Cse237C project4

Yiming Ren
A59005465
Y5ren@ucsd.edu

Baseline:

Compile Estimated Kernel Resource Utilization Summary

Name	Source Location	ALM	ALUT	REG	MLAB	RAM	DSP
a_init			2082	4889	49	0	5
b_init			2476	5269	50	0	6
c_calc			9209	14396	71	91	16
Global Interconnect			2880	3192	0	61	0
System description ROM			0	67	0	2	0
Compile Estimated: Kernel System			16647	27813	170	154	27

In c-calc loop:

Latency is 224, LSU style is Burst-coalesced cached.

Instruction	Load
Width	32 bits
LSU Style	Burst-coalesced cached
Stall-free	No
Global Memory	Yes
Start Cycle	11
Latency	224

Question 1:

DPC++ obtains a few LSU styles which read and write memories in different ways. By default, DPC++ uses Burst/coalesced which buffers contiguous memory requests until it reaches the maximum burst size. Changing the LSU style can change the latency.

First, by adding the title

```
using PrefetchingLSU =
INTEL::lsu<INTEL::prefetch<true>,
INTEL::statically_coalesce<false>>;
```

and pointers referring to the input arrays, I can change the LSU style to prefetching.

Kernel code:

```
h.template parallel_for<c_calc>(range(M, P), [=](auto index)[[intel::kernel_args_restrict]] {

  // Get global position in Y direction.
  int row = index[0];
  // Get global position in X direction.
  int col = index[1];

  float sum = 0.0f;
  auto input_a = a.get_pointer();
  auto input_b = b.get_pointer();

  // Compute the result of one element of c
  for (int i = 0; i < width_a; i++) {
    sum += PrefetchingLSU::load(input_a+i) * PrefetchingLSU::load(input_b+P*i);
  }

  c[index] = sum;
});
```

Performance (LD in c_calc.B2):

Instruction	Load
Width	32 bits
LSU Style	Prefetching
Stall-free	No
Global Memory	Yes
Start Cycle	11
Latency	3

The prefetching LSU is much more efficient compared to the burst-coalesced LSU style as regard to the latency. The reason is pretty straightforward: burst-coalesced LSU style transfers more data in a loop cycle. However, one work item only needs to calculate once in one iteration, which means it only needs 2 sizeof(float)s data to compute the multiplication. The remained data is useless and of course brings a lot of waste of time. On the other hand, prefetching LSU style only read/write 1 sizeof(float) data to the memory in a iteration, hugely reducing the time cycle needed for load/store. Besides, additional pointer computation is needed because we need to an additional ‘plus’ to get the pointer every iteration, but the cycle of a pointer computation is only one, which is ignorable compared to the 200+ cycles LD.

Next, the pipelined LSU could be applied by switching ‘PrefetchingLSU::load’ to

'PipelinedLSU:load' and adding the header

```
using PipelinedLSU = INTEL::lsu<>;
```

Performance (LD in c_calc.B2):

Width	32 bits
Type	Pipelined
Stall-free	No
Start Cycle	11
Latency	7

Pipelined LSU saves the data from global memory to local memory first.

Therefore, the size of the requests determines the latency. Here the bundle of data is obviously larger than sizeof(float) since the latency is larger than the one of prefetching LSU, but its still faster than the default burst-coalesced LSU.

Comparation of prefetching LSU and pipelined LSU:

Name	Source Location	Pipelined	Block Scheduled II	Block Scheduled fMAX	Latency	Speculated Iterations	Max Interleaving Iterations	Brief Info
c_calc.B2	matrix_mul_dpcpp.cpp:138	No	n/a	240.00	26.000000	n/a	n/a	Thread capacity = 27

*Pipelined LSU, Throughput = 1/(1/(240*10^6)*26) = 9.23MHz*

Name	Source Location	Pipelined	Block Scheduled II	Block Scheduled fMAX	Latency	Speculated Iterations	Max Interleaving Iterations	Brief Info
c_calc.B2	matrix_mul_dpcpp.cpp:138	No	n/a	240.00	21.000000	n/a	n/a	Thread capacity = 21

*Prefetching LSU, Throughput = 1/(1/(240*10^6)*21) = 11.4MHz*

Considering the loop latency of prefetching LSU is less, the prefetching LSU would be used in further optimization.

Question 2:

Loop unrolling could be achieved by put a pragma over the for loop:

```
#pragma unroll unroll_factor
```

with the 'unroll_factor' varying from 2 to 8.

For this problem, I changed the m_size to 150 * 8 * 8, so that m is 1200, n is 2400, and p is 4800.

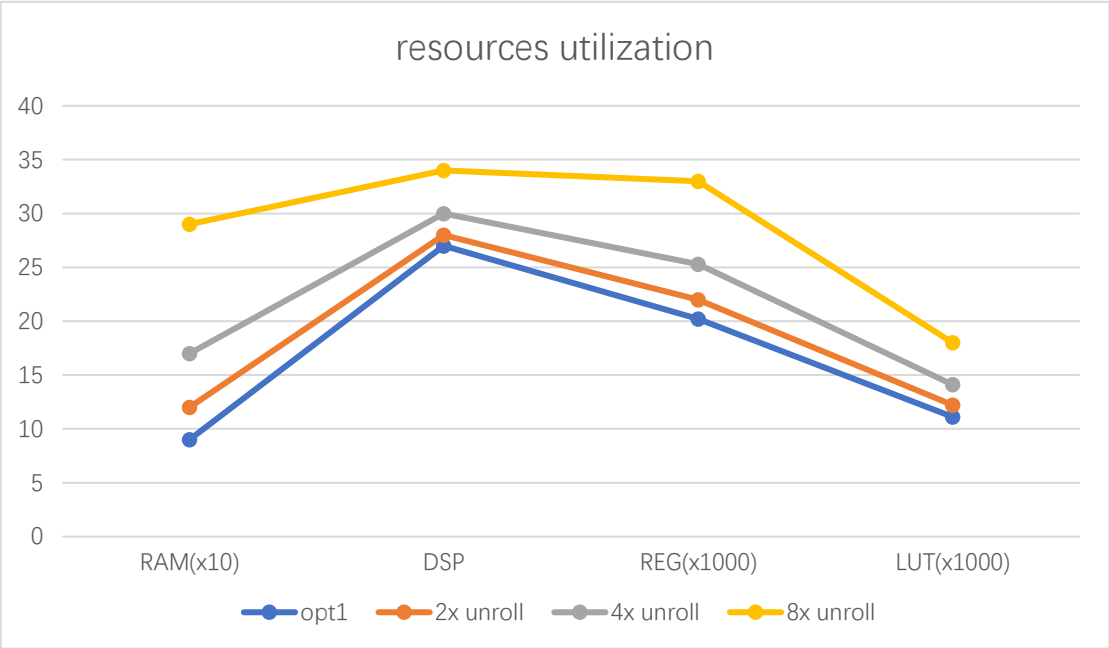
Performance and resources of optimization 1 with matrixes in such sizes:

Name	Source Location	Pipelined	Block Scheduled II	Block Scheduled fMAX	Latency
c_calc.B2	matrix_mul_dpcpp.cpp:139	No	n/a	240.00	21.000000

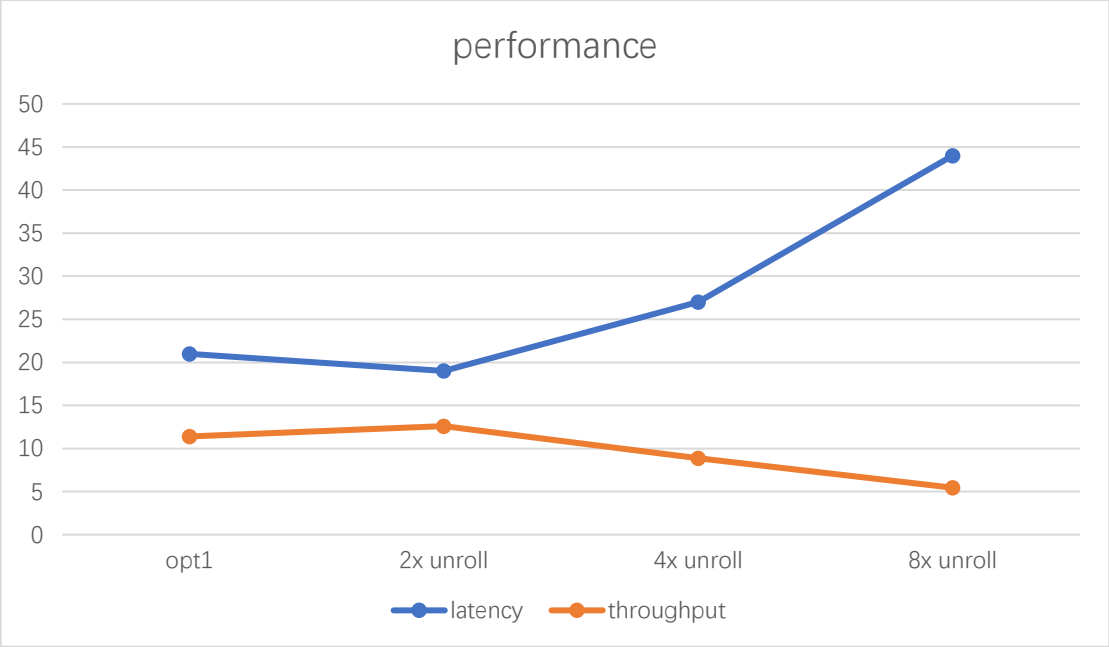
Compile Estimated Kernel Resource Utilization Summary

Name	Source Location	ALM	ALUT	REG	MLAB	RAM	DSP
a_init			2082	4889	49	0	5
b_init			2476	5269	50	0	6
c_calc			3709	6603	77	26	16
Global Interconnect			2880	3192	0	61	0
System description ROM			0	67	0	2	0
Compile Estimated: Kernel System			11147	20020	176	89	27

Resources and performance diagram:



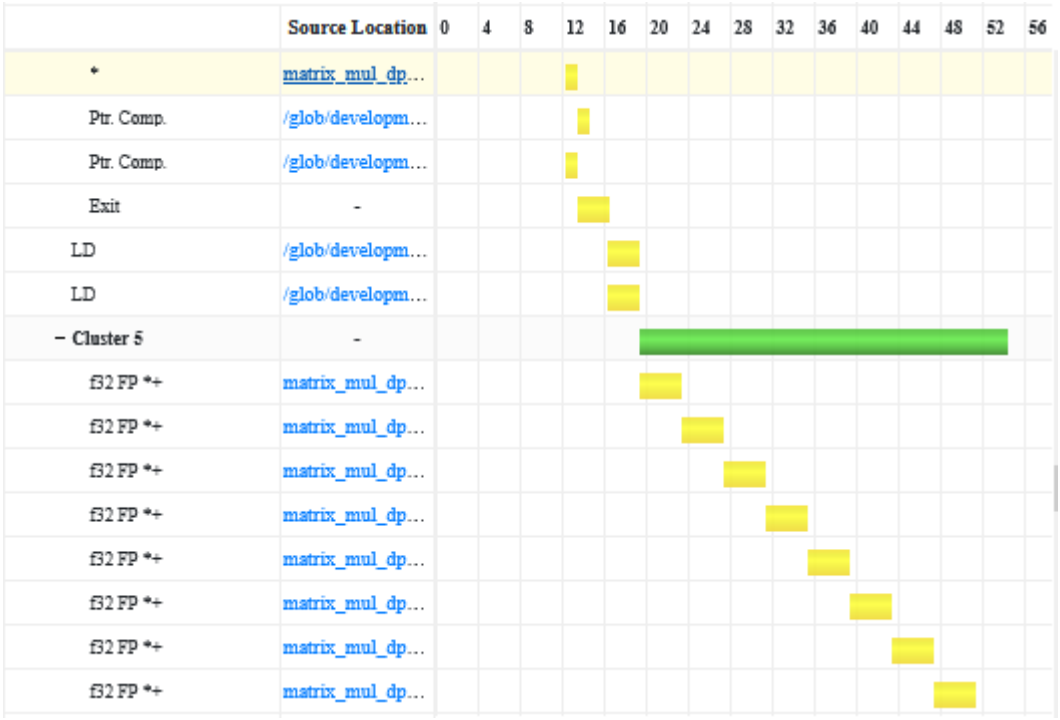
Note: the values in above diagram is balanced to scaling.



From the above diagrams, the resources used without loop unrolling are the least, but loop unrolling with factor as 2 has a better performance than no unrolling. The trends of both performance and resources are decreasing linearly as the factor increasing.

The reason behind that the large factor unrolling is not preferred could be the

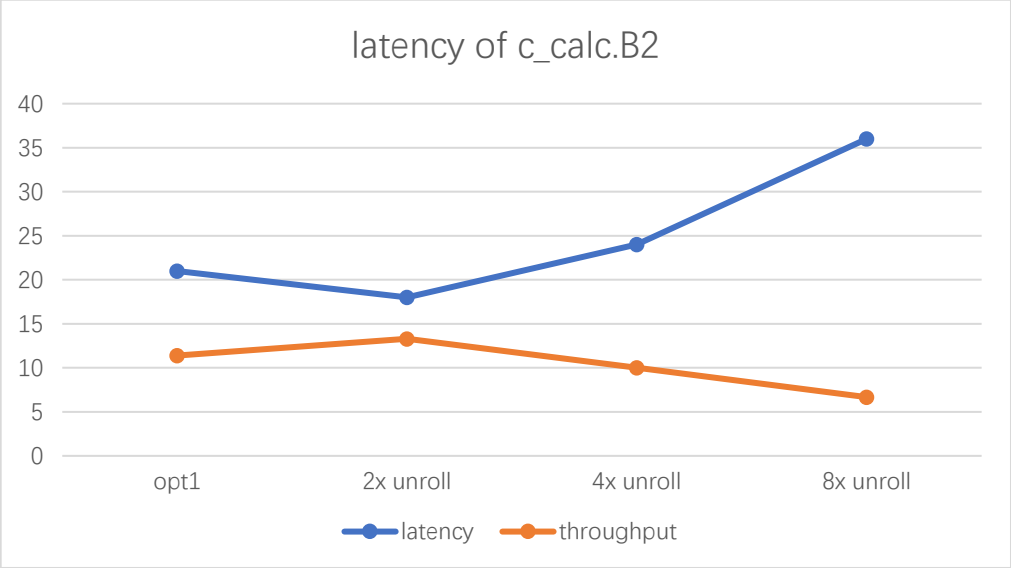
amount of data dependencies. For example, take a look at the schedule of 8x unrolling:



Before Cluster 5, work items were fetching data from memory. They go very fast because highly unrolled loop and prefetching LSU allow them to fetch multiple data at one time. However, during the arithmetic operations, they should read/write to cache continuously. Work items should wait until the previous calculation finish to start the next one because they need the results written to the cache to start. Consequently, 8 calculation in one iteration line up, but not parallel, and we get a long calculation period, which is cluster 5 in the image. While in 2x loop unrolling, the waste of time for waiting is not that much, and the program could take advantage of multiple fetching and doubling the number of arithmetic operations. Considering the tradeoff between performance and resources, 2x loop unrolling is a good choice for optimization.

Question 3:

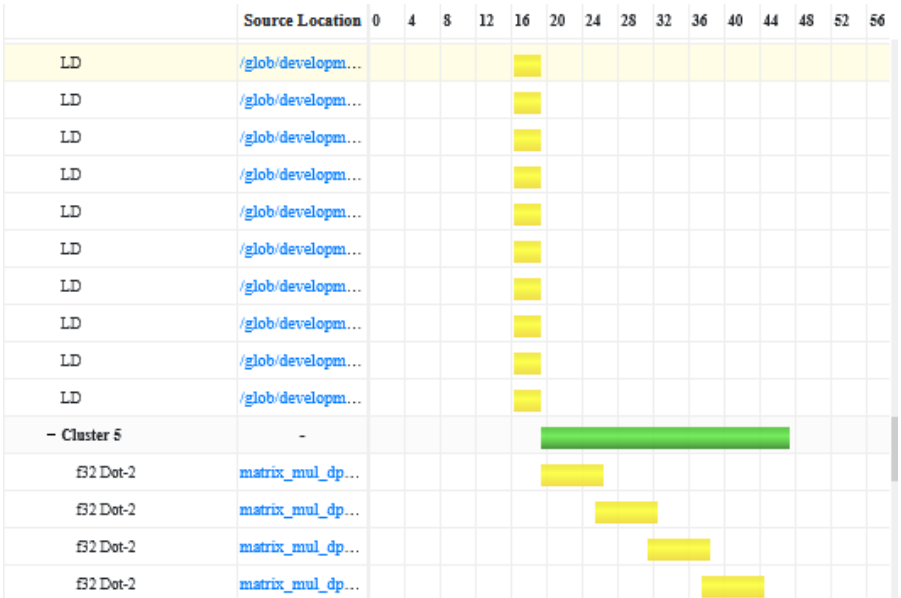
Latency diagram by unrolling loop manually:



The trend of performance of manual loop unrolling with factor varying from 2 to 8 is also decreasing linearly as the factor increasing as expected. Like I stated in the last question, 4x and 8x loop unrolling over parallel the loop so that the duration of all arithmetic operations is too long.

But compared to the pragma, manual loop unrolling has a better performance.

To explain this, also take a look at the schedule of 8x unrolling:



Notice that the instructions under Cluster 5 become ‘f32 Dot-2’. The difference is that ‘f32 Dot-2’ does calculation on 32-bit floating-point of size 2:

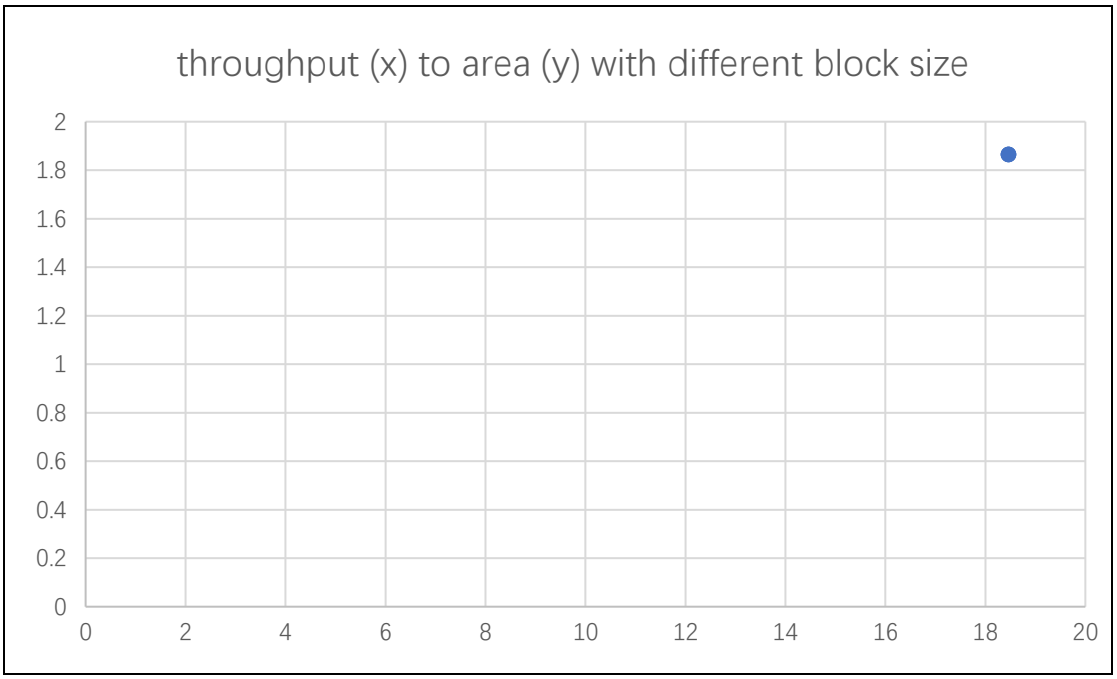
f32 Dot-2:

Instruction	32-bit Floating-point Dot Product of Size 2
Start Cycle	8
Latency	7

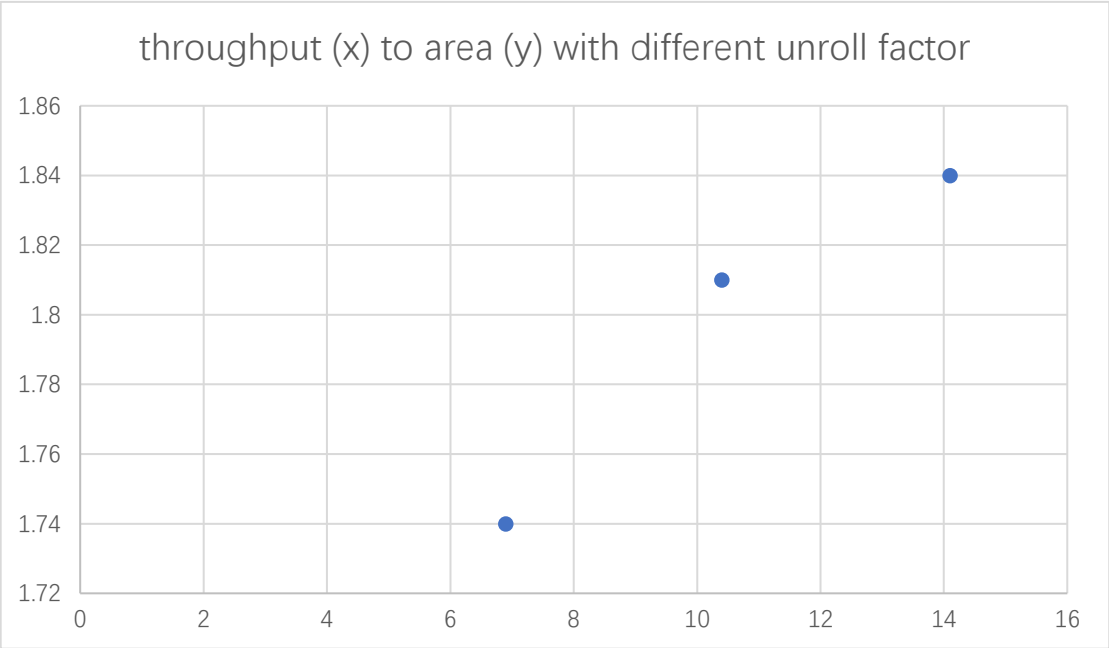
which means that 2 matrix multiplication operations could be done at the same time, and thus we spend half time than pragma doing arithmetic operations. We hence choose the manual loop unrolling as a better optimization.

Question 4:

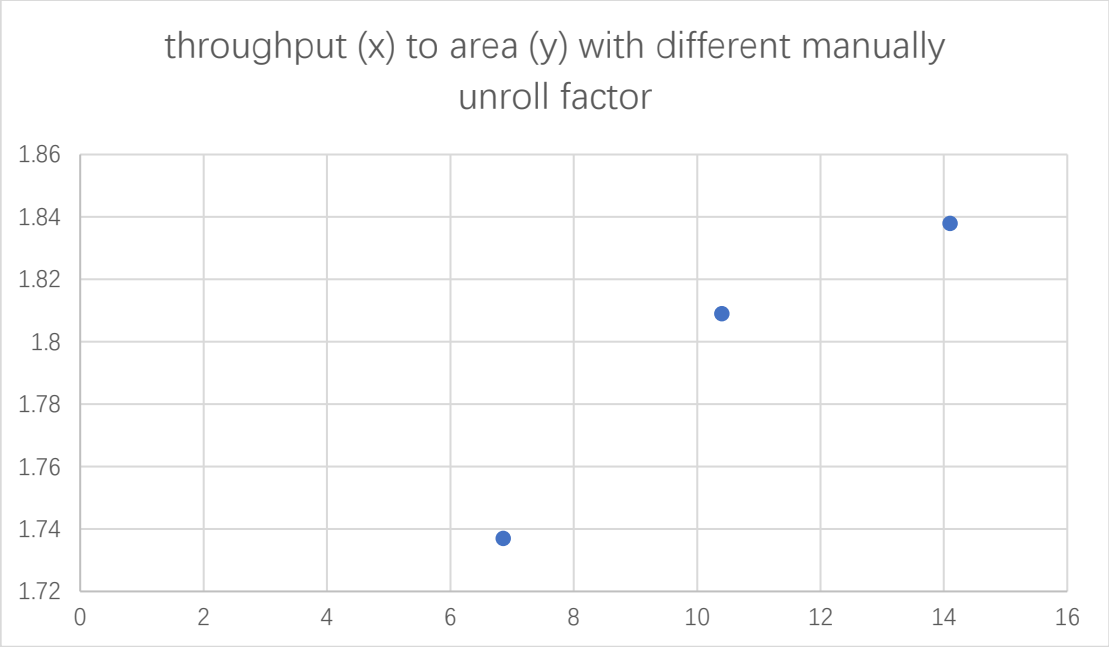
In this question, I will illustrate how each knob tunes the design by diagrams. The knobs are block size, unrolled factor, manually unrolled factor, and matrix size;



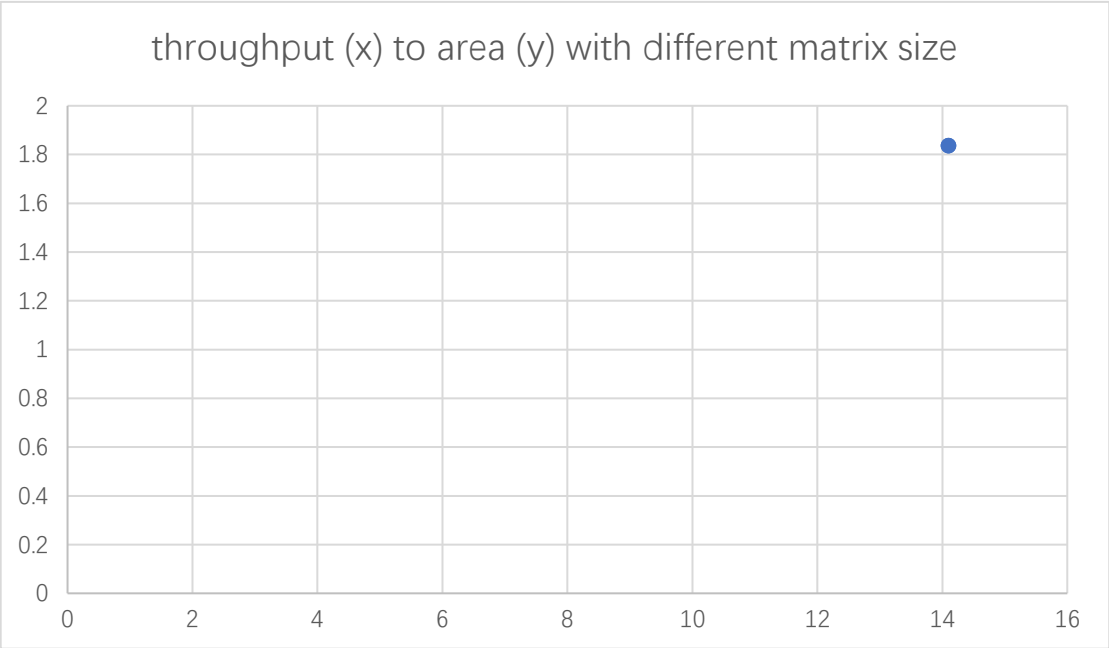
In the above diagram, there are 4 points representing block size as 4, 8, 16, and 32. Actually 4 points are in the same position, so block size has little influence on the performance and area.



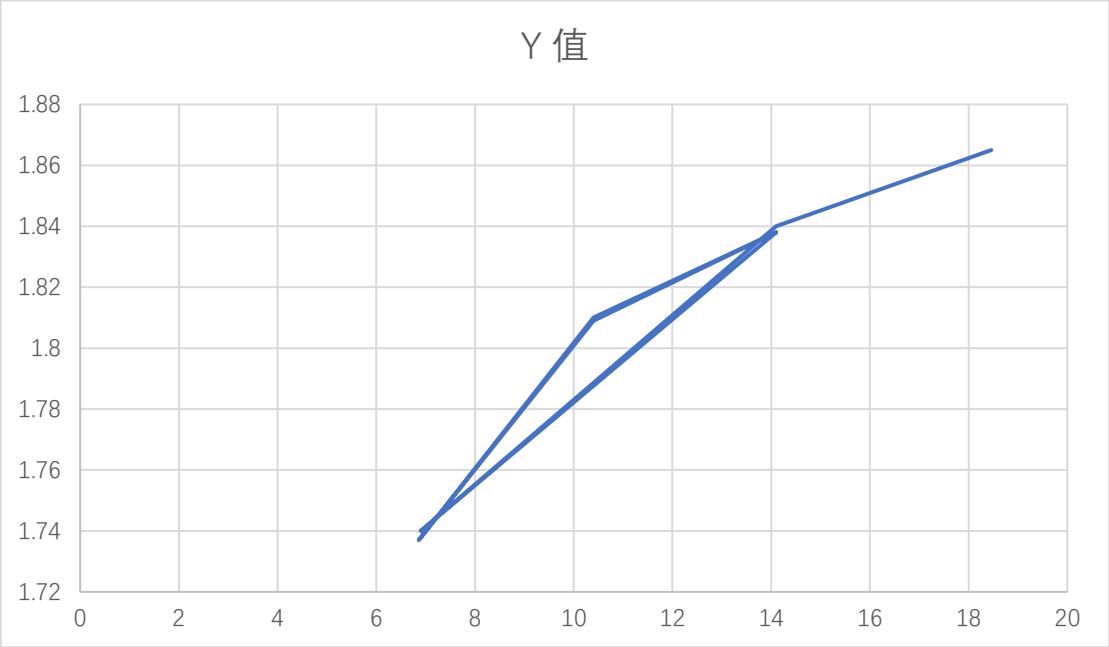
Less unroll factor has poorer resources utilization but better performance.



Less unroll factor has poorer resources utilization but better performance. Note that unroll factor 4 has the worst tradeoff between performance and area.



There are actually 3 points in above diagram. Each represents one of the designs with 4 times, 16 times, and 64 times original matrix size. Now we see that matrix size has little impact on performance and area.



Summary: block and matrix size have nothing to do with both performance and area. Considering the tradeoff, unrolling factor as 4 is always not a good choice. Based on that, we can use a less unrolling factor for performance and larger

unrolling factor if we pursue area. Also, manually loop unroll is recommended.