*: pass*

One of the best solutions has been found thanks to a modification of the original source code. This variation isn't actually hardware related, since it does not expose more parallelism, but is a kind of programming trick that allows to save some clock cycles. Analyzing the behavior of the code, it was noticed that some cycles were lost to carry out the transposition of the matrix by executing the function "tensor_transpose". In particular, the transposition happened always after an execution of the "gemm" function on the same matrix. To speed up the operation, I decided to condense the two functions "tensor_transpose" and "gemm" into a single "gemm_transpose" function, which performs the multiplication on the matrix without transposing it but using instead the right sequence of values. In practice, instead of transposing the matrix, it is left as it is but read as if it were transposed. To be clearer, I will briefly report here the modification applied to the source code.

Each occurrence of the following sequence of function calls:

```
gemm(X, Y, Z, X*Y, S, t1, t2);
tensor_transpose(t2, t1);
```

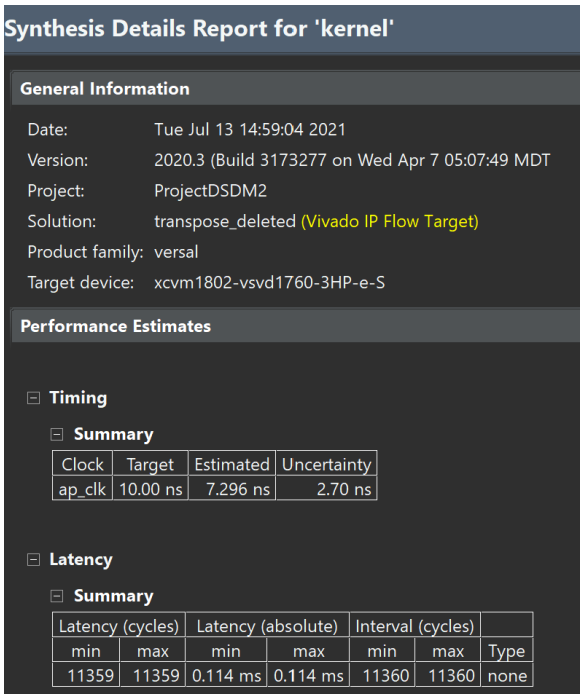
Has been substituted with the single function call:

```
gemm_transpose(X, Y, Z, X*Y, S, t2, t1);
```

And as it can be seen from the code, the latter function performs the matrix multiplication by reading the values of in2 as if it were transposed:

```
34  void gemm_transpose(
35    unsigned m1,
36    unsigned n1,
37    unsigned m2,
38    unsigned n2,
39    const double in1[m1][n1],
40    const double in2[m2][n2],
41    double out[m1][n2]
42  )
43  {
44
45      int h = 0;
46      int l = 0;
47
48    GEMMTR_LOOP1: for (unsigned i = 0; i < m1; i++){
49        h=0;
50      GEMMTR_LOOP2: for (unsigned j = 0; j < n2; j++) {
51        out[i][j] = 0;
52        GEMMTR_LOOP3: for (unsigned k = 0; k < m2; k++) {
53          out[i][j] += in1[i][k] * in2[h][l];
54          l++;
55        }
56      }
57      h++;
58      l=0;
59    }
60  }
```

This modification allowed to skip the call of the "tensor_transpose" function, allowing us to remove it from the code. The result can be seen from the report of the synthesis tool:



Synthesis Details Report for 'kernel'

General Information

Date: Tue Jul 13 14:59:04 2021
Version: 2020.3 (Build 3173277 on Wed Apr 7 05:07:49 MDT
Project: ProjectDSDM2
Solution: transpose_deleted (Vivado IP Flow Target)
Product family: versal
Target device: xcvm1802-vsvd1760-3HP-e-S

Performance Estimates

Timing

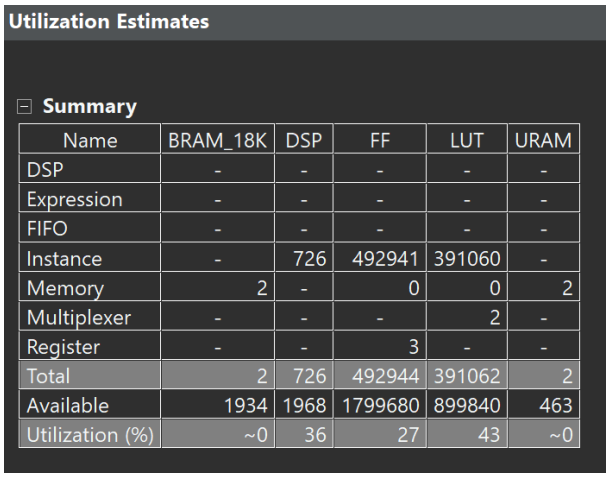
Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00 ns	7.296 ns	2.70 ns

Latency

Summary

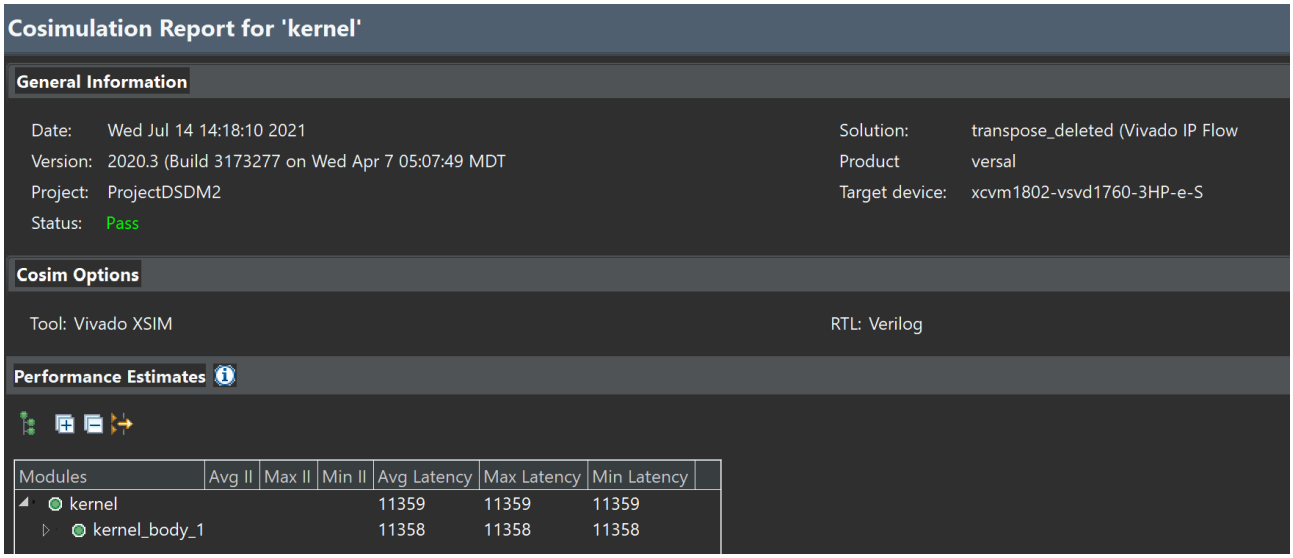
Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
11359	11359	0.114 ms	0.114 ms	11360	11360	none



Utilization Estimates

Summary

Name	BRAM_18K	DSP	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	-	-	-
FIFO	-	-	-	-	-
Instance	-	726	492941	391060	-
Memory	2	-	0	0	2
Multiplexer	-	-	-	2	-
Register	-	-	3	-	-
Total	2	726	492944	391062	2
Available	1934	1968	1799680	899840	463
Utilization (%)	~0	36	27	43	~0



Cosimulation Report for 'kernel'

General Information

Date: Wed Jul 14 14:18:10 2021
Version: 2020.3 (Build 3173277 on Wed Apr 7 05:07:49 MDT
Project: ProjectDSDM2
Status: Pass

Solution: transpose_deleted (Vivado IP Flow
Product versal
Target device: xcvm1802-vsvd1760-3HP-e-S

Cosim Options

Tool: Vivado XSIM
RTL: Verilog

Performance Estimates

Modules	Avg II	Max II	Min II	Avg Latency	Max Latency	Min Latency
kernel				11359	11359	11359
kernel_body_1				11358	11358	11358

This version improved the former one of almost 3000 cycles, reaching a performance of 11359 cycles, which is an improvement of **80,3%** of the original performance. As the testing results and the Co-Simulation report show, the code still behaves as intended and the results are the same. On the next page is reported the complete performance estimate of each loop. It is worth to remember that the same directives of the previous working solution (“Every Loop Unrolling”) are applied here, so there is a combination of automatic pipelining and unrolling on each loop. Some II Violations are still present, but the results are much better than the first version.

Performance & Resource Estimates

Modules && Loops	Issue Type	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
kernel		-	11359	1,140E5	-	11360	-	no	2	726	492944	391062	2
kernel_body_1	II Violation	-	11358	1,140E5	-	11358	-	no	0	726	492941	391060	0
reshape_transpose_1	II Violation	-	674	6,740E3	-	674	-	no	0	0	415	6188	0
RES_LOOP1	II Violation	-	672	6,720E3	62	61	11	yes	-	-	-	-	-
gemm_1	II Violation	-	1400	1,400E4	-	1400	-	no	0	0	110025	79270	0
GEMM_LOOP1	II Violation	-	732	7,320E3	122	61	11	yes	-	-	-	-	-
gemm_transpose_1	II Violation	-	1400	1,400E4	-	1400	-	no	0	242	123775	102084	0
GEMM_LOOP1	II Violation	-	732	7,320E3	122	61	11	yes	-	-	-	-	-
gemm	II Violation	-	1400	1,400E4	-	1400	-	no	0	0	108681	65087	0
GEMM_LOOP1	II Violation	-	732	7,320E3	122	61	11	yes	-	-	-	-	-
gemm_transpose	II Violation	-	1400	1,400E4	-	1400	-	no	0	242	122431	87901	0
GEMM_LOOP1	II Violation	-	732	7,320E3	122	61	11	yes	-	-	-	-	-
transpose_reshape_1	II Violation	-	674	6,740E3	-	674	-	no	0	0	411	6184	0
TRANSRES_LOOP1	II Violation	-	672	6,720E3	62	61	11	yes	-	-	-	-	-
MTRANS_LOOP2		-	11	110.000	2	1	11	yes	-	-	-	-	-
MTRANS_LOOP2		-	11	110.000	2	1	11	yes	-	-	-	-	-
MTRANS_LOOP2		-	11	110.000	2	1	11	yes	-	-	-	-	-
MTRANS_LOOP2		-	11	110.000	2	1	11	yes	-	-	-	-	-
MTRANS_LOOP2		-	11	110.000	2	1	11	yes	-	-	-	-	-
MTRANS_LOOP2		-	11	110.000	2	1	11	yes	-	-	-	-	-
MTRANS_LOOP2		-	11	110.000	2	1	11	yes	-	-	-	-	-
MTRANS_LOOP2		-	11	110.000	2	1	11	yes	-	-	-	-	-
MTRANS_LOOP2		-	11	110.000	2	1	11	yes	-	-	-	-	-
MTRANS_LOOP2		-	11	110.000	2	1	11	yes	-	-	-	-	-
MTRANS_LOOP2		-	11	110.000	2	1	11	yes	-	-	-	-	-
MULT_LOOP1	II Violation	-	1451	1,451E4	242	121	11	yes	-	-	-	-	-

Quite satisfied with this result, I decided to try to solve the II Violations to make pipelining work better. So, I tried again to partition the variables. This can be seen on the next solution.

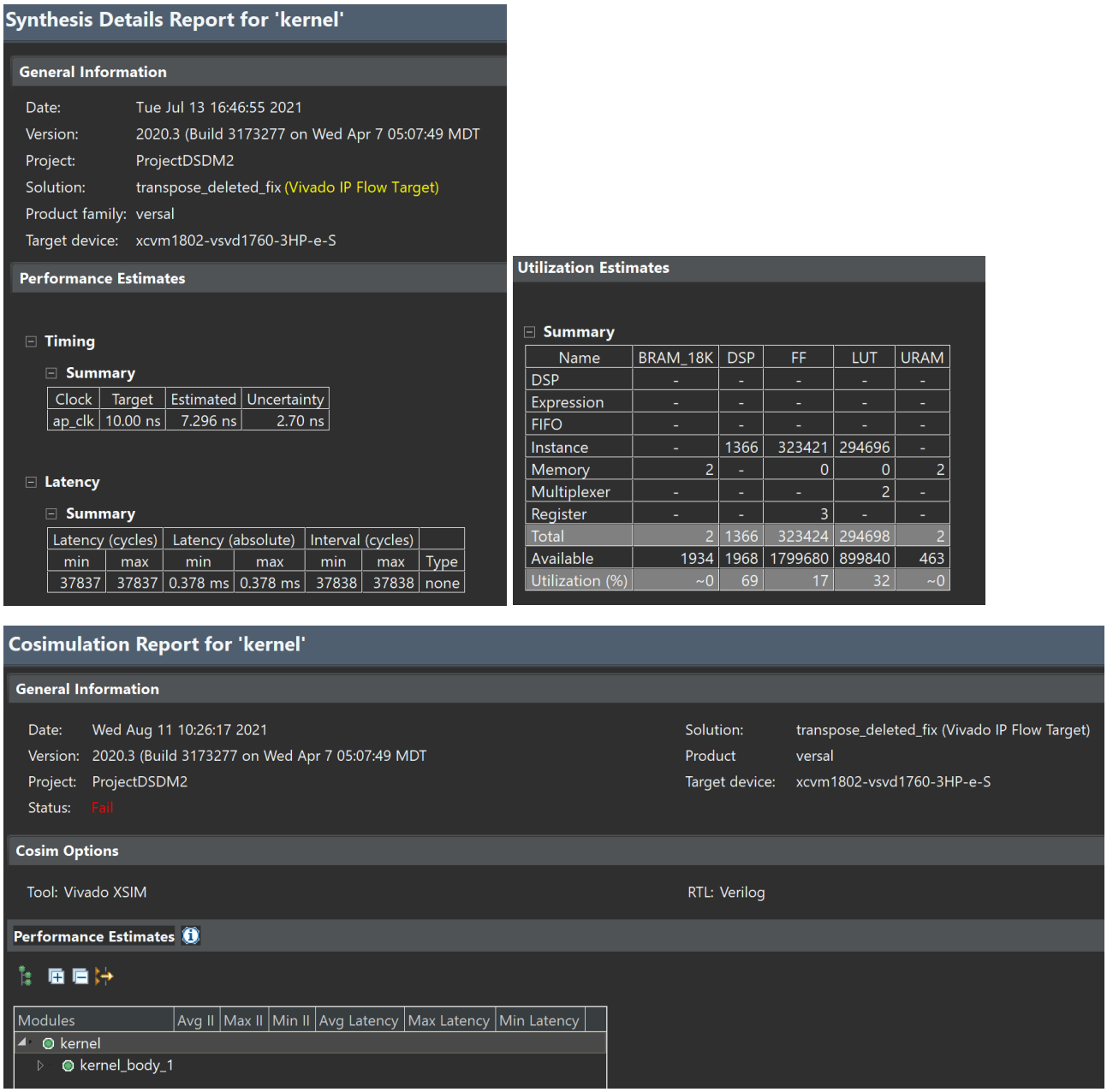
Solution 9: Transpose Deleted + Double Partition

Source file: *src/deleted_transpose/kernel_gemm.c*
User defined directives*: unrolling of every loop executed, complete partitioning on S and r*
Latency: *37837 cycles*
Co-simulation result*: fail*

In order to fix the II Violations given by the previous solution, I tried to apply partitioning on S and r with the "complete" option on every dimension. For clarity, here the directives related to the partitioning:

```
kernel
    HLS TOP name=kernel
    S
    HLS ARRAY_PARTITION variable=S complete dim=0
    D
    u
    r
    HLS ARRAY_PARTITION variable=r complete dim=0
    S_T
    t1
    t2
```

I was confident that, despite the usual long and boring synthesis time, this solution would have brought a considerable improvement on the performance of the component. As the report of Vitis shows, I was wrong.



Synthesis Details Report for 'kernel'

General Information

Date: Tue Jul 13 16:46:55 2021
Version: 2020.3 (Build 3173277 on Wed Apr 7 05:07:49 MDT
Project: ProjectDSDM2
Solution: transpose_deleted_fix (Vivado IP Flow Target)
Product family: versal
Target device: xcvm1802-vsvd1760-3HP-e-S

Performance Estimates

Timing

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00 ns	7.296 ns	2.70 ns

Latency

Summary

Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
37837	37837	0.378 ms	0.378 ms	37838	37838	none

Utilization Estimates

Summary

Name	BRAM_18K	DSP	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	-	-	-
FIFO	-	-	-	-	-
Instance	-	1366	323421	294696	-
Memory	2	-	0	0	2
Multiplexer	-	-	-	2	-
Register	-	-	3	-	-
Total	2	1366	323424	294698	2
Available	1934	1968	1799680	899840	463
Utilization (%)	~0	69	17	32	~0

Cosimulation Report for 'kernel'

General Information

Date: Wed Aug 11 10:26:17 2021
Version: 2020.3 (Build 3173277 on Wed Apr 7 05:07:49 MDT
Project: ProjectDSDM2
Status: Fail

Solution: transpose_deleted_fix (Vivado IP Flow Target)
Product: versal
Target device: xcvm1802-vsvd1760-3HP-e-S

Cosim Options

Tool: Vivado XSIM
RTL: Verilog

Performance Estimates

Modules	Avg II	Max II	Min II	Avg Latency	Max Latency	Min Latency
kernel						
kernel_body_1						

The first thing to be noticed is that the number of clock cycles considerably increased in this solution: more than 26.000 cycles with respect to the previous solution. The DSP utilization also increased, but the remarkable thing is that Co-Simulation failed.

Performance & Resource Estimates

Modules && Loops	Issue Type	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
kernel		-	37837	3,780E5	-	37838	-	no	2	1366	323424	294698	2
kernel_body_1	II Violation	-	37836	3,780E5	-	37836	-	no	0	1366	323421	294696	0
reshape_transpose_1	II Violation	-	674	6,740E3	-	674	-	no	0	0	415	6188	0
RES_LOOP1	II Violation	-	672	6,720E3	62	61	11	yes	-	-	-	-	-
gemm_1		-	1398	1,398E4	-	1398	-	no	0	0	121045	63260	0
GEMM_LOOP1		-	730	7,300E3	120	61	11	yes	-	-	-	-	-
gemm_transpose_1	II Violation	-	8020	8,020E4	-	8020	-	no	0	23	2275	5604	0
GEMMTR_LOOP1_GEMMTR_LOOP2	II Violation	-	8018	8,018E4	39	6	1331	yes	-	-	-	-	-
gemm_clone	II Violation	-	1400	1,400E4	-	1400	-	no	0	0	108681	65087	0
GEMM_LOOP1	II Violation	-	732	7,320E3	122	61	11	yes	-	-	-	-	-
gemm_transpose_clone	II Violation	-	8020	8,020E4	-	8020	-	no	0	23	2253	3296	0
GEMMTR_LOOP1_GEMMTR_LOOP2	II Violation	-	8018	8,018E4	39	6	1331	yes	-	-	-	-	-
transpose_reshape_1	II Violation	-	674	6,740E3	-	674	-	no	0	0	409	1692	0
MTRANS_LOOP2		-	11	110.000	1	1	11	yes	-	-	-	-	-
MTRANS_LOOP2		-	11	110.000	1	1	11	yes	-	-	-	-	-
MTRANS_LOOP2		-	11	110.000	1	1	11	yes	-	-	-	-	-
MTRANS_LOOP2		-	11	110.000	1	1	11	yes	-	-	-	-	-
MTRANS_LOOP2		-	11	110.000	1	1	11	yes	-	-	-	-	-
MTRANS_LOOP2		-	11	110.000	1	1	11	yes	-	-	-	-	-
MTRANS_LOOP2		-	11	110.000	1	1	11	yes	-	-	-	-	-
MTRANS_LOOP2		-	11	110.000	1	1	11	yes	-	-	-	-	-
MTRANS_LOOP2		-	11	110.000	1	1	11	yes	-	-	-	-	-
MTRANS_LOOP2		-	11	110.000	1	1	11	yes	-	-	-	-	-
MTRANS_LOOP2		-	11	110.000	1	1	11	yes	-	-	-	-	-
MULT_LOOP1	II Violation	-	1451	1,451E4	242	121	11	yes	-	-	-	-	-

At this point I decided that, despite the results, I would have kept trying with partitioning. The next solution shows some promising but not satisfying results.

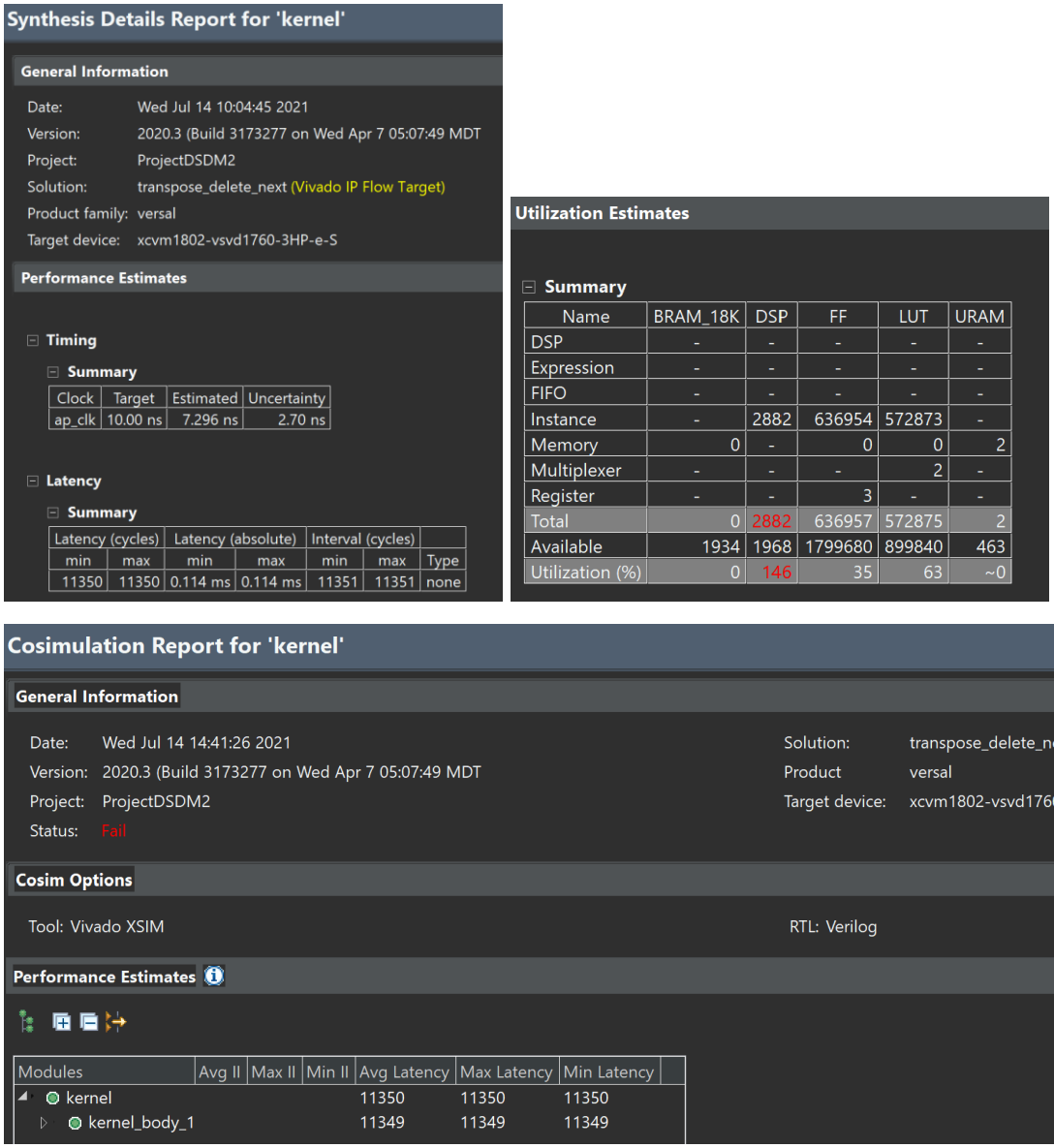
Solution 10: Transpose Deleted + S Partition

Source file: *src/deleted_transpose/kernel_gemm.c*
User defined directives: *unrolling of every loop executed, complete partitioning on S and S_T*
Latency: *11350 cycles*
Co-simulation result: *fail*

The idea behind this solution was to apply partitioning only on the smaller matrix S (and the S_T variable used to store it), being it an 11x11 matrix, smaller than the 11x11x11 tensors. Here the directives applied:

```
kernel
    HLS TOP name=kernel
    S
    HLS ARRAY_PARTITION variable=S complete dim=0
    D
    u
    r
    S_T
    HLS ARRAY_PARTITION variable=S_T complete dim=0
    t1
    t2
```

At first, I was surprised that this result actually obtained the best cycles estimation of all the solutions tried so far, even if it was an improvement of only 9 cycles with respect to the best solution without partitioning:



Synthesis Details Report for 'kernel'

General Information

Date: Wed Jul 14 10:04:45 2021
Version: 2020.3 (Build 3173277 on Wed Apr 7 05:07:49 MDT
Project: ProjectDSDM2
Solution: transpose_delete_next (Vivado IP Flow Target)
Product family: versal
Target device: xcvm1802-vsvd1760-3HP-e-S

Performance Estimates

Timing

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00 ns	7.296 ns	2.70 ns

Latency

Summary

Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
11350	11350	0.114 ms	0.114 ms	11351	11351	none

Utilization Estimates

Summary

Name	BRAM_18K	DSP	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	-	-	-
FIFO	-	-	-	-	-
Instance	-	2882	636954	572873	-
Memory	0	-	0	0	2
Multiplexer	-	-	-	2	-
Register	-	-	3	-	-
Total	0	2882	636957	572875	2
Available	1934	1968	1799680	899840	463
Utilization (%)	0	146	35	63	~0

Cosimulation Report for 'kernel'

General Information

Date: Wed Jul 14 14:41:26 2021
Version: 2020.3 (Build 3173277 on Wed Apr 7 05:07:49 MDT
Project: ProjectDSDM2
Status: Fail

Solution: transpose_delete_next (Vivado IP Flow
Product versal
Target device: xcvm1802-vsvd1760-3HP-e-S

Cosim Options

Tool: Vivado XSIM

RTL: Verilog

Performance Estimates

Modules	Avg II	Max II	Min II	Avg Latency	Max Latency	Min Latency
kernel				11350	11350	11350
kernel_body_1				11349	11349	11349

Inspecting the reports given by the tool we can however find some problems: the DSP utilization is 146%, and the Co-Simulation presents a failure. Since the improvement on the speed of the component is neglectable, I decided to stop trying to apply partitioning to this version of the code, since the utilization of this directive brought more disadvantages than improvements and the debugging of Co-Simulation results was not possible. I decided to revert the changes and try other modifications to be applied to the code.

Solution 11: Mult Extracted

Source file: *src/mult_extracted/kernel_gemm.c*
User defined directives*: unrolling of every loop executed*
Latency: *11361 cycles*
Co-simulation result*: pass*

The rationale behind this solution was trying to extract the multiplication between t2 and D performed in the “kernel_body” function and substituting it with a function call to a dedicated function. This is the definition of the function:

```
87  void mult_operation(double out [Z][X*Y], const double in[X][Y][Z]){
88
89      double tmp = 0.0;
90
91        MULT_LOOP1: for (unsigned i = 0; i < X; ++i)
92          MULT_LOOP2: for (unsigned j = 0; j < Y; ++j)
93            MULT_LOOP3: for (unsigned k = 0; k < Z; ++k){
94              tmp = out[i][j*Y + k];
95              out[i][j*Y + k] = tmp * in[i][j][k];
96            }
97
98  }
```

This modification led to a 11361 cycles solution, as can be seen from the reports:

Synthesis Details Report for 'kernel'

General Information

Date:	Thu Jul 15 15:18:51 2021
Version:	2020.3 (Build 3173277 on Wed Apr 7 05:07:49 MDT
Project:	ProjectDSDM2
Solution:	mult_extracted (Vivado IP Flow Target)
Product family:	versal
Target device:	xcvm1802-vsvd1760-3HP-e-S

Performance Estimates

Timing

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00 ns	7.296 ns	2.70 ns

Latency

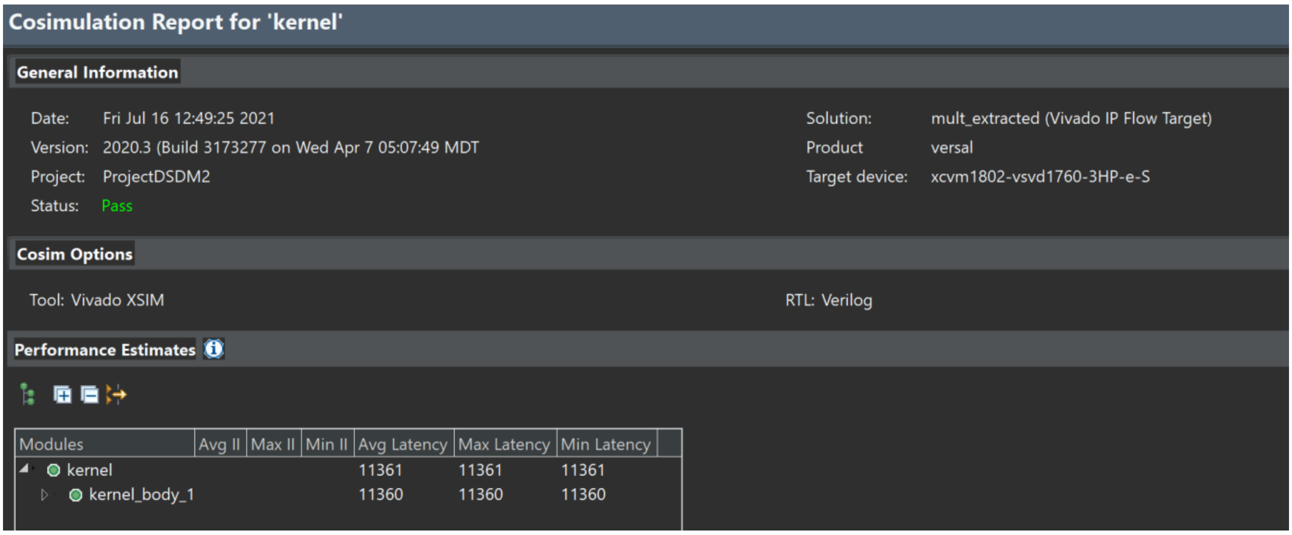
Summary

Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
11361	11361	0.114 ms	0.114 ms	11362	11362	none

Utilization Estimates

Summary

Name	BRAM_18K	DSP	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	-	-	-
FIFO	-	-	-	-	-
Instance	-	726	492945	390185	-
Memory	2	-	0	0	2
Multiplexer	-	-	-	2	-
Register	-	-	3	-	-
Total	2	726	492948	390187	2
Available	1934	1968	1799680	899840	463
Utilization (%)	~0	36	27	43	~0



Cosimulation Report for 'kernel'

General Information

Date: Fri Jul 16 12:49:25 2021
Version: 2020.3 (Build 3173277 on Wed Apr 7 05:07:49 MDT
Project: ProjectDSDM2
Status: Pass
Solution: mult_extracted (Vivado IP Flow Target)
Product versal
Target device: xcvm1802-vsvd1760-3HP-e-S

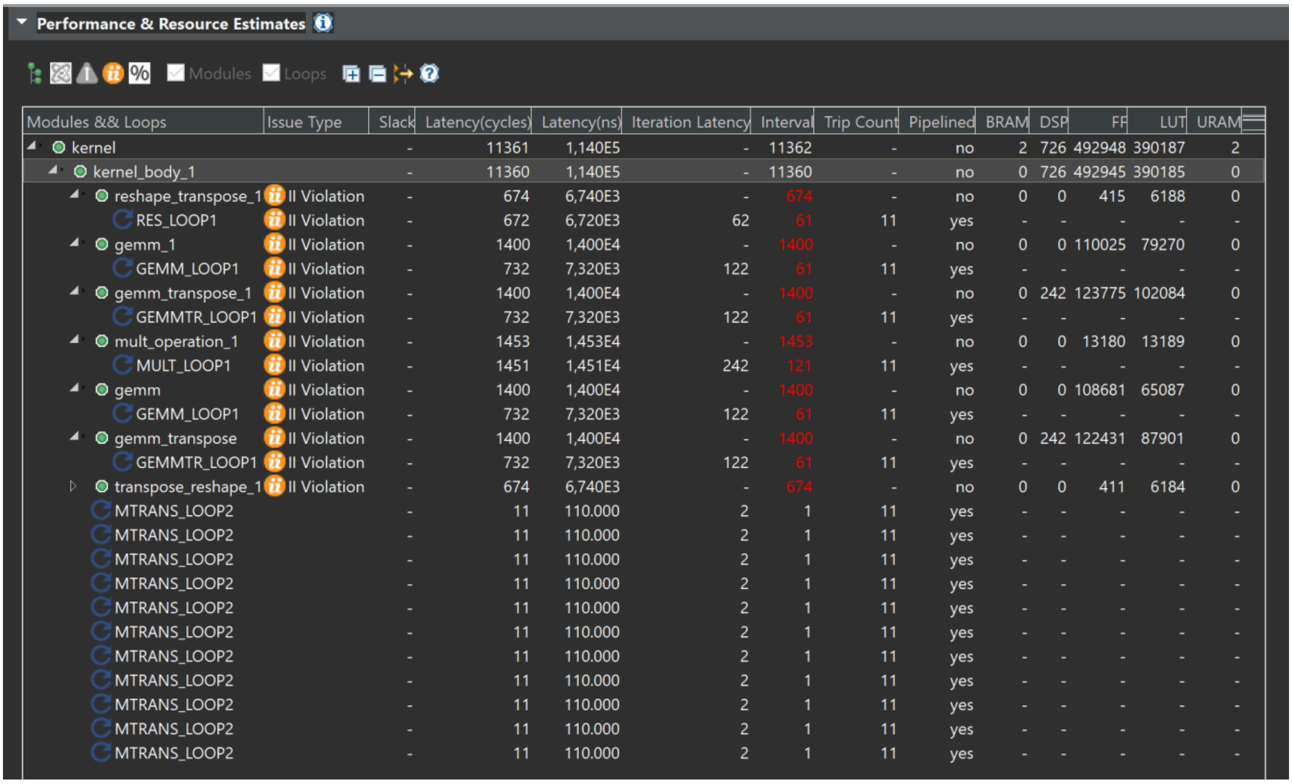
Cosim Options

Tool: Vivado XSIM
RTL: Verilog

Performance Estimates

Modules	Avg II	Max II	Min II	Avg Latency	Max Latency	Min Latency
kernel				11361	11361	11361
kernel_body_1				11360	11360	11360

This result is surely better than the last one, but still could not outperform the results obtained in Solution 8. For completeness, in the next page the details of each loop:



Performance & Resource Estimates

Modules Loops

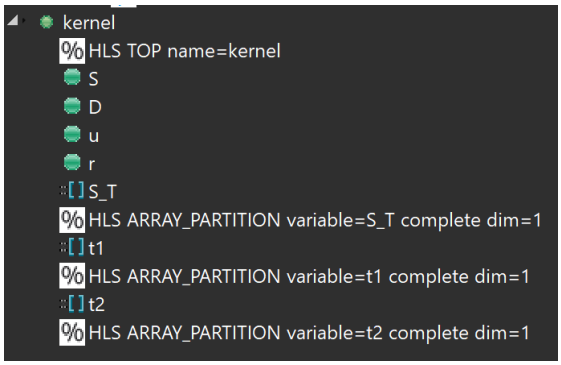
Modules && Loops	Issue Type	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
kernel		-	11361	1,140E5	-	11362	-	no	2	726	492948	390187	2
kernel_body_1		-	11360	1,140E5	-	11360	-	no	0	726	492945	390185	0
reshape_transpose_1	II Violation	-	674	6,740E3	-	674	-	no	0	0	415	6188	0
RES_LOOP1	II Violation	-	672	6,720E3	62	61	11	yes	-	-	-	-	-
gemm_1	II Violation	-	1400	1,400E4	-	1400	-	no	0	0	110025	79270	0
GEMM_LOOP1	II Violation	-	732	7,320E3	122	61	11	yes	-	-	-	-	-
gemm_transpose_1	II Violation	-	1400	1,400E4	-	1400	-	no	0	242	123775	102084	0
GEMMTR_LOOP1	II Violation	-	732	7,320E3	122	61	11	yes	-	-	-	-	-
mult_operation_1	II Violation	-	1453	1,453E4	-	1453	-	no	0	0	13180	13189	0
MULT_LOOP1	II Violation	-	1451	1,451E4	242	121	11	yes	-	-	-	-	-
gemm	II Violation	-	1400	1,400E4	-	1400	-	no	0	0	108681	65087	0
GEMM_LOOP1	II Violation	-	732	7,320E3	122	61	11	yes	-	-	-	-	-
gemm_transpose	II Violation	-	1400	1,400E4	-	1400	-	no	0	242	122431	87901	0
GEMMTR_LOOP1	II Violation	-	732	7,320E3	122	61	11	yes	-	-	-	-	-
transpose_reshape_1	II Violation	-	674	6,740E3	-	674	-	no	0	0	411	6184	0
MTRANS_LOOP2		-	11	110.000	2	1	11	yes	-	-	-	-	-
MTRANS_LOOP2		-	11	110.000	2	1	11	yes	-	-	-	-	-
MTRANS_LOOP2		-	11	110.000	2	1	11	yes	-	-	-	-	-
MTRANS_LOOP2		-	11	110.000	2	1	11	yes	-	-	-	-	-
MTRANS_LOOP2		-	11	110.000	2	1	11	yes	-	-	-	-	-
MTRANS_LOOP2		-	11	110.000	2	1	11	yes	-	-	-	-	-
MTRANS_LOOP2		-	11	110.000	2	1	11	yes	-	-	-	-	-
MTRANS_LOOP2		-	11	110.000	2	1	11	yes	-	-	-	-	-
MTRANS_LOOP2		-	11	110.000	2	1	11	yes	-	-	-	-	-
MTRANS_LOOP2		-	11	110.000	2	1	11	yes	-	-	-	-	-
MTRANS_LOOP2		-	11	110.000	2	1	11	yes	-	-	-	-	-

Once again, I tried to apply partitioning to some variables of this code in order to improve performances.

Solution 12: Mult Extracted Partition

Source file: *src/mult_extracted/kernel_gemm.c*
User defined directives*: unrolling of every loop executed + partitioning on S_T, t1, t2*
Latency: *11241 cycles*
Co-simulation result*: fail*

This solution is the same as number 11 with the partitioning directive applied to S_T, t1 and t2. This time the partitioning has been applied only to the first dimension of the variables and the arrays involved.



The result seems good with respect to the number of clock cycles, with the code obtaining a value of 11241, which makes this the best performing solution so far. Unfortunately, as the following reports show, the solution fails Co-Simulation.

Synthesis Details Report for 'kernel'

General Information

Date:	Fri Jul 16 16:41:57 2021
Version:	2020.3 (Build 3173277 on Wed Apr 7 05:07:49 MDT
Project:	ProjectDSDM2
Solution:	mult_extracted_partition (Vivado IP Flow Target)
Product family:	versal
Target device:	xcvm1802-vsvd1760-3HP-e-S

Performance Estimates

Timing

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00 ns	7.296 ns	2.70 ns

Latency

Summary

Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
11241	11241	0.112 ms	0.112 ms	11242	11242	none

Utilization Estimates

Summary

Name	BRAM_18K	DSP	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	-	-	-
FIFO	-	-	-	-	-
Instance	-	726	493707	408605	-
Memory	44	-	64	33	0
Multiplexer	-	-	-	2	-
Register	-	-	3	-	-
Total	44	726	493774	408640	0
Available	1934	1968	1799680	899840	463
Utilization (%)	2	36	27	45	0

Cosimulation Report for 'kernel'

General Information

Date:	Fri Jul 16 16:59:06 2021	Solution:	mult_extracted_partition (Vivado IP Flow Target)
Version:	2020.3 (Build 3173277 on Wed Apr 7 05:07:49 MDT	Product	versal
Project:	ProjectDSDM2	Target device:	xcvm1802-vsvd1760-3HP-e-S
Status:	Fail		

Cosim Options

Tool: Vivado XSIM RTL: Verilog

Performance Estimates

Modules	Avg II	Max II	Min II	Avg Latency	Max Latency	Min Latency
kernel				11241	11241	11241
kernel_body_1				11240	11240	11240

The utilization percentages are all still under 50%, so the component has still resources available to further improvement of the performance.
The II Violation are still present on the loops, as can be seen from the detailed report in the following page:

Performance & Resource Estimates

Modules && Loops	Issue Type	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
kernel		-	11241	1,120E5	-	11242	-	no	44	726	493774	408640	0
kernel_body_1		-	11240	1,120E5	-	11240	-	no	0	726	493707	408605	0
reshape_transpose_1	II Violation	-	674	6,740E3	-	674	-	no	0	0	191	5865	0
RES_LOOP1	II Violation	-	672	6,720E3	62	61	11	yes	-	-	-	-	-
gemm_1	II Violation	-	1400	1,400E4	-	1400	-	no	0	0	110021	77257	0
GEMM_LOOP1	II Violation	-	732	7,320E3	122	61	11	yes	-	-	-	-	-
gemm_transpose_1	II Violation	-	1400	1,400E4	-	1400	-	no	0	242	123771	100071	0
GEMMTR_LOOP1	II Violation	-	732	7,320E3	122	61	11	yes	-	-	-	-	-
mult_operation_1	II Violation	-	1333	1,333E4	-	1333	-	no	0	0	7653	41419	0
MULT_LOOP1	II Violation	-	1331	1,331E4	121	121	11	yes	-	-	-	-	-
gemm	II Violation	-	1400	1,400E4	-	1400	-	no	0	0	108681	63024	0
GEMM_LOOP1	II Violation	-	732	7,320E3	122	61	11	yes	-	-	-	-	-
gemm_transpose	II Violation	-	1400	1,400E4	-	1400	-	no	0	242	122431	85838	0
GEMMTR_LOOP1	II Violation	-	732	7,320E3	122	61	11	yes	-	-	-	-	-
transpose_reshape_1	II Violation	-	674	6,740E3	-	674	-	no	0	0	7086	5143	0
TRANSRES_LOOP1	II Violation	-	672	6,720E3	62	61	11	yes	-	-	-	-	-
MTRANS_LOOP2		-	11	110.000	2	1	11	yes	-	-	-	-	-
MTRANS_LOOP2		-	11	110.000	2	1	11	yes	-	-	-	-	-
MTRANS_LOOP2		-	11	110.000	2	1	11	yes	-	-	-	-	-
MTRANS_LOOP2		-	11	110.000	2	1	11	yes	-	-	-	-	-
MTRANS_LOOP2		-	11	110.000	2	1	11	yes	-	-	-	-	-
MTRANS_LOOP2		-	11	110.000	2	1	11	yes	-	-	-	-	-
MTRANS_LOOP2		-	11	110.000	2	1	11	yes	-	-	-	-	-
MTRANS_LOOP2		-	11	110.000	2	1	11	yes	-	-	-	-	-
MTRANS_LOOP2		-	11	110.000	2	1	11	yes	-	-	-	-	-
MTRANS_LOOP2		-	11	110.000	2	1	11	yes	-	-	-	-	-
MTRANS_LOOP2		-	11	110.000	2	1	11	yes	-	-	-	-	-

This solution seems to be a good starting point for further work, since the estimation of clock cycles is very good, but unfortunately inspecting the failure in Co-Simulation is a difficult task.

Solution 13: Mult Extracted Loop Merging

This solution will be just mentioned because it didn't bring any improvement with respect to solution 11. The solution is the same as solution 11 with the LOOP_MERGE directive set on the "gemm" loops. It is reported here just for completeness.

4 - **Final considerations**

Among all the solutions considered, the number 8 is the one that obtained the best results. It can perform the computation in 11359 cycles on the target architecture without using too much of the space available.

This is surely not the best solution in the absolute way, since I believe that better results could be obtained: the target hardware has still resources available and not all the warnings have been solved by the solutions presented here. Moreover, even if the original source code includes a certain degree of serial computations, which cannot really be optimized for the execution in hardware beside some pipelining (for example, every time an operation requires the results of a previous one), other things could be really sped up. I am referring in particular to the operations between elements of the matrices: with the correct number of multipliers they could be done in parallel by saving a considerable amount of time, spending just the cycles needed to read data, perform a multiplication and propagating the output. While this is quite easy to think at an intuitive level, it is not immediate to instruct the HLS tool to do exactly that.

In the end, what I learned from this project is that HLS tools offer a great bridge between high level code and hardware implementation, making the process of obtaining an architecture much faster and, most importantly, allowing the designer to reason at a higher level, closer to human thinking, without having to deal with hardware description languages. This is very powerful since the architecture can be tested and eventually modified according to the desired behaviour by editing a more understandable code. The drawback is that the synthesis tools sometimes act like a “black box” where the output is hard to predict. Some results surprised me since they performed way worse or way better than my expectations. I believe that this kind of tools require great experience and an extensive knowledge of the hardware architectures in order to be exploited at full power.