NETCOPE TECHNOLOGIES

WHITE PAPER

TRADECOPE

LOWER THE LATENCY OF YOUR TRADING SYSTEM

Acceleration of trading algorithms with Tradecope

Introduction

In the field of **Ultra-low latency** or **High-frequency trading** (HFT), primary focus is on the execution speed of trading algorithms. Therefore, FPGA providing much faster accelerated execution than CPU processors, represents a more suitable and more profitable platform for execution of trading algorithms.

Up to the present, most trading algorithms have been implemented using C/C++ code, because by writing algorithms in C/C++ traders can get the best performance out of their trading machine, considering the complexity of the trading algorithm specification. But for a long time the limitation of C/C++ code has consisted in the platform selection, because only standard CPU processor compilers were available.

On the other hand, **Field Programmable Gate Array** (FPGA) circuit, as an alternative platform today, offers significantly lower latency compared with processors. But the standard FPGA development process requires an algorithm to be written in Hardware Description Language (HDL), which is based on different design principles than C/C++.

Vivado HLS is a compiler that translates the C/ C++ code directly into HDL code appropriate for FPGA. As a consequence, a trading algorithm written in C/C++ can be re-targeted to FPGA without knowledge of any HDL. It means, the C/C++ code is used in software during simulation and also as an input during translation to HDL code. Furthermore, no special knowledge of the FPGA domain is required.

NETCOPE TECHNOLOGIES

Process of algorithm translation from C/C++ to HDL

The process of C/C++ code translation into the desired design described by HDL code consists of three basic steps. Assume that the original C/C++ code is available and already verified considering its desired functionality.

For illustration, an example code of a common decision factor from the HFT field - **weighted mid-price**, also known as **microprice** - is used in the whole paper. Microprice is defined by formula $(P^a \times Q^b + P^b \times Q^a)/(Q^b + Q^a)$, where P^a is ask price, P^b is bid price, Q^a is ask quantity, and Q^b is bid quantity.

The **first step** is a straightforward compilation of the original C/C++ code by Vivado HLS compiler without any further actions and modifications. System frequency in the illustrative example with code shown in Fig. 1 is set to **322** MHz (period is **2.99** ns). As the compiler, Vivado HLS version **2018.1** was used to get all outputs presented in the paper.

```
int64_t compute_microprice (int64_t ask_price, uint16_t ask_qty,
int64_t bid_price, uint16_t bid_qty)
{
   return ((ask_price * bid_qty) + (bid_price * ask_qty))/(bid_qty + ask_qty);
}
```

Figure 1: C/C++ code for microprice according to the formula $(P^a \times Q^b + P^b \times Q^a)/(Q^b + Q^a)$

Vivado HLS compiles C/C++ code in Fig. 1 with results after C-synthesis shown in Fig. 2-3. *Virtex Ultrascale+ (xcvu7p-flvb2104-2-i)* was selected as a FPGA part in the settings of Vivado HLS project.

```
================================================================
== Utilization Estimates
================================================================
* Summary:
+-----------------+---------+-------+---------+--------+-----+
|       Name      | BRAM_18K| DSP48E|    FF   |   LUT  | URAM|
+-----------------+---------+-------+---------+--------+-----+
|DSP              |        -|      -|        -|       -|    -|
|Expression       |        -|      -|        0|      95|    -|
|FIFO             |        -|      -|        -|       -|    -|
|Instance         |        -|     32|     1661|     967|    -|
|Memory           |        -|      -|        -|       -|    -|
|Multiplexer      |        -|      -|        -|     341|    -|
|Register         |        -|      -|      541|       -|    -|
+-----------------+---------+-------+---------+--------+-----+
|Total            |        0|     32|     2202|    1403|    0|
+-----------------+---------+-------+---------+--------+-----+
|Available        |     2880|   4560|  1576320|  788160|  640|
+-----------------+---------+-------+---------+--------+-----+
|Utilization (%)  |        0|   ~0  |    ~0   |   ~0   |    0|
+-----------------+---------+-------+---------+--------+-----+
```

Figure 2: Utilization estimates summary from original C/C++ code (without optimizations)

```
Latency (clock cycles):
  * Summary:
  +-----+-----+-----+-----+---------+
  |  Latency  |  Interval | Pipeline|
  | min | max | min | max |   Type  |
  +-----+-----+-----+-----+---------+
  |   75|   75|   75|   75|   none  |
  +-----+-----+-----+-----+---------+
```

Figure 3: Latency summary from original C/C++ code (without optimizations)

If the default output design meets all requirements, generated design HDL files can be then used directly in FPGA vendor tools to generate final bitstream and load it into FPGA.

Otherwise, further optimizations must be performed on the source code. The exemplar design in the paper shows very high latency of **224.25 ns** (**75** clock periods in Fig. 3), which is much higher than our requirements. Therefore, the **next step** covers various optimization techniques that better describe the purpose of the code as the whole as well as the set of several interdependent parts.

The new optimized C/C++ code is depicted in Fig. 4 and Fig. 5. Some of the common HLS optimization techniques were applied.

To mention few of used optimizations briefly:

- Data types of the function arguments were changed to **narrower types with optional bit-width** and arbitrary precision.
- The algorithm was split into several steps with partial results stored in **temporary local variables**.
- The division operator was replaced by multiplication using reciprocal of the original divisor, prepared as a **constant table** in advance.
- HLS directives were added to force Vivado HLS compiler implementing multiplication operators by LookUp Tables (LUT) instead of DSP blocks, and the table by LUTRAM instead of BRAM.

Detailed explanation of these optimization techniques as well as other techniques are available in our Tradecope HFT guideline using Vivado HLS provided to all our customers.

```
template<unsigned int N, unsigned int R>
static void reciprocal_table_init (ap_uint<R> reciprocal_table[N])
{
   reciprocal_table[0] = 0;
   for (uint64_t i = 1; i < N; i++)
   {
   double reciprocal = 1.0 / static_cast<double>(i);
   reciprocal_table[i] = ap_uint<R> (reciprocal * (1ULL << R));
   }
}
```

Figure 4: Initialization of the constant table with all scaled reciprocals

```
template<unsigned int N, unsigned int M, unsigned int R>
ap_int<N> compute_microprice (ap_int<N> ask_price, ap_uint<M> ask_qty,
                              ap_int<N> bid_price, ap_uint<M> bid_qty)
```

```
{
   const uint64_t reciprocal_table_depth = 1ULL << (M+1);
   static ap_uint<R> reciprocal_table[reciprocal_table_depth];
   reciprocal_table_init<reciprocal_table_depth, R>(reciprocal_table);
   ap_int<64> price_spread = 0;
   ap_uint<33> size_sum = 0;
   ap_int<96> weighted_spread = 0;
   ap_uint<R> size_sum_reciprocal = 0;
   ap_int<96> retval_tmp = 0;
   ap_int<64> retval = 0;
   #pragma HLS resource variable=weighted_spread core=Mul_LUT
   #pragma HLS resource variable=retval_tmp core=Mul_LUT
   #pragma HLS resource variable=reciprocal_table core=ROM_nP_LUTRAM
   // Step 1
   price_spread = ask_price - bid_price;
   size_sum = ask_qty + bid_qty;
   // Step 2
   weighted_spread = price_spread * bid_qty;
   size_sum_reciprocal = reciprocal_table[size_sum];
   // Step 3
   retval_tmp = (weighted_spread * size_sum_reciprocal) >> R;
   // Step 4
   retval = retval_tmp + bid_price;
   return retval;
}
```

Figure 5: Optimized version of the C/C++ code for microprice shown in Fig. 1 with N=64, M=12, and R=26.

The optimized output design reaches the properties shown in Fig. 6-7.

```
================================================================
== Utilization Estimates
================================================================
* Summary:
+-----------------+---------+-------+---------+--------+-----+
|       Name      | BRAM_18K| DSP48E|    FF   |   LUT  | URAM|
+-----------------+---------+-------+---------+--------+-----+
|DSP              |        -|      -|        -|       -|    -|
|Expression       |        -|      -|        0|     162|    -|
|FIFO             |        -|      -|        -|       -|    -|
|Instance         |        -|      0|        0|   10941|    -|
|Memory           |        -|      -|       26|    3328|    -|
|Multiplexer      |        -|      -|        -|      38|    -|
|Register         |        -|      -|      504|       -|    -|
+-----------------+---------+-------+---------+--------+-----+
|Total            |        0|      0|      530|   14469|    0|
+-----------------+---------+-------+---------+--------+-----+
|Available        |     2880|   4560|  1576320|  788160|  640|
+-----------------+---------+-------+---------+--------+-----+
|Utilization (%)  |        0|      0|    ~0   |       1|    0|
+-----------------+---------+-------+---------+--------+-----+
```

Figure 6: Utilization estimates summary from optimized C/C++ code

```
Latency (clock cycles):
    * Summary:
    +-----+-----+-----+-----+---------+
    |  Latency  |  Interval | Pipeline|
    | min | max | min | max |   Type  |
    +-----+-----+-----+-----+---------+
    |    5|    5|    5|    5|   none  |
    +-----+-----+-----+-----+---------+
```

Figure 7: Latency summary from optimized C/C++ code

The optimized design shows **significantly lower latency** in Fig. 7 equal to **14.95 ns** (only **5** clock periods compared with the original **75** periods) and **elimination of all previously used DSP blocks** (Fig. 6). This all is given at the price of increased FF and LUT resources, which are plentiful in FPGA circuits.

In the **third step**, the functionality of the modified algorithm is verified by simulation using **gcc** compiler. But the standard gcc compiler ignores all HLS directives (pragmas) used in the code. Therefore, the generated HDL design must be further verified using Vivado HLS co-simulation or user-defined HDL testbench. In most cases, C/RTL co-simulation is enough and it does not require additional effort, because the same C/C++ testbench code, used in simulation, is also automatically used in co-simulation.

On top of that, Tradecope provides a complex framework to test the desired functionality of C/C++ code in the wider context of a HFT machine. The framework consists of the whole **software model** (including all processing steps starting by receiving feeds from pcaps and ending by sending orders) and **management scripts** for user-friendly usage. And the best thing about the framework, **the model behaves completely in the same way as the final design in FPGA except the processing speed. Hence the trader does not even need a physical FPGA card to try his algorithm by himself.**

For more information do not hesitate to contact us.