

Санкт-Петербургский политехнический университет Петра Великого  
Институт компьютерных наук и технологий  
Кафедра компьютерных систем и программных технологий

Курсовая работа

Предмет: Проектирование реконфигурируемых гибридных вычислительных систем

Тема: Умножение матриц

**Студенты:**

Соболь В.

Темнова А.С.

Группа: 13541/3

**Преподаватель:**

Антонов А.П.

Санкт-Петербург  
2019

# Содержание

<b>1. Аннотация</b>	<b>3</b>
<b>2. Введение</b>	<b>3</b>
<b>3. «Простое» умножение матриц</b>	<b>4</b>
3.1. Исходный код . . . . .	4
3.2. Скрипт . . . . .	5
3.3. Моделирование . . . . .	6
3.4. Решение без оптимизаций . . . . .	7
3.5. Конвейеризация внутреннего цикла . . . . .	8
3.6. Разбиение входных данных . . . . .	11
3.7. Конвейеризация всего устройства . . . . .	14
3.8. Сравнение . . . . .	14
<b>4. Блочное умножение матриц</b>	<b>15</b>
4.1. Исходный код . . . . .	17
4.2. Скрипт . . . . .	20
4.3. Моделирование . . . . .	21
4.4. Оптимизация потока данных . . . . .	21
4.4.1. Анализ модулей в иерархии . . . . .	23
4.4.2. Анализ решения . . . . .	26
4.5. Конвейеризация внутреннего цикла умножения . . . . .	26
4.5.1. Анализ модулей в иерархии . . . . .	28
4.5.2. Анализ решения . . . . .	31
4.6. Конвейеризация внешнего цикла умножения . . . . .	32
4.6.1. Анализ модулей в иерархии . . . . .	33
4.6.2. Анализ решения . . . . .	36
4.7. Сравнение решений . . . . .	36
<b>5. Вывод</b>	<b>37</b>
<b>Список литературы</b>	<b>38</b>

# 1. Аннотация

В данной работе рассматриваются два подхода к вычислению произведения двух матриц. Сначала рассматривается «простая» реализация, то есть та, которая принимает две матрицы как вход и выводит результат их умножения. Затем рассматривается блочное умножение матриц. Здесь входные матрицы подаются в функцию порциями, а функция вычисляет частичные результаты.

# 2. Введение

Умножение матриц — это бинарная операция, которая объединяет две матрицы в третью. Сама операция может быть описана как линейная операция над векторами, которые составляют две матрицы. Самая распространенная форма умножения матриц — это матричное произведение.

Матричное произведение  $AB$  создаёт матрицу размерности  $n \times p$ , исходные матрицы  $A$  имеет размерность  $n \times m$  и матрица  $B$  имеет размерность  $m \times p$ .

Произведение матриц выполняется по следующей формуле:

$$\mathbf{A} = \begin{bmatrix} A_{11} & A_{12} & \cdots & A_{1m} \\ A_{21} & A_{22} & \cdots & A_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n1} & A_{n2} & \cdots & A_{nm} \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} B_{11} & B_{12} & \cdots & B_{1p} \\ B_{21} & B_{22} & \cdots & B_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ B_{m1} & B_{m2} & \cdots & B_{mp} \end{bmatrix}$$
$$\mathbf{AB} = \begin{bmatrix} (\mathbf{AB})_{11} & (\mathbf{AB})_{12} & \cdots & (\mathbf{AB})_{1p} \\ (\mathbf{AB})_{21} & (\mathbf{AB})_{22} & \cdots & (\mathbf{AB})_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ (\mathbf{AB})_{n1} & (\mathbf{AB})_{n2} & \cdots & (\mathbf{AB})_{np} \end{bmatrix}$$

Рис. 2.1. Произведение матриц

где операция  $(AB)_{ij}$  вычисляется как  $(AB)_{ij} = \sum_{k=1}^m A_{ik}B_{kj}$ .

Умножение матриц является фундаментальной операцией в численных алгоритмах. Вычисление продукта между большими матрицами может занять значительное время. Следовательно, это критически важная часть многих проблем численных вычислений. По сути, матрицы представляют собой линейные преобразования между векторными пространствами; Умножение матриц обеспечивает способ составления линейных преобразований. Приложения включают в себя линейно изменяющиеся координаты (например, перемещение, вращение в графике), проблемы больших размеров в статистической физике (например, метод матрицы переноса) и графовые операции (например, определение, существует ли путь от одной вершины к другой). Таким образом, это хорошо изученная проблема, и существует множество алгоритмов, направленных на повышение ее производительности и уменьшение использования памяти.

### 3. «Простое» умножение матриц

В данном разделе исследуется «простой» алгоритм умножения матриц.

#### 3.1. Исходный код

Ниже приведены исходный код устройства и исходный код теста для данного устройства.

```
1 #include "matmul.h"
2
3 void matmul(int A[N][M], int B[M][P], int AB[N][P]) {
4     /* for each row and column of AB */
5     row: for(int i = 0; i < N; ++i) {
6         col: for(int j = 0; j < P; ++j) {
7             /* compute (AB)i,j */
8             int ABij = 0;
9             product: for(int k = 0; k < M; ++k) {
10                 ABij += A[i][k] * B[k][j];
11             }
12             AB[i][j] = ABij;
13         }
14     }
15 }
```

Рис. 3.1. Исходный код устройства

```
1 #define N 128
2 #define M 128
3 #define P 128
```

Рис. 3.2. Заголовочный файл

```

1 #include <stdio.h>
2 #include "matmul.h"
3
4 void data(int seed, int A[N][M], int B[M][P], int AB[N][P]) {
5     for(int i = 0; i < N; ++i) {
6         for(int j = 0; j < P; ++j) {
7             int ABij = 0;
8             for(int k = 0; k < M; ++k) {
9                 A[i][k] = (i * N + k) * seed;
10                B[k][j] = (k * M + j) * seed;
11                ABij += A[i][k] * B[k][j];
12            }
13            AB[i][j] = ABij;
14        }
15    }
16 }
17
18 int matrix_equal(int A[N][P], int B[N][P]) {
19     for(int i = 0; i < N; ++i) {
20         for(int j = 0; j < P; ++j) {
21             if(A[i][j] != B[i][j]) {
22                 return 0;
23             }
24         }
25     }
26     return 1;
27 }
28
29 int main() {
30     int A_in[N][M], B_in[M][P];
31     int AB_actual[N][P], AB_expected[N][P];
32
33     int pass = 1;
34
35     for (int i = 1; i < 4; ++i){
36         data(i, A_in, B_in, AB_expected);
37
38         matmul(A_in, B_in, AB_actual);
39
40         if (!matrix_equal(AB_actual, AB_expected)){
41             pass = 0;
42         }
43     }
44
45     if(pass){
46         printf("Test_passed\n");
47         return 0;
48     } else{
49         printf("Test_failed\n");
50         return -1;
51     }
52 }
53

```

Рис. 3.3. Исходный код теста

## 3.2. Скрипт

Ниже приведен скрипт для автоматизации выполнения исследований.

```

1 open_project      -reset complete_matmul
2
3 add_files         matmul.c
4 add_files         matmul.h
5 add_files -tb     matmul_test.c
6
7 set_top           matmul
8
9 set solutions [list no_flags pipeline_inner reshape_part_inner]
10
11 foreach sol $solutions {
12     open_solution -reset $sol
13     set_part {xa7a12tcs325-1q}
14     create_clock -period 10
15     set_clock_uncertainty 0.1
16
17     if {$sol == "pipeline_inner" || $sol == "reshape_part_inner"} {
18         set_directive_pipeline -II 1 matmul/col
19     }
20     if {$sol == "reshape_part_inner"} {
21         set_directive_array_reshape -type complete -dim 2 matmul A
22         set_directive_array_reshape -type complete -dim 1 matmul B
23     }
24     csim_design
25     csynth_design
26 }
27
28 exit

```

Рис. 3.4. Скрипт выполнения

### 3.3. Моделирование

Как видно по результатам моделирования, тест проходит успешно.

```

INFO: [APCC 202-1] APCC is done.
      Generating csim.exe
Test passed
INFO: [SIM 211-1] CSim done with 0 errors.
INFO: [SIM 211-3] ***** CSIM finish *****

```

Рис. 3.5. Результат моделирования

### 3.4. Решение без оптимизаций

## Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.470	0.10

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
8421633	8421633	8421633	8421633	none

Рис. 3.6. Performance estimates

Utilization Estimates				
Summary				
Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	3	0	213
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	-	-	-	-
Multiplexer	-	-	-	77
Register	-	-	213	-
Total	0	3	213	290
Available	40	40	16000	8000
Utilization (%)	0	7	1	3

Рис. 3.7. Utilization estimates

Performance Profile		Resource Profile			
	Pipelined	Latency	Iteration Latency	Initiation Interval	Trip count
matmul	-	8421633	-	8421634	-
row	no	8421632	65794	-	128
col	no	65792	514	-	128
product	no	512	4	-	128

Рис. 3.8. Performance profile

	BRAM	DSP	FF	LUT	Bits P0	Bits P1	Bits P2	Banks/Depth	Words	W*Bits*Banks
matmul	0	3	213	290						
I/O Ports(3)					96					
Instances(0)	0	0	0	0						
Memories(0)	0		0	0	0			0	0	0
Expressions(11)	0	3	0	213	160	142	0			
Registers(14)			213		229					
Channels(0)	0		0	0	0		0		0	0
Multiplexers(5)	0		0	77	57		0			
DSP(0)		0								

Рис. 3.9. Resource profile

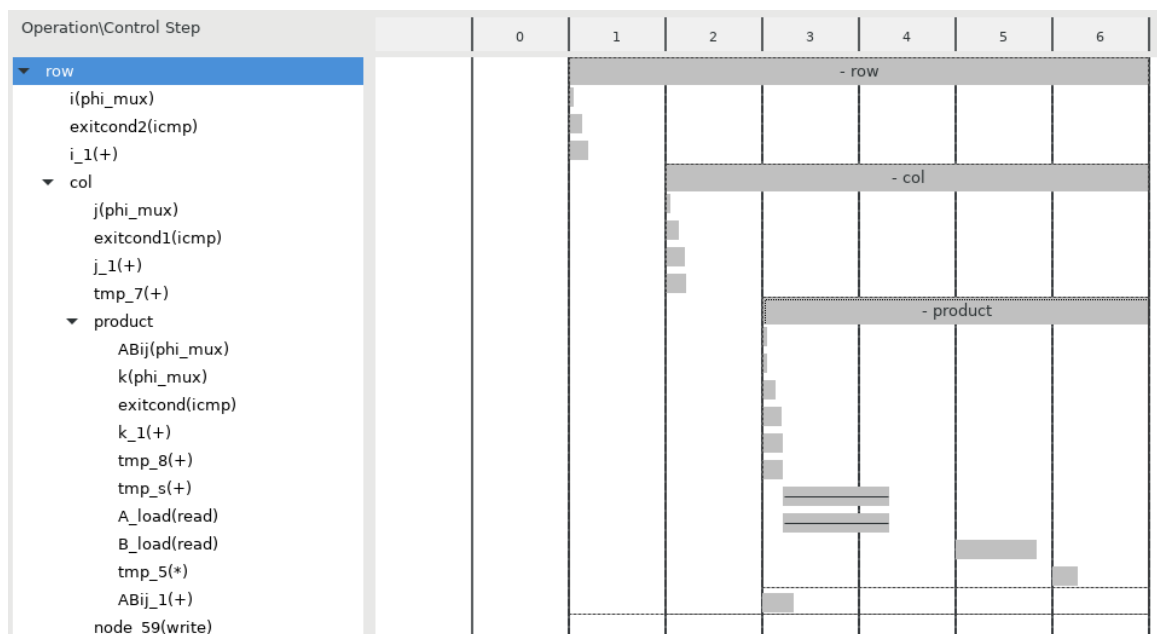


Рис. 3.10. Scheduler viewer

По данным, приведённым выше, можно сказать, что:

1. Устройство не требовательно к ресурсам
2. Все вычисления выполняются последовательно, что приводит к задержке более 8млн тактов для умножения матриц размерностью  $12 \times 128$

### 3.5. Конвейеризация внутреннего цикла

В данном решении применяется директиву pipeline к циклу col с целевым значением П равным 1. В результате, самый внутренний цикл должен быть полностью развернут, и мы ожидаем, что полученная схема будет включать примерно М операторов и иметь задержку примерно  $N * P$  циклов.



### Performance Estimates

#### Timing (ns)

##### Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	9.010	0.10

#### Latency (clock cycles)

##### Summary

Latency		Interval		
min	max	min	max	Type
1048582	1048582	1048582	1048582	none

Рис. 3.11. Performance estimates

### Utilization Estimates

#### Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	6	0	4822
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	-	-	-	-
Multiplexer	-	-	-	1542
Register	-	-	1604	-
<b>Total</b>	<b>0</b>	<b>6</b>	<b>1604</b>	<b>6364</b>
Available	40	40	16000	8000
<b>Utilization (%)</b>	<b>0</b>	<b>15</b>	<b>10</b>	<b>79</b>

Рис. 3.12. Utilization estimates

Performance Profile		Resource Profile			
	Pipelined	Latency	Iteration Latency	Initiation Interval	Trip count
matmul	-	1048582	-	1048583	-
row_col	yes	1048580	69	64	16384

Рис. 3.13. Performance profile

Performance Profile		Resource Profile									
		BRAM	DSP	FF	LUT	Bits P0	Bits P1	Bits P2	Banks/Depth	Words	W*Bits*Banks
matmul		0	6	1604	6364						
I/O Ports(3)						96					
Instances(0)		0	0	0	0						
Memories(0)		0		0	0	0			0	0	
Expressions(211)		0	6	0	4822	3624	2462	16			
Registers(72)				1604		1631					
Channels(0)		0		0	0	0			0	0	
Multiplexers(12)		0		0	1542	120			0		
DSP(0)			0								

Рис. 3.14. Resource profile

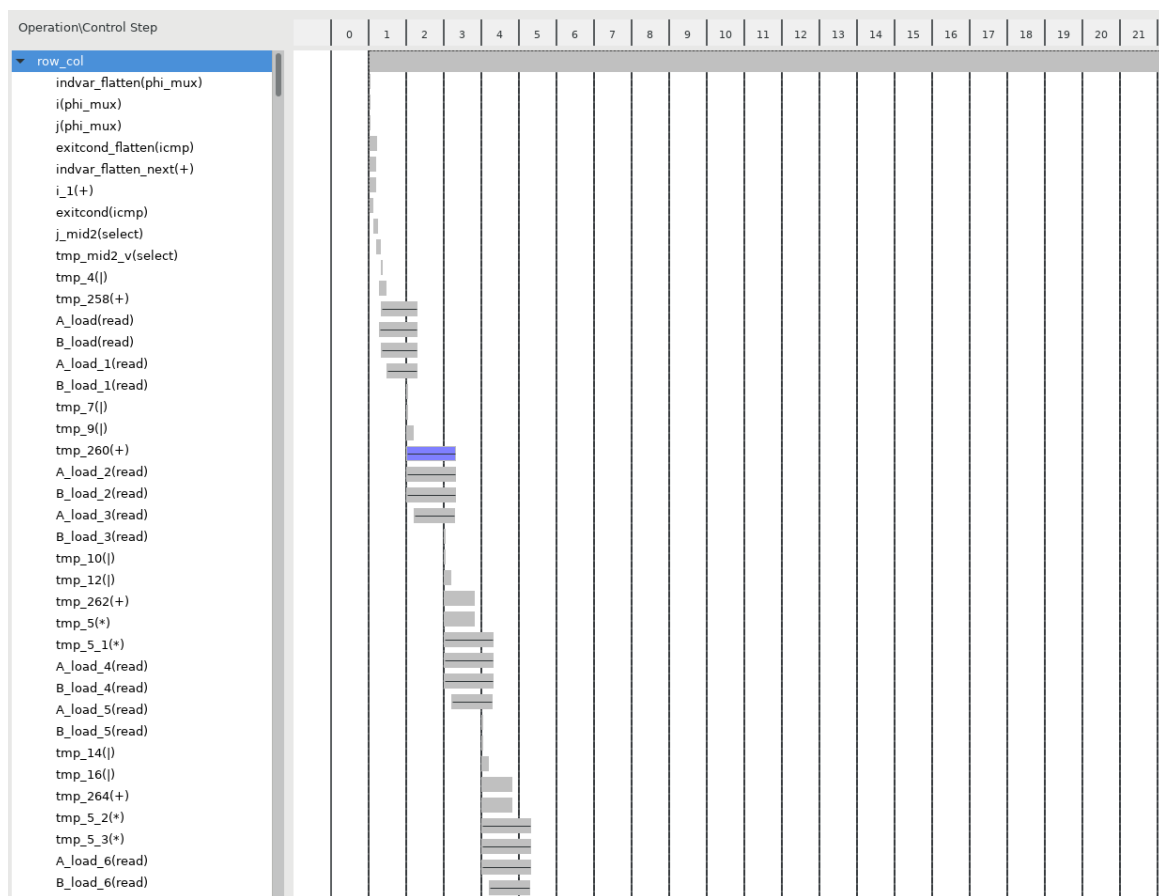


Рис. 3.15. Scheduler viewer

По данным, приведённым выше, можно сказать, что:

1. Устройство стало более требовательно к ресурсам, в сравнении с предыдущим решением, но это всё ещё не критично
2. Конвейеризация внутреннего цикла позволила сократить задержку до приблизительно 1 млн тактов для умножения матриц размерностью  $12 \times 128$
3. По данным из Scheduler viewer видно, что вычисления упираются в доступ к данным

### 3.6. Разбиение входных данных

Так как предыдущее решение упирается в доступ к данным, его можно попробовать улучшить, используя разбиение данных.

Выполнение большого количества операций в каждом цикле требует возможности обеспечить все необходимые операнды и хранить результаты каждой операции. Можно использовать директиву `array_partition` для увеличения количества обращений, которые могут быть выполнены в каждой памяти. Если каждый доступ к памяти может быть определен во время компиляции, тогда разбиение массива — это простой и эффективный способ увеличить количество обращений к памяти, которое может быть выполнено в каждый такт.

В данном решении, мы используем немного другой вид разбиения массива, используя директиву `array_reshape` для выполнения разбиения массива. Эта директива не только разделяет адресное пространство памяти в отдельные блоки памяти, но затем объединяет блоки памяти в одну память. Это преобразование увеличивает ширину данных памяти, используемой для хранения массива, но не изменяет общее количество сохраняемых битов. Различия показаны на рисунке ниже.

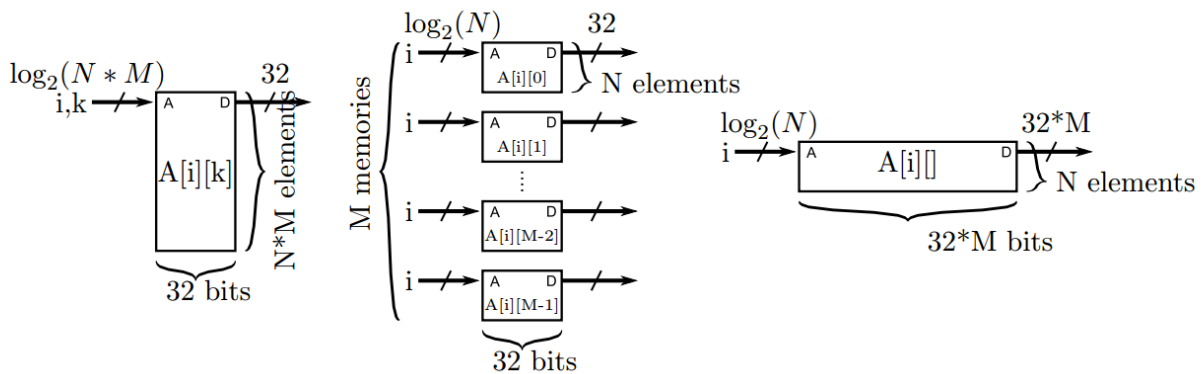


Рис. 3.16. Различия между `array_partition` и `array_reshape`

На рисунке выше, слева находится исходный массив, состоящий из  $N \times M$  элементов. В середине массив был преобразован с использованием директивы `array_partition`, в результате чего было получено  $M$  памяти, каждая из которых содержит  $N$  элементов. Справа массив был преобразован с использованием директивы `array_reshape`, в результате чего в одной памяти было  $N$  местоположений, и каждое местоположение содержит  $M$  элементов исходного массива.

Как `array_reshape` массива, так и `array_partition` увеличивают количество элементов массива, которые можно прочитать за каждый такт. Они также поддерживают одинаковые параметры, позволяя циклическое и блочное разбиение или разбиение по разным измерениям многомерного массива. В случае `array_reshape` массива каждый элемент должен иметь одинаковый адрес в преобразованном массиве, тогда как с `array_partition`, адреса в преобразованном массиве могут быть не связаны. Хотя может показаться, что всегда стоит использовать `array_partition`, потому что он более гибкий, он делает каждую отдельную память меньше, что иногда может привести к неэффективному использованию памяти.

Директива `array_reshape` приводит к большим блокам памяти, которые иногда могут быть более эффективно отображены в ресурсы ПЛИС.

В частности, наименьшая гранулярность блоков RAM (BRAM) в Xilinx Virtex Ultrascale [1] + составляют 18 Кбит с несколькими различными поддерживаемыми комбинациями гл-

бины и ширины. Когда разделы массива становятся меньше, чем около 18 Кбит, тогда BRAMs больше не используется эффективно. Если мы начнем с оригинального массива, который является 4-битным массивом с размерами [1024] [4], этот массив может помещаться в один ресурс BRAM, настроенный как память 4 Кбит x 4. Полное разбиение этого массива во втором измерении приведет к 4 1 Кбит x 4 памяти, каждая из которых гораздо меньше, чем один ресурс BRAM. Применение `array_reshape` массива вместо использования `array_partition` массива приводит к памяти, которая составляет 1 Кбит x 16, поддерживаемая конфигурация BRAM.

Performance Estimates

▣ Timing (ns)

▣ Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	9.010	0.10

▣ Latency (clock cycles)

▣ Summary

Latency		Interval		
min	max	min	max	Type
16391	16391	16391	16391	none

Рис. 3.17. Performance estimates

Utilization Estimates				
▣ Summary				
Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	384	0	7186
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	-	-	-	-
Multiplexer	-	-	-	75
Register	0	-	14362	96
Total	0	384	14362	7357
Available	40	40	16000	8000
Utilization (%)	0	960	89	91

Рис. 3.18. Utilization estimates

Performance Profile		Resource Profile			
	Pipelined	Latency	Iteration Latency	Initiation Interval	Trip count
▼ ● matmul	-	16391	-	16392	-
● row_col	yes	16389	7	1	16384

Рис. 3.19. Performance profile

	BRAM	DSP	FF	LUT	Bits P0	Bits P1	Bits P2	Banks/Depth	Words	W*Bits*Banks
matmul	0	384	14170	7261	8224					
I/O Ports(3)					8224					
Instances(0)	0	0	0	0						
Memories(0)	0	0	0	0	0			0	0	0
Expressions(265)	0	384	0	7186	8221	8230	16			
Registers(455)			14170	14170						
Channels(0)	0	0	0	0	0		0		0	0
Multiplexers(7)	0	0	0	75	42		0			
DSP(0)		0								

Рис. 3.20. Resource profile

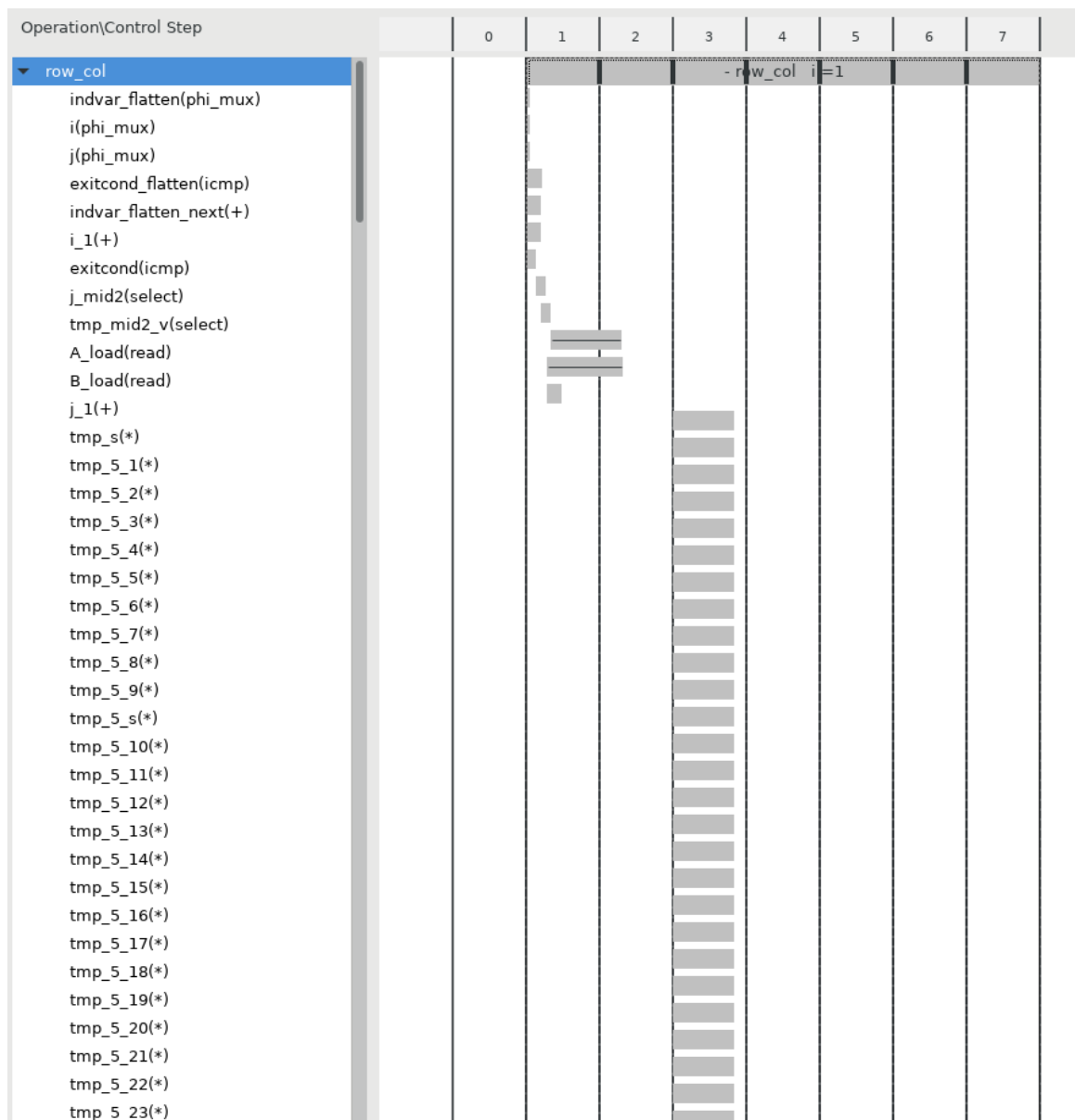


Рис. 3.21. Scheduler viewer

По данным, приведённым выше, можно сказать, что:

1. Устройство стало более требовательно к ресурсам, и не может быть реализовано на выбранной микросхеме

2. Вычисления на внутреннем цикле выполняются полностью параллельно, что позволяет достичь задержки в 16 тысяч тактов для умножения матриц размерностью  $12 \times 128$

### 3.7. Конвейеризация всего устройства

Конвейеризация всего устройства будет требовать значительно больше ресурсов, но в тоже время может столь же значительно сократить время вычислений.

```
WARNING: [ANALYSIS 214-1] Tool encounters 16384 load/store instructions to analyze which may result in long runtime.
ERROR: [XFORM 203-1403] Unsupported enormous number of load/store instructions: 'matmul'.
ERROR: [HLS 200-70] Failed building synthesis data model.
command 'ap_source' returned error code
while executing
"source [lindex $::argv 1] "
("uplevel" body line 1)
invoked from within
"uplevel \#0 { source [lindex $::argv 1] } "
INFO: [HLS 200-112] Total elapsed time: 900.983 seconds; peak allocated memory: 75.784 MB.
INFO: [Common 17-206] Exiting vivado_hls at Mon Dec 23 16:27:01 2019...
```

Рис. 3.22. Конвейеризация всего устройства

К сожалению, Vivado HLS не смог синтезировать созданную схему.

### 3.8. Сравнение

Ниже приведено сравнение созданных решений.

Performance Estimates				
Timing (ns)				
Clock		no_flags	pipeline_inner	reshape_part_inner
ap_clk	Target	10.00	10.00	10.00
	Estimated	8.470	9.010	9.010
Latency (clock cycles)				
		no_flags	pipeline_inner	reshape_part_inner
Latency	min	8421633	1048582	16391
	max	8421633	1048582	16391
Interval	min	8421633	1048582	16391
	max	8421633	1048582	16391

Рис. 3.23. Performance estimates

Utilization Estimates			
	no_flags	pipeline_inner	reshape_part_inner
BRAM_18K0	0	0	0
DSP48E	3	6	384
FF	213	1604	14362
LUT	290	6364	7357

Рис. 3.24. Utilization estimates

По сравнению видно, что в результате оптимизации удалось сократить время вычисления с 8 млн тактов до 16391 тактов, но при этом значительно возросли требования к вычислительным ресурсам.

## 4. Блочное умножение матриц

Блочная матрица интерпретируется как разделенная на разные подматрицы. Это можно визуализировать, рисуя различные горизонтальные и вертикальные линии на элементах матрицы. Полученные «блоки» можно рассматривать как подматрицы исходной матрицы. В качестве альтернативы, мы можем рассматривать исходную матрицу как матрицу блоков. Это, естественно, приводит ко многим иерархическим алгоритмам в линейной алгебре, где мы вычисляем матричные операции, такие как умножение матриц, на матрицах больших блоков, разбивая их на меньшие матричные операции на самих блоках.

Например, когда мы говорим об операции умножения матриц между матрицами  $A$  и  $B$ , мы обычно можем рассматривать каждый элемент матриц  $A_{11}$  или  $B_{23}$  как одно целое число или, возможно, комплексное число. В качестве альтернативы, мы можем рассматривать каждый элемент в этих матричных операциях как блок исходной матрицы. В этом случае, пока размеры отдельных блоков совместимы, мы просто должны вместо этого выполнять правильные матричные операции оригинальных скалярных операций.

Так, например, чтобы вычислить  $AB_{11}$ , необходимо вычислить два матричных произведения и сумму двух матриц, для вычисления  $A_{11}B_{11} + A_{12}B_{21} + A_{13}B_{31}$ .

Разбиение матриц на блоки оказывается очень полезной техникой по ряду причин. Одна из причин заключается в том, что такое разбиение — это простой способ найти больше структуры в алгоритме, который мы можем исследовать. Фактически, некоторые из оптимизаций, которые мы уже видели как преобразования цикла, такие как развертывание цикла, можно рассматривать как конкретные простые формы разбиения на блоки.

Другая причина в том, что мы можем выбрать способ разбиения матрицы в соответствии с естественной структурой матрицы. Если матрица имеет большой блок нулей, то многие отдельные произведения могут быть равны нулю. Если мы хотим пропустить эти отдельные произведения, это может быть затруднено в статически запланированном конвейере, тогда как может быть проще пропустить большой блок нулей. Многие матрицы являются блочно-диагональными, где блоки на диагонали отличны от нуля, а блоки вне диагонали равны нулю.

Еще одна причина заключается в том, что блочная декомпозиция приводит к множеству небольших вычислений, работающих с меньшими наборами данных. Это увеличивает локальность данных вычислений.

В процессорных системах обычно выбирают размеры блоков, которые удобно соответствуют иерархии памяти процессора или естественному размеру векторных типов данных, поддерживаемых процессором. Точно так же в FPGA мы можем выбрать размеры разбиения на блоки, чтобы соответствовать доступному объему встроенной памяти или количеству операторов умножения, которые мы можем выделить.



Рис. 4.1. Пример разбиения матрицы на блоки

Также в проектах, работающих с большими наборами данных, такими как большие матрицы, иногда не все данные могут быть доступны сразу. Поскольку маловероятно, что наш вычислитель сможет сразу обрабатывать все входные данные, мы можем создать вычислитель, который получает входные данные только непосредственно перед тем, как это потребуется.

Это позволяет вычислителю более эффективно использовать доступную внутрикристальную память. Это называется потоковой архитектурой, поскольку мы передаем входные данные (и, возможно, выходные данные) по одной порции за раз, а не все сразу. Потоковые архитектуры распространены во многих приложениях. В некоторых случаях это происходит из-за сознательного выбора дизайна, который мы делаем, чтобы разбить большие вычисления на несколько меньших вычислений.

Например, мы можем спроектировать матричную систему умножения, которая считывает и обрабатывает один блок данных за раз из внешней памяти. В других случаях мы можем обработать поток данных, потому что данные отбираются в реальном времени из физического мира, например, из аналого-цифрового преобразователя. В других случаях данные, которые мы обрабатываем, могут быть просто созданы в последовательности от



предыдущего вычисления или ускорителя.

Одним из потенциальных преимуществ потоковой передачи является сокращение памяти, которую мы можем использовать для хранения входных и выходных данных. Здесь предполагается, что мы можем работать с данными порциями, создавать частичные результаты, а затем мы закончим с этими данными, поэтому нам не нужно их хранить. Когда поступают следующие данные, мы можем перезаписать старые данные, что приведет к уменьшению объема памяти.

## 4.1. Исходный код

Ниже приведены исходный код устройства и исходный код теста для данного устройства.

```
1 #include "matmul.h"
2
3 void matmul(hls::stream<blockvec> &Arows, hls::stream<blockvec> &Bcols, blockmat
  ↪ & ABpartial, int iteration){
4     #pragma HLS DATAFLOW
5
6     static DTYPE A[BLOCK_SIZE][SIZE];
7     if(iteration % (SIZE/BLOCK_SIZE) == 0){ //only load the A rows when necessary
8         loadA: for(int i = 0; i < SIZE; i++) {
9             blockvec tempA = Arows.read();
10            for(int j = 0; j < BLOCK_SIZE; j++) {
11                A[j][i] = tempA.block[j];
12            }
13        }
14    }
15
16    DTYPE AB[BLOCK_SIZE][BLOCK_SIZE] = { 0 };
17    partialsum: for(int k=0; k < SIZE; k++) {
18        blockvec tempB = Bcols.read();
19        innerB: for(int i = 0; i < BLOCK_SIZE; i++) {
20            for(int j = 0; j < BLOCK_SIZE; j++) {
21                AB[i][j] = AB[i][j] + A[i][k] * tempB.block[j];
22            }
23        }
24    }
25
26    writeoutput: for(int i = 0; i < BLOCK_SIZE; i++) {
27        for(int j = 0; j < BLOCK_SIZE; j++) {
28            ABpartial.matrix[i][j] = AB[i][j];
29        }
30    }
31
32 }
```

Рис. 4.2. Исходный код устройства

```

1 #pragma once
2
3 #include "hls_stream.h"
4 #include <iostream>
5 #include <iomanip>
6 #include <vector>
7
8 typedef int DTYPE;
9 const int BLOCK_SIZE = 4;
10 const int SIZE = 128;
11
12 typedef struct {
13     DTYPE block[BLOCK_SIZE];
14 } blockvec;
15
16 typedef struct {
17     DTYPE matrix[BLOCK_SIZE][BLOCK_SIZE];
18 } blockmat;
19
20 void matmul(hls::stream<blockvec> &Arows, hls::stream<blockvec> &Bcols, blockmat
    ↪ & ABpartial, int iteration);

```

Рис. 4.3. Заголовочный файл

```

1 #include <stdio.h>
2 #include "matmul.h"
3
4 void data(int seed, int A[SIZE][SIZE], int B[SIZE][SIZE], int AB[SIZE][SIZE]) {
5     for(int i = 0; i < SIZE; ++i) {
6         for(int j = 0; j < SIZE; ++j) {
7             int ABij = 0;
8             for(int k = 0; k < SIZE; ++k) {
9                 A[i][k] = (i * SIZE + k) * seed;
10                B[k][j] = (k * SIZE + j) * seed;
11                ABij += A[i][k] * B[k][j];
12            }
13            AB[i][j] = ABij;
14        }
15    }
16 }
17
18 int matrix_equal(int A[SIZE][SIZE], int B[SIZE][SIZE]) {
19     for(int i = 0; i < SIZE; ++i) {
20         for(int j = 0; j < SIZE; ++j) {
21             if(A[i][j] != B[i][j]) {
22                 return 0;
23             }
24         }
25     }
26     return 1;
27 }

```

Рис. 4.4. Исходный код теста, часть 1

```

1 void block_matmul(int A[SIZE][SIZE], int B[SIZE][SIZE], int AB[SIZE][SIZE]) {
2     hls::stream<blockvec> A_matrix("A_matrix");
3     hls::stream<blockvec> B_matrix("B_matrix");
4     blockvec A_matrix_block, B_matrix_block;
5     blockmat block_out;
6
7     int it = 0;
8     for(int row = 0; row < SIZE; row = row + BLOCK_SIZE) {
9         for(int col = 0; col < SIZE; col = col + BLOCK_SIZE) {
10
11             for(int k = 0; k < SIZE; k++) {
12                 for(int i = 0; i < BLOCK_SIZE; i++) {
13                     if(it % (SIZE/BLOCK_SIZE) == 0) {
14                         A_matrix_block.block[i] = A[row+i][k];
15                     }
16                     B_matrix_block.block[i] = B[k][col+i];
17                 }
18                 if(it % (SIZE/BLOCK_SIZE) == 0) {
19                     A_matrix.write(A_matrix_block);
20                 }
21                 B_matrix.write(B_matrix_block);
22             }
23
24             matmul(A_matrix, B_matrix, block_out, it);
25
26             for(int i = 0; i < BLOCK_SIZE; i++){
27                 for(int j = 0; j < BLOCK_SIZE; j++){
28                     AB[row+i][col+j] = block_out.matrix[i][j];
29                 }
30             }
31
32             it = it + 1;
33         }
34     }
35 }

```

Рис. 4.5. Исходный код теста, часть 2

```

1 int main() {
2     int A_in[SIZE][SIZE], B_in[SIZE][SIZE];
3     int AB_actual[SIZE][SIZE], AB_expected[SIZE][SIZE];
4
5     int pass = 1;
6
7     for (int i = 1; i < 4; ++i){
8         data(i, A_in, B_in, AB_expected);
9
10        block_matmul(A_in, B_in, AB_actual);
11
12        if (!matrix_equal(AB_actual, AB_expected)){
13            pass = 0;
14        }
15    }
16
17    if (pass){
18        printf("Test_passed\n");
19        return 0;
20    } else{
21        printf("Test_failed\n");
22        return -1;
23    }
24 }
25 }

```

Рис. 4.6. Исходный код теста, часть 3

Входные массивы A и B разбиваются на блоки, представляющие собой непрерывный набор строк и столбцов соответственно. Используя эти блоки, мы вычисляем часть произведения AB. Затем мы передаем следующий набор блоков, вычисляем еще одну часть AB, пока не закончится умножение всей матрицы.

SIZE определяет количество строк и столбцов в матрицах для умножения.

Константа BLOCK\_SIZE определяет количество строк из A и количество столбцов из B, с которыми мы работаем при каждом выполнении. Это также определяет, сколько данных мы одновременно передаем в функцию. Выходные данные, которые мы получаем из функции при каждом выполнении, представляют собой часть  $BLOCK\_SIZE \times BLOCK\_SIZE$  матрицы AB.

Тип данных blockvec используется для передачи строк BLOCK\_SIZE в A и столбцов B в функцию при каждом выполнении.

Тип данных blockmat используется для хранения частичных результатов для матрицы AB. Тип данных blockmat представляет собой структуру, состоящую из массива  $BLOCK\_SIZE \times BLOCK\_SIZE$  и содержит результирующие значения от одного выполнения функции matmul.

Класс hls stream — это один из способов в Vivado HLS для создания структуры данных FIFO, которая хорошо работает при моделировании и синтезе. Элементы отправляются в последовательном порядке с помощью функции write и извлекаются с использованием функции read.

## 4.2. Скрипт

Ниже приведен скрипт для автоматизации выполнения исследований.

```

1 open_project      -reset block_matmul
2
3 add_files         matmul.cpp
4 add_files         matmul.h
5 add_files -tb     matmul_test.cpp
6
7 set_top           matmul
8
9 set solutions [list no_flags pipeline_inner pipeline_all]
10
11 foreach sol $solutions {
12     open_solution -reset $sol
13     set_part {xa7a12tcsg325-1q}
14     create_clock -period 10
15     set_clock_uncertainty 0.1
16
17     if {$sol == "pipeline_inner" || $sol == "pipeline_all"} {
18         set_directive_pipeline -II 1 matmul/loadA
19         set_directive_pipeline -II 1 matmul/writeoutput
20     }
21
22     if {$sol == "pipeline_inner"} {
23         set_directive_pipeline -II 1 matmul/innerB
24     }
25     if {$sol == "pipeline_all"} {
26         set_directive_pipeline -II 1 matmul/partialsum
27     }
28
29     csim_design
30     csynth_design
31 }
32
33 exit

```

Рис. 4.7. Скрипт выполнения

### 4.3. Моделирование

Как видно по результатам моделирования, тест проходит успешно.

```

INFO: [SIM 211-2] ***** CSIM start *****
INFO: [SIM 211-4] CSIM will launch GCC as the compiler.
    Compiling ../../../../matmul_test.cpp in debug mode
    Compiling ../../../../matmul.cpp in debug mode
    Generating csim.exe
Test passed
INFO: [SIM 211-1] CSim done with 0 errors.
INFO: [SIM 211-3] ***** CSIM finish *****

```

Рис. 4.8. Результат моделирования

### 4.4. Оптимизация потока данных

Директива `dataflow` в начале функции создает конвейер между частями функции, то есть, циклом `loadA`, `partialsum` и `writeout`. Использование этой директивы уменьшит  $\Pi$  функции `matmul`. Однако это ограничено наибольшим интервалом из всех трех частей

кода. То есть максимальный интервал для функции больше или равен интервалу трех частей: loadA, partialsum, writeoutput.

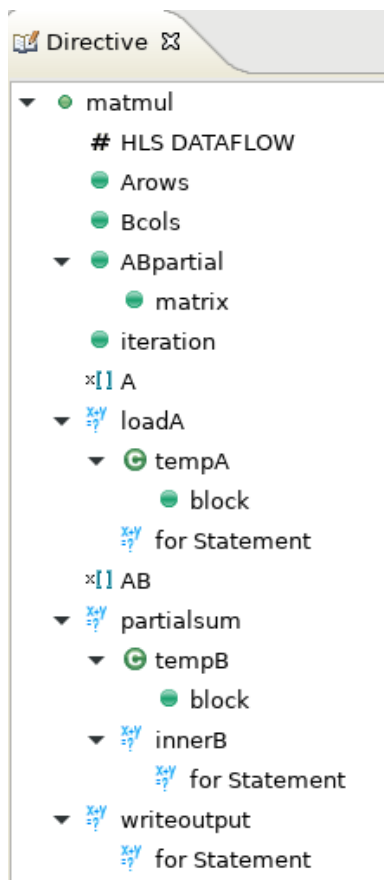


Рис. 4.9. Директивы

### Performance Estimates

#### Timing (ns)

##### Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.470	0.10

#### Latency (clock cycles)

##### Summary

Latency		Interval		Type
min	max	min	max	
7487	7487	7446	7446	dataflow

Рис. 4.10. Performance estimates

## Utilization Estimates

### Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	94
FIFO	2	-	67	124
Instance	-	3	454	775
Memory	0	-	64	8
Multiplexer	-	-	-	54
Register	-	-	9	-
<b>Total</b>	<b>2</b>	<b>3</b>	<b>594</b>	<b>1055</b>
Available	40	40	16000	8000
<b>Utilization (%)</b>	<b>5</b>	<b>7</b>	<b>3</b>	<b>13</b>

Рис. 4.11. Utilization estimates

	Negative Slack	BRAM	DSP	FF	LUT	Latency	Interval	Pipeline type
matmul	-	2	3	594	1055	7487	7446	dataflow
Loop_memset_AB_proc9	-	0	3	274	407	7445	7445	none
Block_proc8	-	0	0	152	223	1~769	1 ~ 769	none
Loop_writeoutput_pro	-	0	0	26	119	41	41	none
matmul_entry5	-	0	0	2	26	0	0	none

Рис. 4.12. Module hierarchy

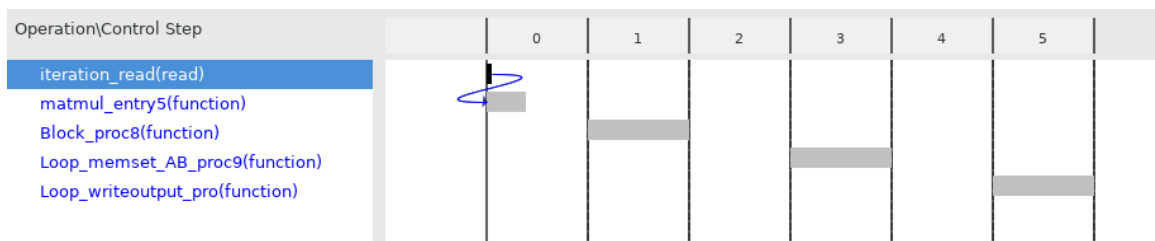


Рис. 4.13. Scheduler viewer

### 4.4.1. Анализ модулей в иерархии

#### 1. Loop\_memset\_AB\_proc9

	Pipelined	Latency	Iteration Latency	Initiation Interval	Trip count
Loop_memset_AB_proc9	-	7445	-	7445	-
memset_AB	no	19	5	-	4
partialsum	no	7424	58	-	128
innerB	no	56	14	-	4
innerB.1	no	12	3	-	4

Рис. 4.14. Performance profile

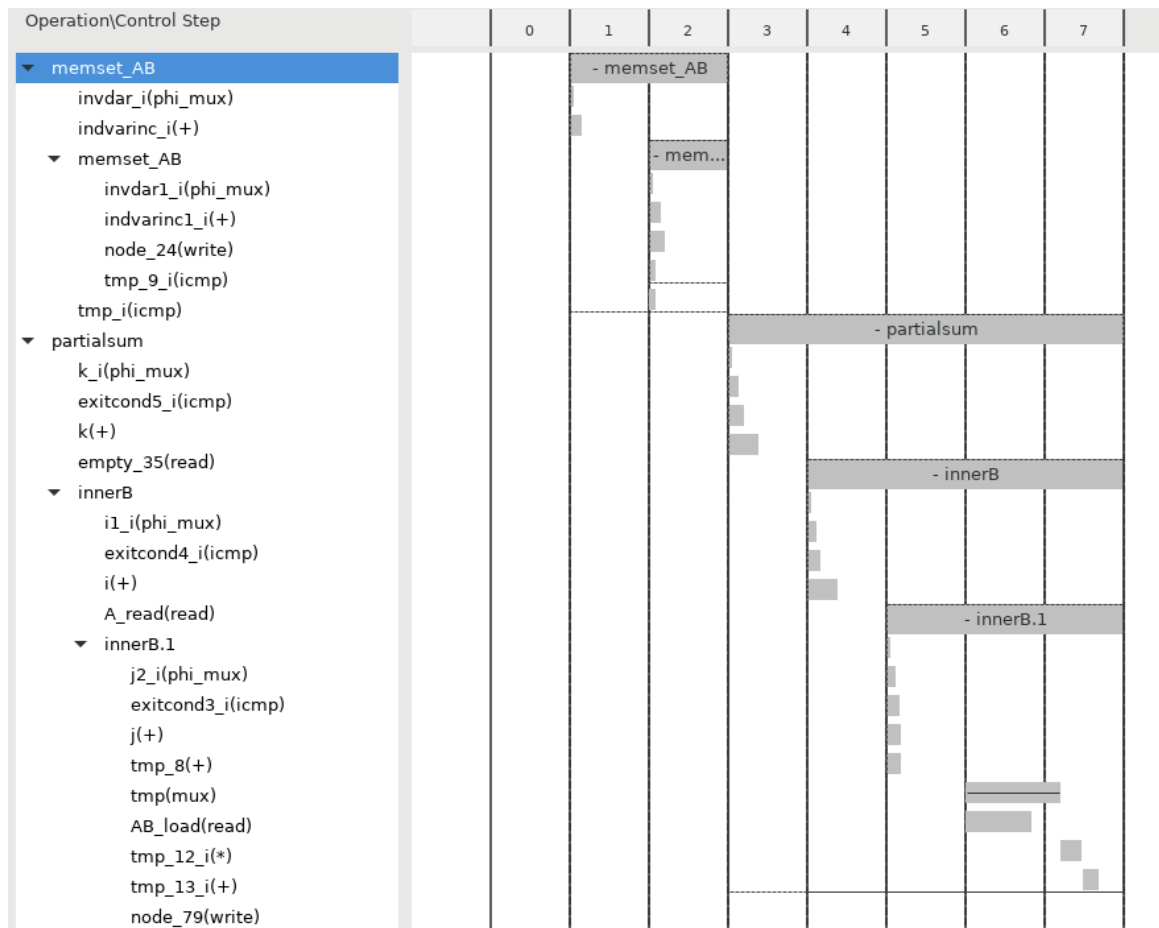


Рис. 4.15. Scheduler viewer

## 2. Block\_proc8

Performance Profile		Resource Profile			
	Pipelined	Latency	Iteration Latency	Initiation Interval	Trip count
▼ ● Block_proc8	-	1~769	-	1 ~ 769	-
▼ ● loadA	no	768	6	-	128
● loadA.1	no	4	1	-	4

Рис. 4.16. Performance profile



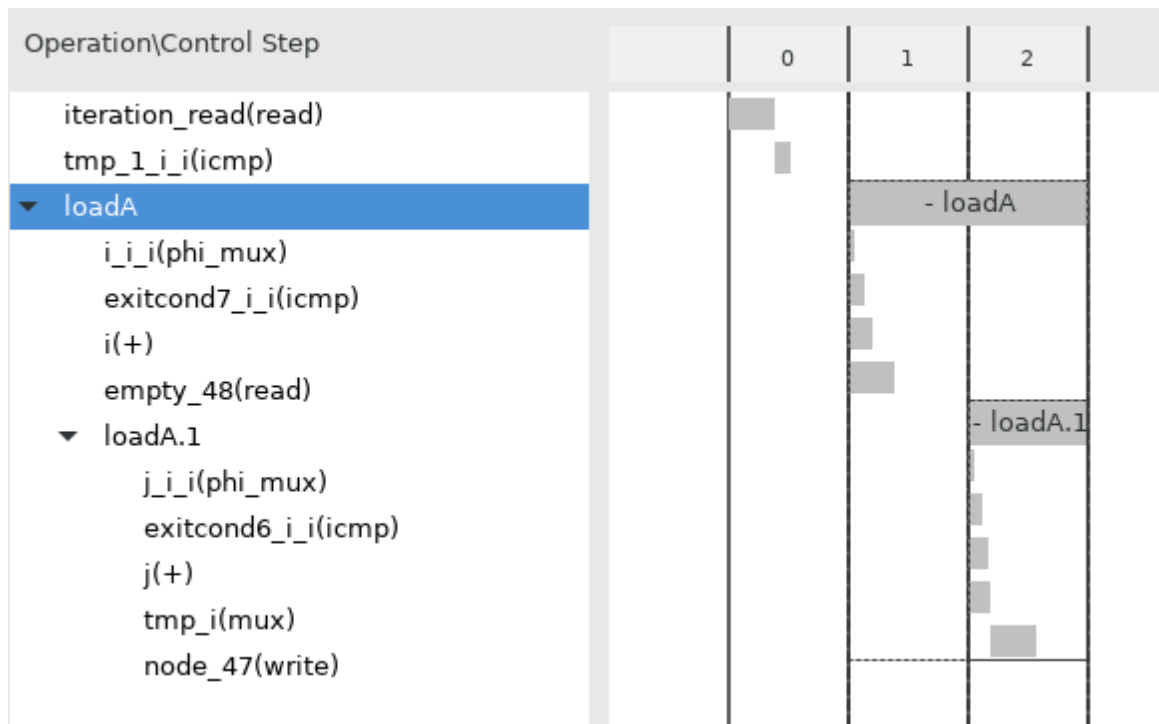


Рис. 4.17. Scheduler viewer

### 3. Loop\_writeoutput\_pro

Performance Profile		Resource Profile			
	Pipelined	Latency	Iteration Latency	Initiation Interval	Trip count
Loop_writeoutput_pro	-	41	-	41	-
writeoutput	no	40	10	-	4
writeoutput.1	no	8	2	-	4

Рис. 4.18. Performance profile

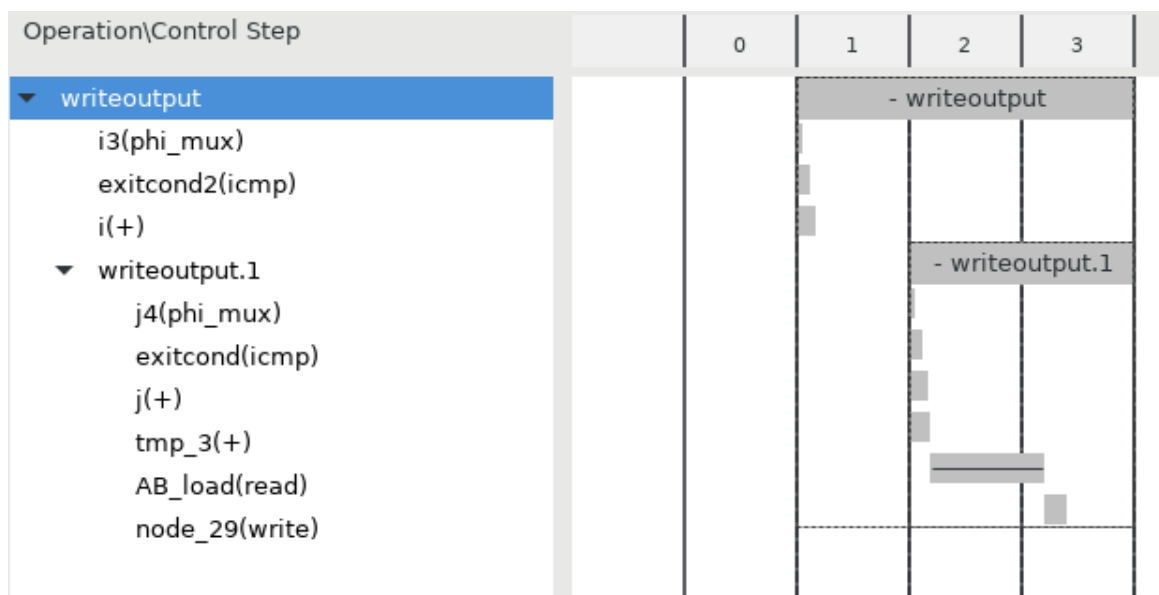
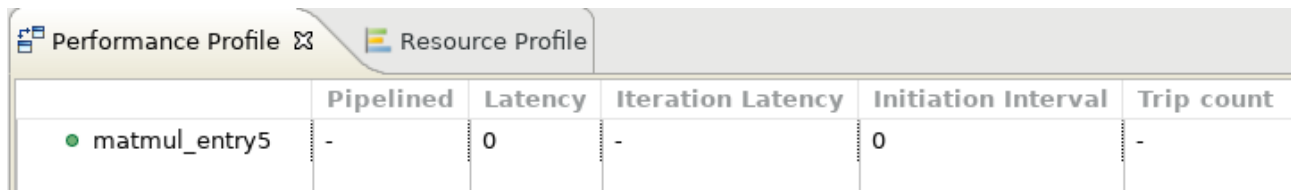


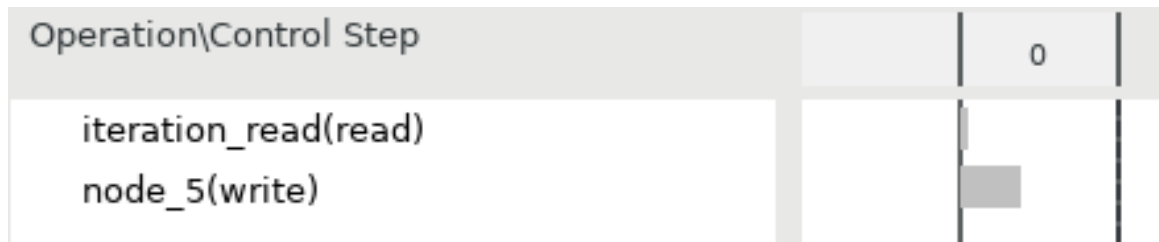
Рис. 4.19. Scheduler viewer

#### 4. matmul\_entry5



	Pipelined	Latency	Iteration Latency	Initiation Interval	Trip count
● matmul_entry5	-	0	-	0	-

Рис. 4.20. Performance profile



Operation\Control Step		0	
iteration_read(read)			
node_5(write)			

Рис. 4.21. Scheduler viewer

#### 4.4.2. Анализ решения

По данным выше видно, что:

1. Решение не расходует много ресурсов
2. Вычисления укладываются в 7487 тактов
3. Наибольшая задержка (7445) приходится на partialsum

### 4.5. Конвейеризация внутреннего цикла умножения

Проблему с задержками в различных модулях иерархии можно решить конвейеризацией вычислений.

В данном решении конвейеризируются внутренний цикл модуля partialsum и остальные модули полностью.

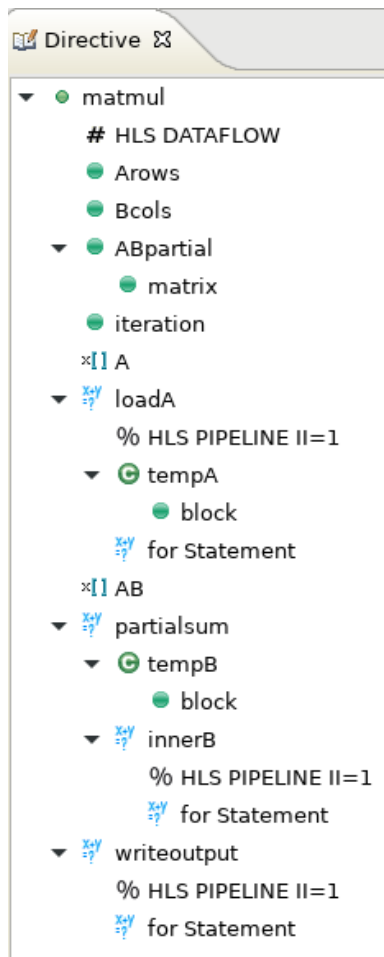


Рис. 4.22. Директивы

## Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.470	0.10

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
2851	3108	2838	2838	dataflow

Рис. 4.23. Performance estimates

Utilization Estimates				
Summary				
Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	94
FIFO	0	-	5	43
Instance	-	12	575	1180
Memory	4	-	0	0
Multiplexer	-	-	-	54
Register	-	-	9	-
Total	4	12	589	1371
Available	40	40	16000	8000
Utilization (%)	10	30	3	17

Рис. 4.24. Utilization estimates

	Negative Slack	BRAM	DSP	FF	LUT	Latency	Interval	Pipeline type
matmul	-	4	12	589	1371	2851~3108	2838	dataflow
Loop_memset_AB_proc9	-	0	12	448	667	2837	2837	none
Loop_writeoutput_pro	-	0	0	29	210	10	10	none
Block_proc8	-	0	0	96	277	1~258	1 ~ 258	none
matmul_entry6	-	0	0	2	26	0	0	none

Рис. 4.25. Module hierarchy

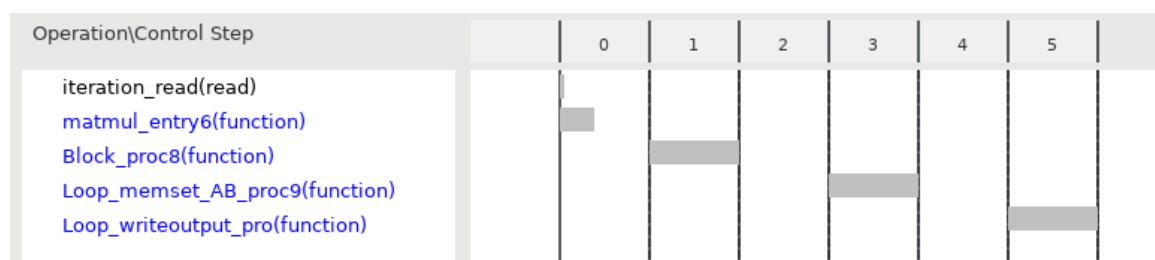


Рис. 4.26. Scheduler viewer

#### 4.5.1. Анализ модулей в иерархии

##### 1. Loop\_memset\_AB\_proc9

Performance Profile		Resource Profile			
	Pipelined	Latency	Iteration Latency	Initiation Interval	Trip count
Loop_memset_AB_proc9	-	2837	-	2837	-
memset_AB	no	19	5	-	4
partialsum	no	2816	22	-	128
innerB	yes	19	8	4	4

Рис. 4.27. Performance profile

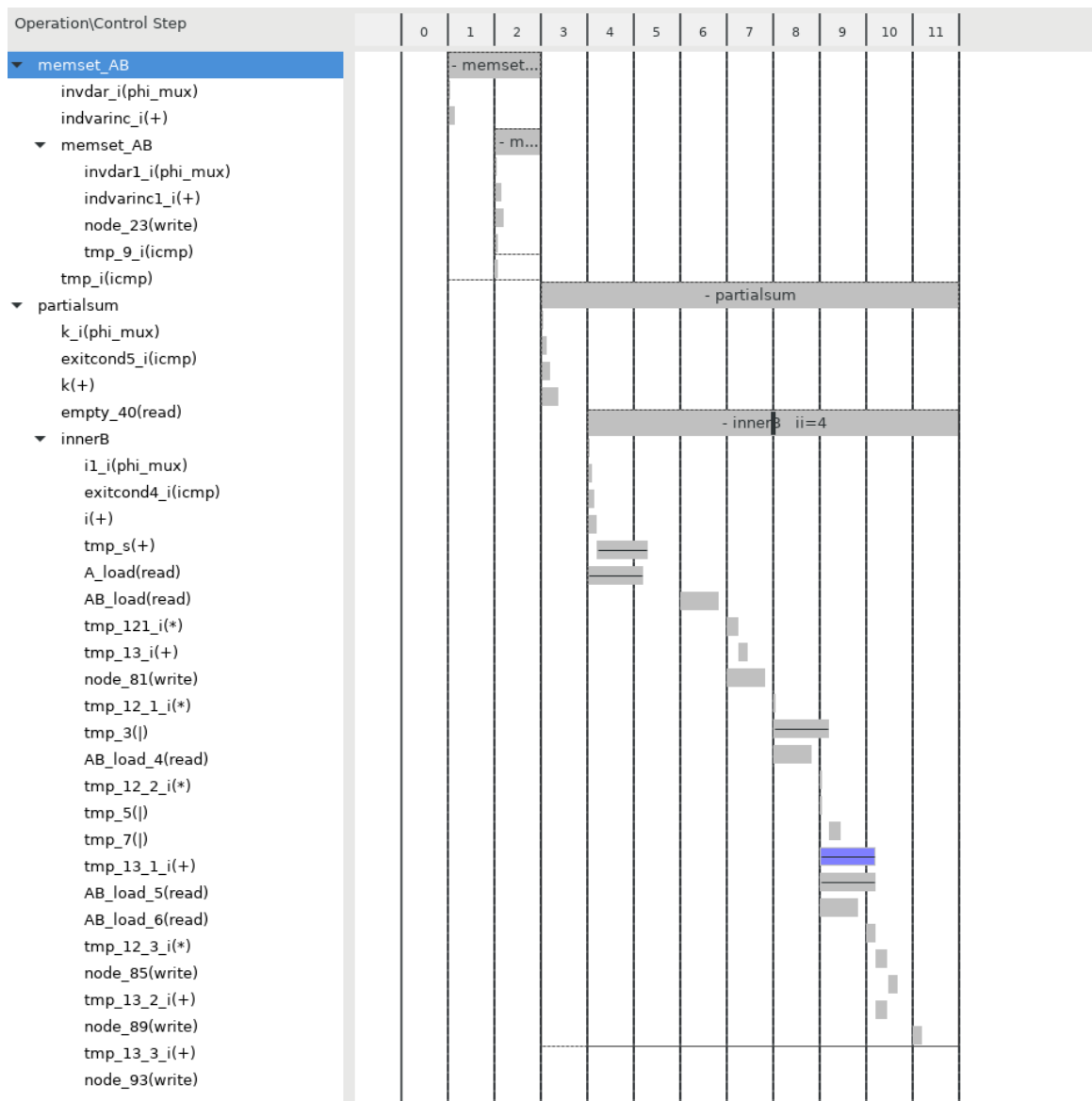


Рис. 4.28. Scheduler viewer

## 2. Block\_proc8

Performance Profile

Resource Profile

	Pipelined	Latency	Iteration Latency	Initiation Interval	Trip count
<div> <div>Loop_writeoutput_pro</div> <div> <div>writeoutput</div> </div> </div>	-	10	-	10	-
	yes	8	3	2	4

Рис. 4.29. Performance profile

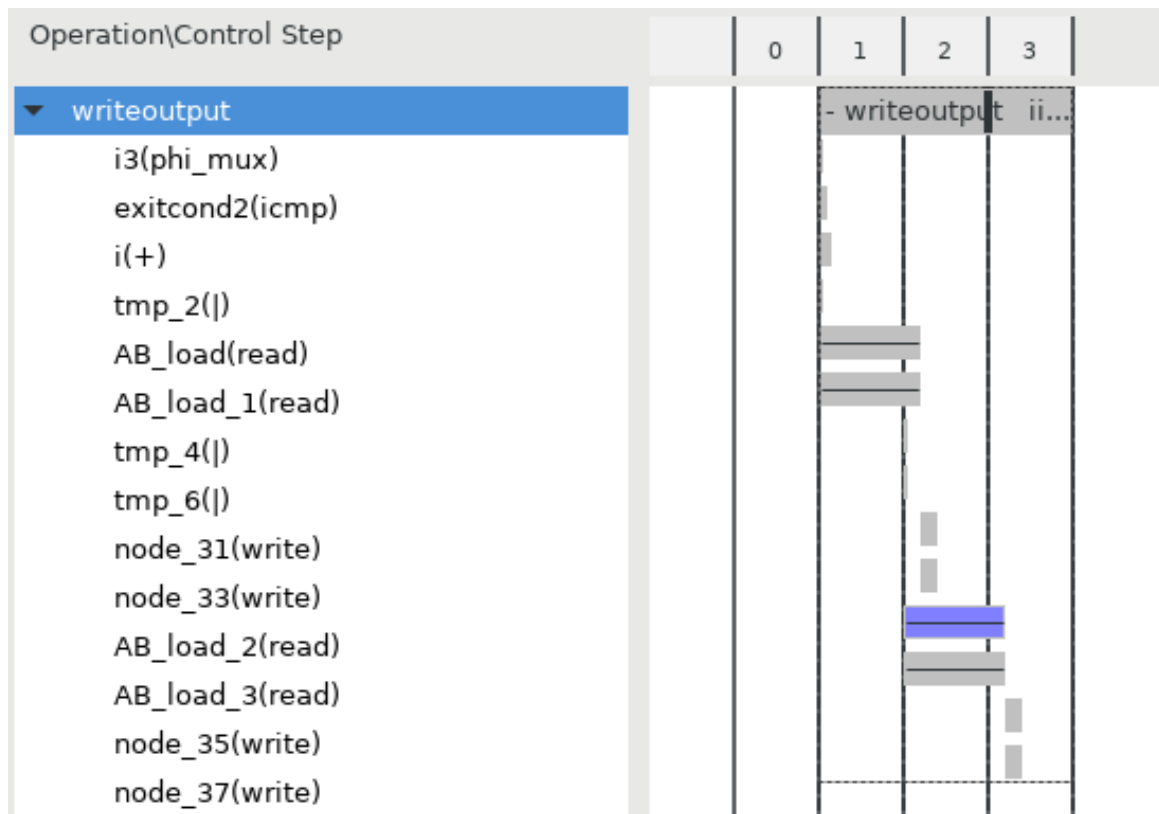


Рис. 4.30. Scheduler viewer

### 3. Loop\_writeoutput\_pro

Performance Profile		Resource Profile			
	Pipelined	Latency	Iteration Latency	Initiation Interval	Trip count
Block_proc8	-	1~258	-	1 ~ 258	-
loadA	yes	256	3	2	128

Рис. 4.31. Performance profile



Рис. 4.32. Scheduler viewer

4. matmul\_entry5

Performance Profile		Resource Profile			
	Pipelined	Latency	Iteration Latency	Initiation Interval	Trip count
● matmul_entry6	-	0	-	0	-

Рис. 4.33. Performance profile

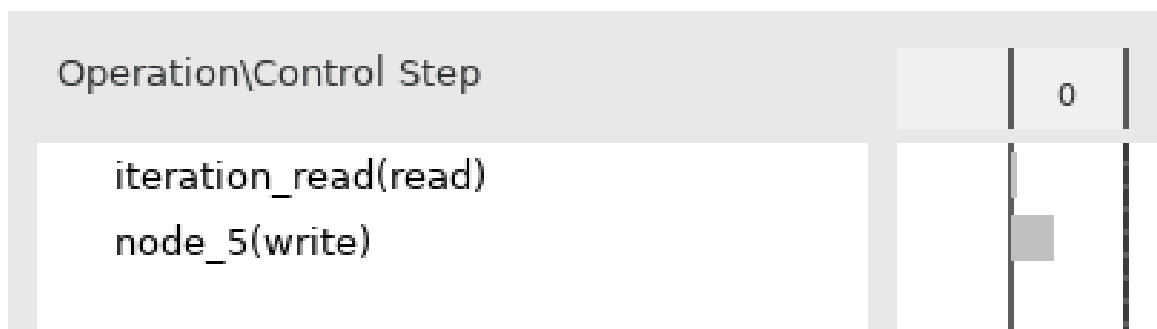


Рис. 4.34. Scheduler viewer

#### 4.5.2. Анализ решения

По данным выше видно, что:

1. Решение не расходует много ресурсов
2. Вычисления укладываются в 3108 тактов
3. Наибольшая задержка (2837) всё ещё приходится на partialsum, но в сравнении с предыдущим решением её удалось сократить

4. Также удалось сократить задержки в остальных модулях, но это не влияет на общую задержку устройства, так как она определяется наибольшей задержкой среди модулей

## 4.6. Конвейеризация внешнего цикла умножения

В данном решении, применяется полная конвейеризация вычислений в модуле partialsum.

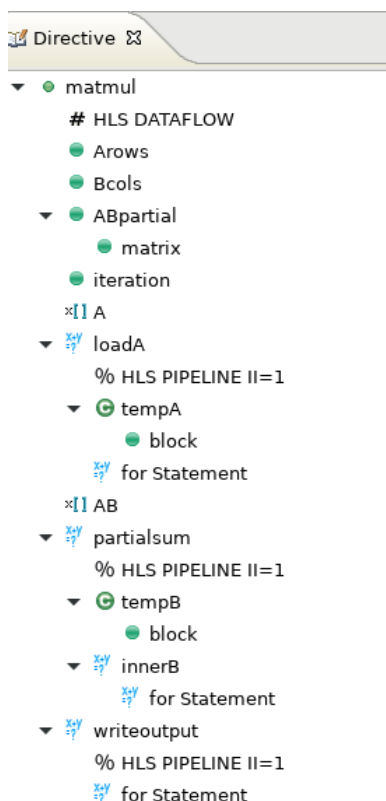


Рис. 4.35. Директивы

## Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.470	0.10

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
2081	2081	2071	2071	dataflow

Рис. 4.36. Performance estimates



Utilization Estimates				
Summary				
Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	94
FIFO	8	-	229	295
Instance	-	48	1497	1921
Memory	2	-	0	0
Multiplexer	-	-	-	54
Register	-	-	9	-
Total	10	48	1735	2364
Available	40	40	16000	8000
Utilization (%)	25	120	10	29

Рис. 4.37. Utilization estimates

	Negative Slack	BRAM	DSP	FF	LUT	Latency	Interval	Pipeline type
matmul	-	10	48	1735	2364	2081	2071	dataflow
Loop_memset_AB_proc9	-	0	48	1451	1463	2070	2070	none
Loop_writeoutput_pro	-	0	0	29	210	10	10	none
Block_proc8	-	0	0	15	222	1~130	1 ~ 130	none
matmul_entry5	-	0	0	2	26	0	0	none

Рис. 4.38. Module hierarchy

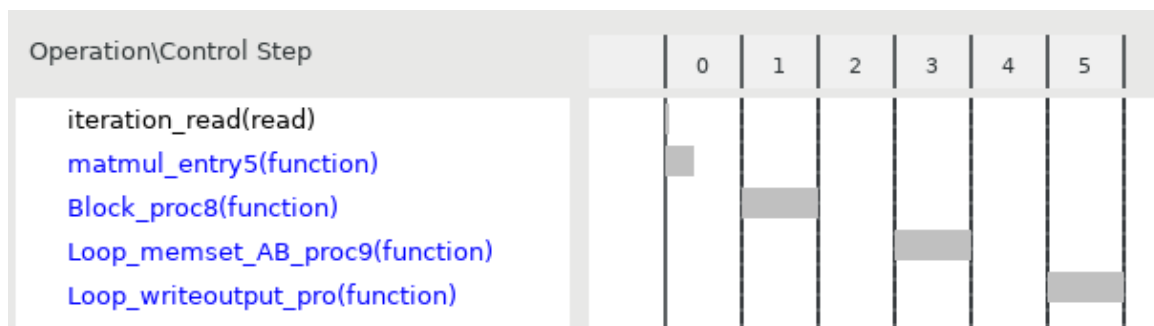


Рис. 4.39. Scheduler viewer

#### 4.6.1. Анализ модулей в иерархии

##### 1. Loop\_memset\_AB\_proc9

Performance Profile		Resource Profile			
	Pipelined	Latency	Iteration Latency	Initiation Interval	Trip count
Loop_memset_AB_proc9	-	2070	-	2070	-
memset_AB	no	19	5	-	4
partialsum	yes	2048	16	16	128

Рис. 4.40. Performance profile

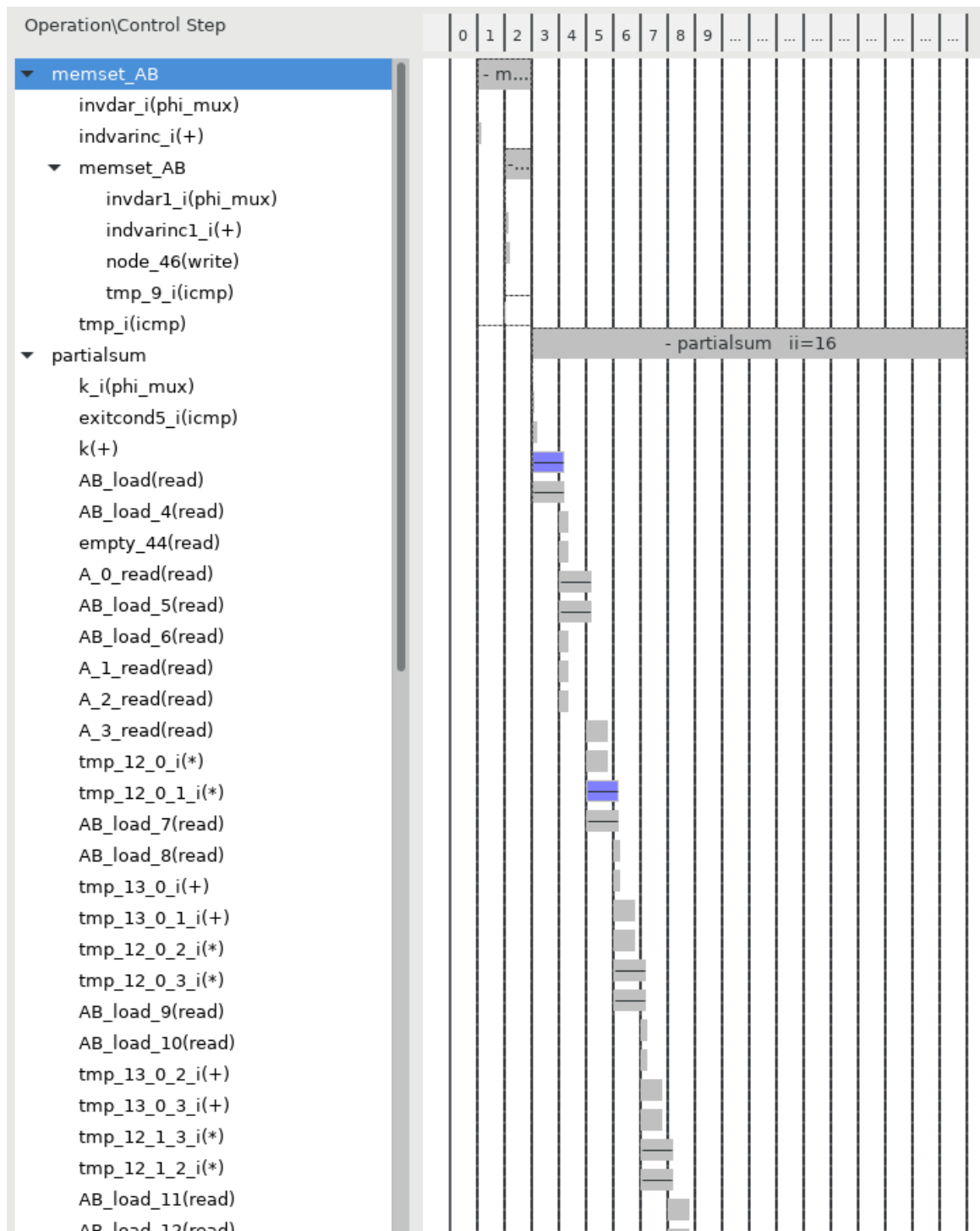


Рис. 4.41. Scheduler view

## 2. Block\_proc8

Performance Profile

Resource Profile

	Pipelined	Latency	Iteration Latency	Initiation Interval	Trip count
<div> <div>Loop_writeoutput_pro</div> <div>writeoutput</div> </div>	-	10	-	10	-
	yes	8	3	2	4

Рис. 4.42. Performance profile

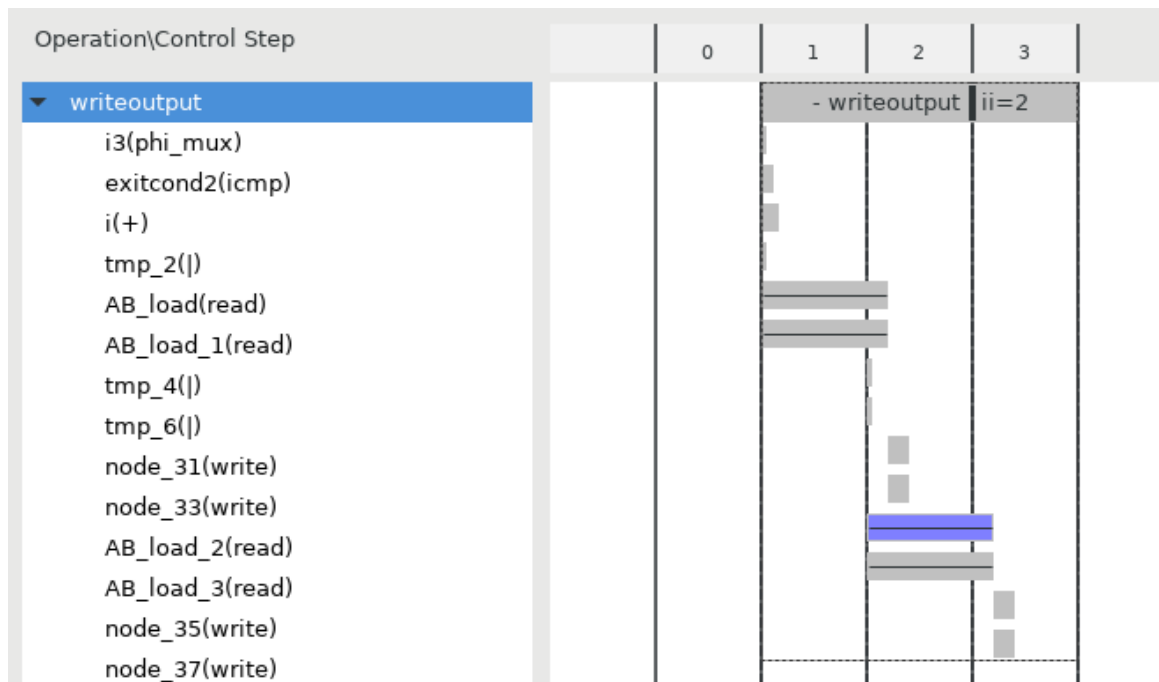


Рис. 4.43. Scheduler view

### 3. Loop\_writeoutput\_pro

Performance Profile		Resource Profile			
	Pipelined	Latency	Iteration Latency	Initiation Interval	Trip count
Block_proc8	-	1~130	-	1 ~ 130	-
loadA	yes	128	2	1	128

Рис. 4.44. Performance profile

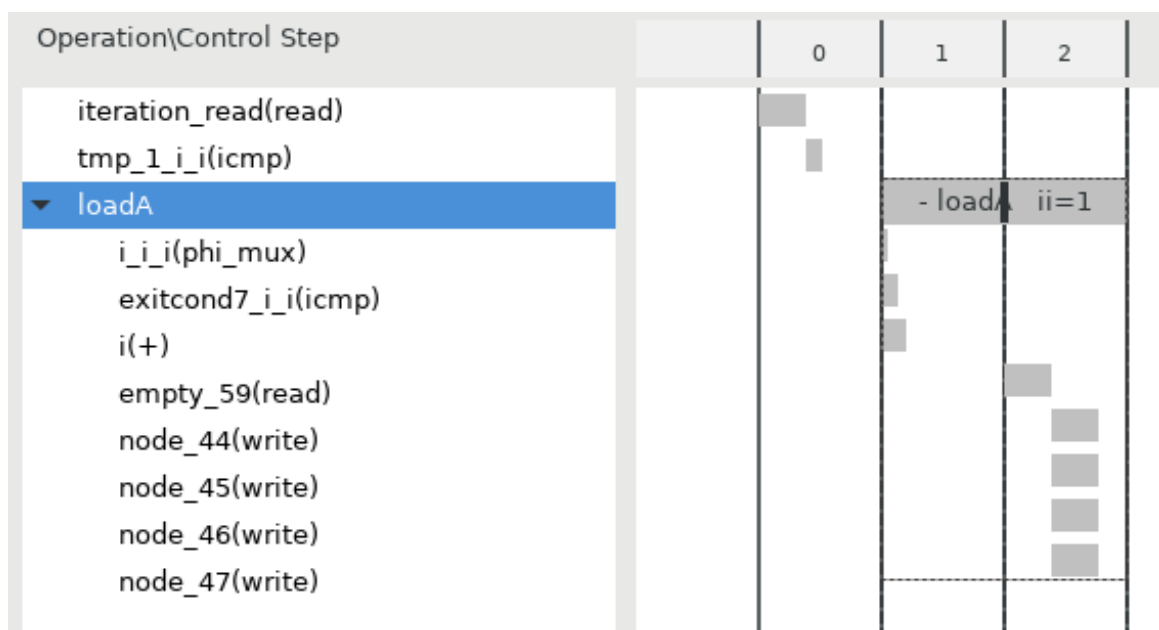
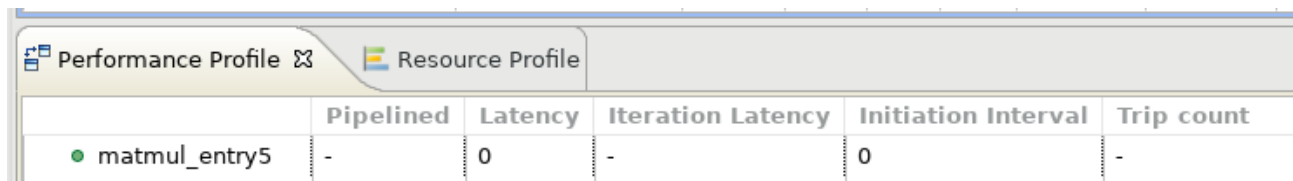


Рис. 4.45. Scheduler view

#### 4. matmul\_entry5



	Pipelined	Latency	Iteration Latency	Initiation Interval	Trip count
● matmul_entry5	-	0	-	0	-

Рис. 4.46. Performance profile

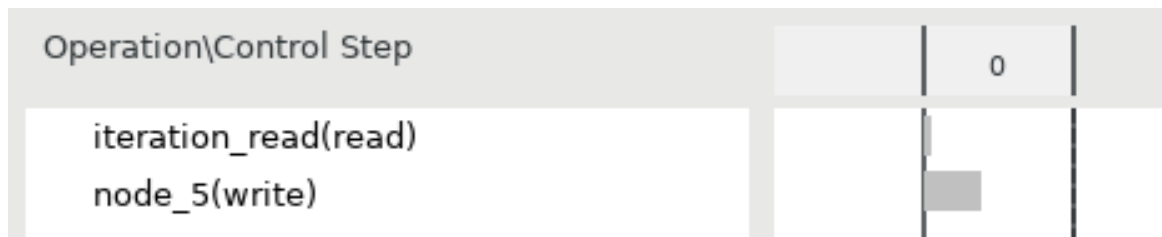


Рис. 4.47. Scheduler viewer

#### 4.6.2. Анализ решения

По данным выше видно, что:

1. Решение начало расходовать слишком много ресурсов, и не может быть реализовано на заданной микросхеме
2. Вычисления укладываются в 2081 такт
3. Наибольшая задержка (2070) всё ещё приходится на partialsum, но в сравнении с предыдущим решением её удалось сократить

#### 4.7. Сравнение решений

Ниже приведено сравнение созданных решений.

Performance Estimates				
▣ Timing (ns)				
Clock		no_flags	pipeline_all	pipeline_inner
ap_clk	Target	10.00	10.00	10.00
	Estimated	8.470	8.470	8.470
▣ Latency (clock cycles)				
		no_flags	pipeline_all	pipeline_inner
Latency	min	7487	2081	2851
	max	7487	2081	3108
Interval	min	7446	2071	2838
	max	7446	2071	2838

Рис. 4.48. Performance estimates

Utilization Estimates			
	no_flags	pipeline_all	pipeline_inner
BRAM_18K2	2	10	4
DSP48E	3	48	12
FF	594	1735	589
LUT	1055	2364	1371

Рис. 4.49. Utilization estimates

По сравнению решений можно сделать вывод, что оптимальным вариантом оптимизации является конвейеризация внутреннего цикла умножения.

## 5. Вывод

Блочное умножение матриц предоставляет другой способ для вычисления матричного произведения. Алгоритм вычисляет частичные результаты матрицы результатов, передавая поднабор входных матриц в функцию. Затем эта функция вычисляется несколько раз, чтобы завершить вычисление умножения всей матрицы.

Вычисление произведение блоков матриц менее ресурсозатратно и имеет большую скорость вычислений, в сравнении с обычным умножением матриц, но данное сравнение некорректно. В отличие от обычного умножения, блочное вычисляет лишь часть результирующего значения, а формирование полного результата требует многократного вычисления частичного результата.

Таким образом, если учесть количество вызовов функции блочного умножения, обычное умножение оказывается более быстрым.

Блочное умножение имеет смысл применять в следующих ситуациях:

1. Исходные матрицы не могут быть полностью размещены в памяти устройства
2. Исходные данные поступают постепенно, и можно производить вычисления по мере поступления
3. Вычисления отдельных блоков можно распараллелить и за счёт этого получить ускорение вычислений

## Список литературы

- [1] Kastner R., Matai J., Neuendorffer S. Parallel Programming for FPGAs // ArXiv e-prints. — 2018. — May. — 1805.03648.