CSE565M: Acceleration of Algorithms in Reconfigurable Logic

Learn by Doing: DFT (Pt. 2)

Anthony Cabrera

FL24::L08

Washington University in St. Louis

Table of contents

Matrix Vector Multiplication Optimizations

Baseline Code

```
#define SIZE 8
typedef int BaseType;

void matrix_vector(BaseType M[SIZE][SIZE], BaseType V_In[SIZE],
     BaseType V_Out[SIZE]) {
  BaseType i, j;
data_loop:
  for (i = 0; i < SIZE; i++) {
    BaseType sum = 0;
  dot_product_loop:
    for (j = 0; j < SIZE; j++) {
      sum += V_In[j] * M[i][j];
    }
    V_Out[i] = sum;
  }
}
```

Figure 1: Simple code implementing a matrix-vector multiplication.

This relatively simple code has many design choices that can be performed when mapping to hardware.

Design Choices

Memory organization is one of the more important decisions. The question boils down to *where do you store the data from your code?* There are a number of options when mapping variables to hardware. The variable could simply be a set of wires (if its value never needs saved across a cycle), a register, RAM or FIFO. All of these options provide tradeoffs between performance and area.

Another major factor is the amount of parallelism that is available within the code. Purely sequential code has few options for implementation. On the other hand, code with a significant amount of parallelism has implementation options that range from purely sequentially to fully parallel.

Design Choices

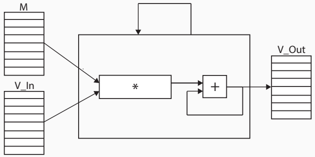


Figure 2: A possible implementation of matrix-vector multiplication from the code in Figure 1.

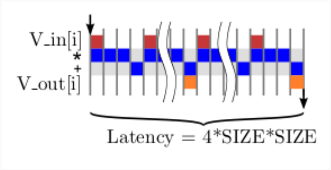


Figure 3: Resulting schedule. Does not consume a lot of area, but the task latency and task interval are relatively large.

Pipelining and Parallelism

Exploit Paralellism in `dot_product_loop`

The expression `sum += V_In[j] * M[i][j];` is executed in each iteration of the loop. In this case, the sum variable has been completely eliminated and replaced with multiple intermediate values in the larger expression.

```c
#define SIZE 8
typedef int BaseType;

void matrix_vector(BaseType M[SIZE][SIZE], BaseType V_In[SIZE],
                   BaseType V_Out[SIZE]) {
  BaseType i, j;
data_loop:
  for (i = 0; i < SIZE; i++) {
    BaseType sum = 0;
    V_Out[i] = V_In[0] * M[i][0] + V_In[1] * M[i][1] +
               V_In[2] * M[i][2] + V_In[3] * M[i][3] +
               V_In[4] * M[i][4] + V_In[5] * M[i][5] +
               V_In[6] * M[i][6] + V_In[7] * M[i][7];
  }
}
```

Figure 4: The mat-vec mult example with a manually unrolled inner loop. Can also achieve this with `#pragma HLS unroll`

For comparison

```
#define SIZE 8
typedef int BaseType;

void matrix_vector(BaseType M[SIZE][SIZE], BaseType V_In[SIZE],
     BaseType V_Out[SIZE]) {
  BaseType i, j;
data_loop:
  for (i = 0; i < SIZE; i++) {
    BaseType sum = 0;
  dot_product_loop:
    for (j = 0; j < SIZE; j++) {
      sum += V_In[j] * M[i][j];
    }
    V_Out[i] = sum;
  }
}
```

What Does the Benefit Look Like?

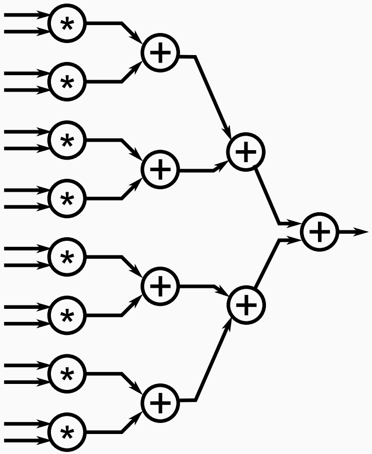
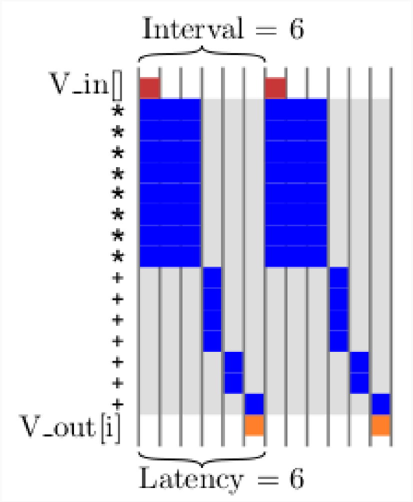


Figure 5: A data flow graph of the expression resulting from the unrolled inner loop from Figure 4.

What Does the Benefit Look Like?

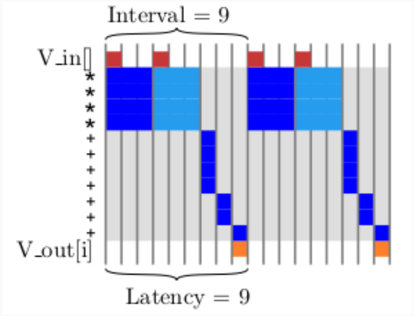


What Does the Benefit Look Like?

- If we wish to achieve the minimum task latency for the expression resulting from the unrolled inner loop, all eight of the multiplications should be executed in parallel.
- Assuming that the multiplication has a latency of 3 cycles and addition has a latency of 1 cycle, then all of the `V_In[j] * M[i][j]` operations are completed by the third time step.
- The summation of these eight intermediate results using an adder tree takes $\log 8 = 3$ cycles.
- Hence, the body of `data_loop` now has a latency of 6 cycles for each iteration and requires 8 multipliers and 7 adders.
- We went from a latency of $4 \times \text{SIZE} \times \text{SIZE}$ cycles to 6 cycles

What if I can't afford all those multipliers?

We can reuse multipliers but at the expense of latency, i.e., we run into a **area/latency tradeoff**.



Additional Benefits from Pipelining?

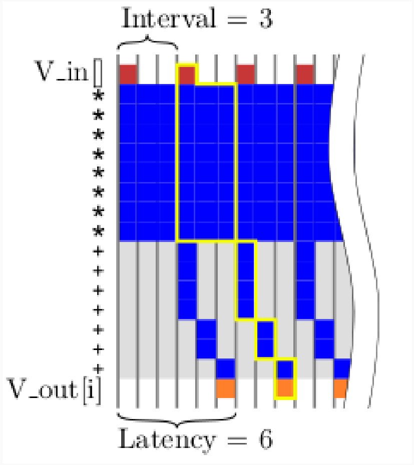
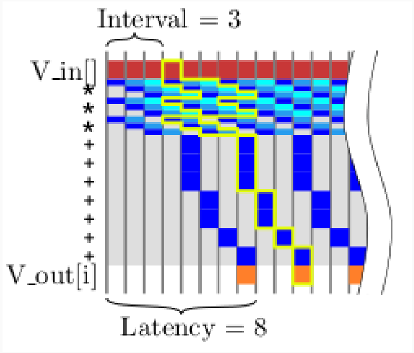


Figure 6: `#pragma HLS pipeline II=3`

What if we pipelined the multiplier?



Varying shades of blue represent each mult that needsd to happen in each inner loop

Storage Tradeoffs and Array Partitiioning

Storage Knobs pt 1

```
#define SIZE 8
typedef int BaseType;

void matrix_vector(BaseType M[SIZE][SIZE], BaseType V_In[SIZE],
    BaseType V_Out[SIZE]) {
#pragma HLS array_partition variable=M dim=2 complete
#pragma HLS array_partition variable=V_In complete
  BaseType i, j;
data_loop:
  for (i = 0; i < SIZE; i++) {
#pragma HLS pipeline II=1
    BaseType sum = 0;
  dot_product_loop:
    for (j = 0; j < SIZE; j++) {
      sum += V_In[j] * M[i][j];
    }
    V_Out[i] = sum;
  }
}
```

Figure 7: Matrix-vector multiplication with a particular choice of array partitioning and pipelining.

Storage Knobs pt 1

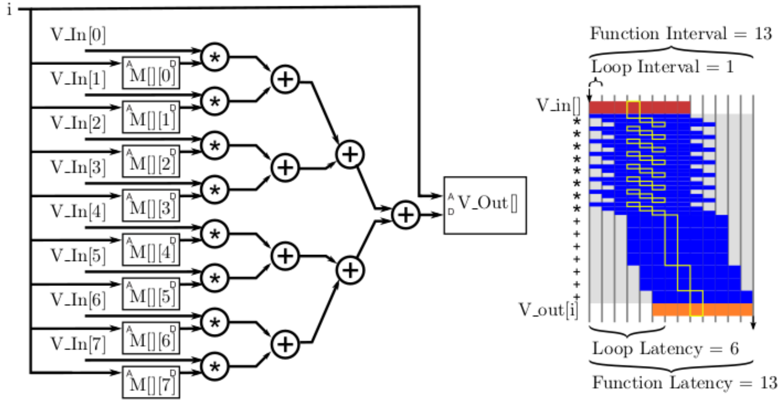


Figure 4.12: Matrix-vector multiplication architecture with a particular choice of array partitioning and pipelining. The pipelining registers have been elided and the behavior is shown at right.

Storage Knobs pt 2

```c
#define SIZE 8
typedef int BaseType;

void matrix_vector(BaseType M[SIZE][SIZE], BaseType V_In[SIZE],
    BaseType V_Out[SIZE]) {
#pragma HLS array_partition variable=M dim=2 cyclic factor=2
#pragma HLS array_partition variable=V_In cyclic factor=2
  BaseType i, j;
data_loop:
  for (i = 0; i < SIZE; i++) {
    BaseType sum = 0;
  dot_product_loop:
    for (j = 0; j < SIZE; j+=2) {
#pragma HLS pipeline II=1
      sum += V_In[j] * M[i][j];
      sum += V_In[j+1] * M[i][j+1];
    }
    V_Out[i] = sum;
  }
}
```

Figure 8: The inner loop of matrix-vector multiply manually unrolled by a factor of two. What do the arrays look like?

Putting it all together (dft.c)

References i