CSE237C: Lab 4 Matrix Multiplication on Intel DevCloud Using DPC++

Weihong Xu
wexu@ucsd.edu

November 18, 2020

1 Introduction

This report introduces matrix multiplication using DPC++ on Intel DevCloud platform. After making proper optimizations, the design space exploration is conducted to determine the best block matrix multiplication architecture.

2 Optimize Load Transfers

2.1 Implementation

The default load store unit (LSU) in the baseline implementation normally requires hundreds of cycles to fetch the data, becoming the major bottleneck in the c_calc kernel. To reduce the latency caused by IO, I utilize the *PrefetchingLSU* and *BurstCoalescedLSU* primitives to perform coalescing continuous memory access.

Folder **mm_optimized1** contains the optimized code. The code implementation is given as follows:

```
h.template parallel_for<c_calc>(range(M, P), [=](auto index) {
    // Get global position in Y direction.
    int row = index[0];
    // Get global position in X direction.
    int col = index[1];

    auto ptr_a = a.get_pointer()+row*width_a;
    auto ptr_b = b.get_pointer()+col;

    float sum = 0.0f;
    int add_offset = 0;
    // Compute the result of one element of c
    for (int i = 0; i < width_a; i++)
    {
        float a_temp = PrefetchingLSU::load(ptr_a+i);
        float b_temp = PrefetchingLSU::load(ptr_b+add_offset);
        sum += a_temp * b_temp;

        add_offset += width_b;
    }

    auto ptr_c = c.get_pointer();
    BurstCoalescedLSU::store(ptr_c + row*width_c + col, sum);
});
```

The functionality of the associated implementation is verified using the testbench. The synthesis results of **Opt. 1** and the baseline designs are shown as follows:

Table 1: Synthesis results and comparison.

Design	Est. Freq. (MHz)	c_calc Latency	Utilization					Speedup
			ALUT	REG	MLAB	RAM	DSP	
Baseline	240	242	16,647	27,875	172	154	27	1×
Opt. 1	240	21	11,173	20,213	201	89	30	11.5×

2.2 Question 1

- *Describe your modification and discuss why it achieves a lower latency.*

  I take the advantages of *PrefetchingLSU* and *BurstCoalescedLSU* primitives to perform coalescing continuous memory access. The clock cycles required to read the data are significantly reduced from 242 to 21 cycles. This is because the continuous memory read/write maximizes the use of available global memory bandwidth and alleviate the bottleneck of baseline design. Also, the computation-to-communication ratio is increased, which achieves a speedup of 11×.

3 Loop Unrolling

In this section, the matrix multiplication with size $N, M, P = 256$ is considered.

3.1 Using Unrolling Pragma

The loop unrolling pragmas with different factors are used to unroll the inner c_calc loop. Three unrolling factors are considered, including 2, 4, and 8. Folder **mm_optimized2** includes the code. The synthesis results are summarized in Table 2.

3.2 Question 2

- *What are the effects and general trends of performing unrolling using the pragma? Are the results as expected?*

Table 2: Synthesis results and comparison.

Design	Est. Freq. (MHz)	c_calc Latency	Utilization					Speedup
			ALUT	REG	MLAB	RAM	DSP	
$N = 150, M = 300, P = 600$								
Baseline	240	242	$16,647$	$27,875$	172	154	27	1×
Opt. 1	240	21	$11,173$	$20,213$	201	89	30	11.5×
$N, M, P = 256$								
Opt. 2, Baseline	240	242	$16,647$	$27,875$	172	154	27	1×
Opt. 2, factor = 2	240	15	$11,942$	$21,762$	193	115	32.5	16.1×
Opt. 2, factor = 4	240	23	$13,635$	$24,995$	209	167	34.5	10.5×
Opt. 2, factor = 8	240	39	$17,171$	$32,246$	212	288	38.5	6.2×
Opt. 3, factor = 2	240	15	$11,852$	$21,773$	198	115	33.5	16.1×
Opt. 3, factor = 4	240	21	$13,357$	$24,909$	232	167	35.5	11.5×
Opt. 3, factor = 8	240	34	$17,079$	$33,001$	294	274	39.5	7.1×

The pragma unrolling factor = 2 achieves peak performance with 16.1× speedup compared to the **Opt. 2, Baseline** design. This is at the expense of additional DSP resources. The resource consumptions of ALUT, REG, MLAB, and RAM decrease. It can be observed that larger unrolling factors does not provide higher throughput, which is not as expected.

3.3 Manual Loop Unrolling

The manual loop unrolling is implemented by the following code. As compared, three unrolling factors are considered, including 2, 4, and 8. Folder **mm_optimized3** includes the associated implementation. The synthesis results are summarized in Table 2.

```
auto ptr_a = a.get_pointer()+row*width_a;
auto ptr_b = b.get_pointer()+col*width_b;

float sum = 0.0f;

// Compute the result of one element of c
for (int i = 0; i < width_a; i+=2)
{
    float a_temp1 = PrefetchingLSU::load(ptr_a+i);
    float b_temp1 = PrefetchingLSU::load(ptr_b+i);
    float a_temp2 = PrefetchingLSU::load(ptr_a+i+1);
    float b_temp2 = PrefetchingLSU::load(ptr_b+i+1);
    sum +=  a_temp1 * b_temp1 + a_temp2 * b_temp2;
}

auto ptr_c = c.get_pointer();
BurstCoalescedLSU::store(ptr_c + row*width_c + col, sum);
```

3.4 Question 3

- *What are the effects and general trends of performing manual unrolling? Are the results as expected?*

  From Table 2, the manual unrolling reduces the latency and improves the throughput by a factor of 7.1× to 16.1×. The required ALUT, REG, MLAB, and RAM are slightly less than those of pragma unrolling. But manual unrolling consumes more DSPs. This is expected since an adder tree is implemented.

4 Block Matrix Multiplication (BMM)

The design space exploration is conducted as the following process. First, the parameters and pragmas are automatically modified by a BASH script. Then the revised design is synthesized and corresponding performance metrics are extracted. Folder **mm_block** includes the processing script as well as the code.

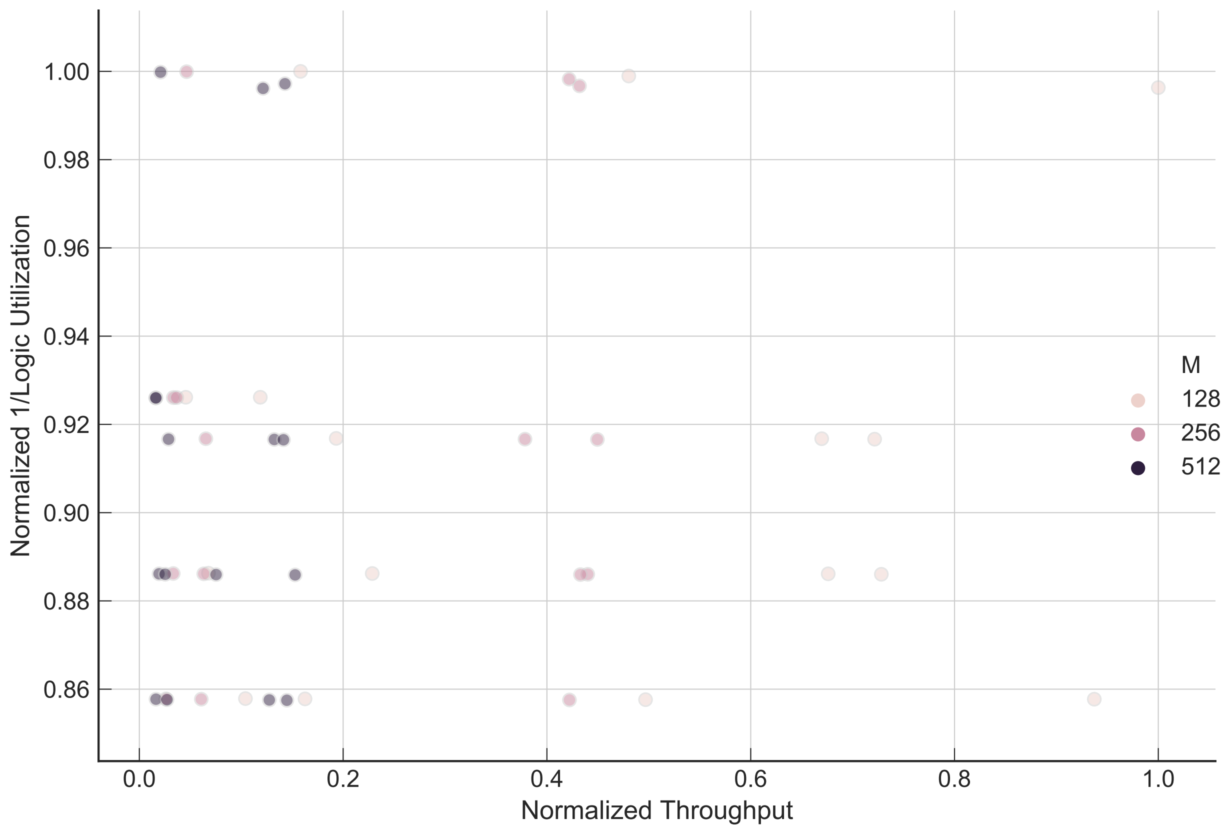

Figure 1: Normalized throughput and logic utilization for matrix multiplication with three different sizes. $N = 128, 256, 512$

The design space exploration is three-dimension, where three design parameters are discussed, including matrix size, block size, and loop unrolling factor. The matrix sizes are 128, 256, and 512. The block sizes are 4, 8, 16, and 32. The loop unrolling factors are 1, 2, 4, and 8. The normalized throughput and logic utilization are summarized in Figure 1.

Table 3: Pareto front for three types of matrix multiplication sizes.

Matrix Size	Block Size	Unrolling Factor	c_calc Logic	Throughput
$N = 128$	16	8	28439	2943.6
$N = 256$	32	4	26163	1323.2
$N = 512$	32	2	25287	450.0

Table 3 gives the Pareto fronts for all three types of matrix multiplication.