

Optimizing the matrix-matrix product with HCE - Extended abstract

G.Brusco, A. Marongiu, P.Palazzari
Ylichron
Rome

Introduction

The goal of High Level Synthesis is to allow the design of dedicated hardware architectures through conventional programming languages, like the ANSI-C. All the low level hardware details should be hidden to the end-user and all the optimizations to be performed should be introduced by means of the high level language. In this work we shortly introduce the HARWEST Compiling Environment (HCE) which is a C to VHDL compiler, able to extract the parallelism from the original C code and to generate an optimized parallel architecture to be implemented on FPGA technology. Then we discuss the optimized HCE implementation of the matrix-matrix product, analyzing its performances as a function of system parameters.

The HARWEST Compiling Environment

The HARWEST Compiling Environment (HCE) is one of the first outcomes of the HARWEST research project, funded by the Italian Ministry for University and Research, aimed at creating a fully automated HW/SW co-design environment. In the HCE, the system specifics are given by means of an ANSI C program: actually we adopt both a subset of the ANSI C, not allowing the use of pointers and of recursive functions, and a small extension toward C++, supporting the two SystemC `sc_bv` and `sc_fixed` data types. As these two data types are implemented as C++ objects, with their methods and dedicated operations, a C++ compiler is needed to run the HCE C input programs. This, together with the `&` operator used when declaring a function parameter, are the only C++ features accepted by the HCE.

Starting at a very high level of abstraction (C language), the correctness of the specifics is checked by running and debugging, on a conventional system, the C program; of course, we do not afford the task of proving the correctness of a C program and we consider a program to be correct once it works correctly on some given sets of input data. Once we are satisfied with the program behaviour, i.e. after the debugging and testing phase, the input specifics are translated into the Control and Data Flow Graph (CDFG) model of computation. Given a set of resources (number of basic building block modules), the CDFG is mapped into a Data Path and a Control Finite State Machine (FSM) which enforces the behaviour expressed by the starting C program. Such an architecture, represented by the Data Path and the Control FSM, is further optimized (some redundant logic is removed and connections are implemented and optimized) and translated into an equivalent synthesizable VHDL code. Finally, the VHDL program is processed through the standard proprietary design tools (translation and map into the target technology, place and route) to produce the configuration bit stream.

It is worth to be underlined that the debugging phase is executed only at very high level of abstraction (on the C program), while all the other steps are fully automated and result to be correct by construction: this fact ensures that, once we have a working C program, the final architecture will show the same program behaviour.

The basic matrix-matrix product

We want to implement the general matrix-matrix product, which takes as input the two matrices \mathbf{a} and \mathbf{b} and returns the matrix $\mathbf{c}=\mathbf{a}\mathbf{b}$; for optimization reasons, the \mathbf{b} matrix will be loaded in the transposed form \mathbf{b}^T :

```
void mm_mul(float a[m][l],float b_T[l][n],float c[m][n])
{
    for(int i=0;i<m;i++)
        for(int j=0;j<n;j++)
            {
                c[i][j]=0;
                for(int k=0;k<l;k++)
                    c[i][j]+=a[i][k]*b_T[j][k];
            }
}
```

As in the case of the optimization for cache memories, we resort to a blocked formulation of the matrix product, based on the function $MAD(C_{ij}, A_{ik}, B_{kj})$ defined as $C_{ij} = C_{ij} + A_{ik} * B_{kj}$, being C_{ij} , A_{ik} , B_{kj} $N \times N$ matrices of fixed size (in our example, $N=64$) and $+$, $*$ the usual sum and multiply matrix operators. The block matrix M_{ij} is the $N \times N$ block extracted from position (i,j) of the global matrix \mathbf{m} . According to this notation, the `mm_mul` algorithm can be written as

```
void mm_mul(float a[m][l],float b_T[l][n],float c[m][n])
{
    for(int i=0;i<m;i+=N)
        for(int j=0;j<n;j+=N)
            {
                Ci,j =0; /* matrix Ci,j set to zero */
                for(int k=0;k<l;k+=N)
                    MAD(Ci,j, Ai,k, BTj,k)
            }
}
```

Implementation of the basic matrix-matrix product in the HCE

Let us focus on the $MAD(C_{i,j}, A_{i,k}, B_{j,k}^T)$ function, defined as

```
void MAD(float C [N][N], float A[N][N], float B_T[N][N])
{
    for(int i=0;i<N;i++)
        for(int j=0;j<N;j++)
            for(int k=0;k<l;k++)
                C[i][j]+= A[i][k]*B_T[j][k];
}
```

In order to efficiently implement the MAD function, we should store the \mathbf{A} and \mathbf{B}_T matrices columnwise, that is each column A_i and B_j^T of these matrices has to be stored in a different BlockRAM module of the FPGA (we are referring to the Xilinx Virtex5 FPGA family); furthermore, in order to avoid the underutilization of the available memory modules, the i^{th} columns of the \mathbf{A} and \mathbf{B}^T matrices are stored in the same Block RAM module, as shown in figure 1. Using the HARWEST naming style, the \mathbf{A} and \mathbf{B}_T matrices are “split” along the 2^{nd} dimension and they are declared as a unique array through the statement

```
float a_bt[2*N][N] /*#HWST split 2 N*/
```

which partitions the $\mathbf{a_bt}$ matrix into N $\mathbf{a_bt_i}[2*N][1]$ matrices ($i=0, 1, \dots, N-1$) which can all be accessed in parallel. The lower N elements of $\mathbf{a_bt_i}$ contain the i^{th} column of matrix \mathbf{A} and the upper N elements contain the i^{th} column of the \mathbf{B}^T matrix.

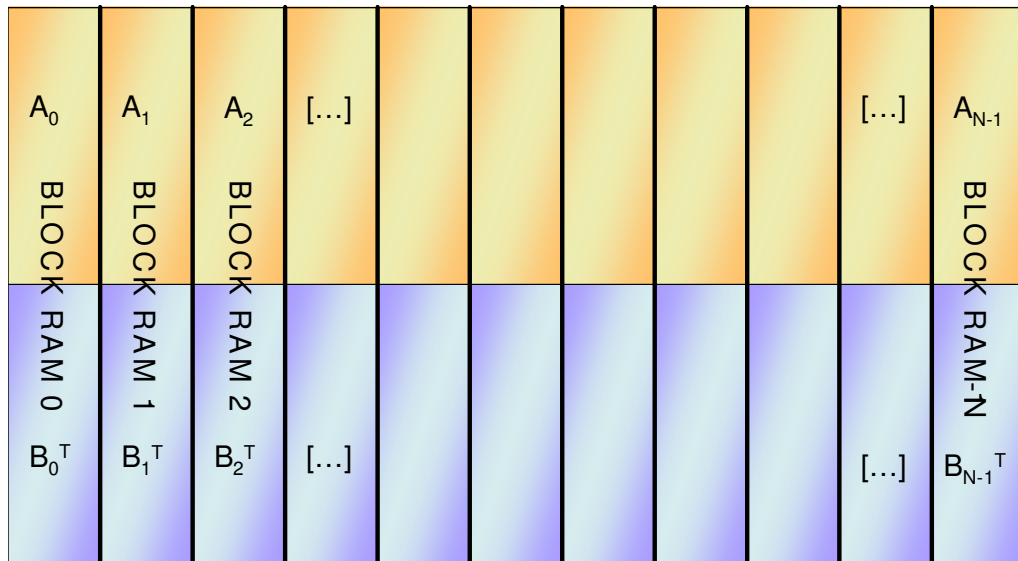


Figure 1: columnwise distribution of \mathbf{A} and \mathbf{B}^T matrices within the FPGA

Thanks to this data distribution, all the N accesses to $\mathbf{A}[i][k]$ ($k=0, 1, \dots, N-1$) can be performed in parallel at the beginning of each iteration of the first for cycle (indexed by i); similarly all the N accesses to $\mathbf{B}_T[j][k]$ ($k=0, 1, \dots, N-1$) can be performed in parallel at the beginning of each iteration of the second for cycle (indexed by j). For each pair (i, j) is possible to compute in parallel the N products $\mathbf{tmp}[k] = \mathbf{A}[i][k] * \mathbf{B}_T[j][k]$ ($k=0, 1, \dots, N-1$) and sum them with a classic tree adder scheme.

From previous discussion, it seems reasonable to think to a pipelined hardware architecture that has the data path structure as shown in figure 2.

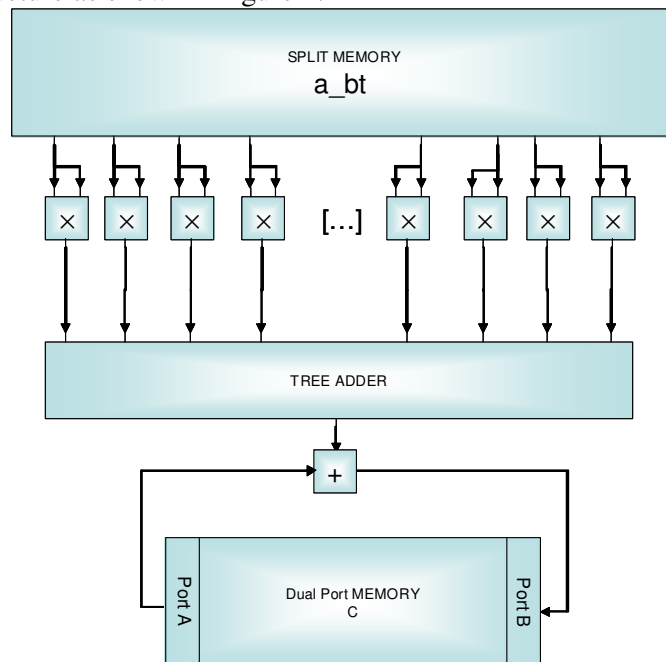


Figure 2: structure of the data path to compute the MAD function

From what we have said, it should be clear that we are going toward a structure with the following behaviour:

```

for(int i=0;i<N;i++)
{
  Read in parallel the N values tmp_a[k] = a_bt[i][k]; (k=0,1,...,N-1)
  for(int j=0;j<N;j++)
  {
    Read in parallel the N values tmp_b[k] = a_bt[j+N][k]; (k=0,1,...,N-1)
    Compute in parallel the N products tmp[k] = tmp_a[k] * tmp_b[k]
    Compute the TreeAddVal = TreeAdd(tmp[0],...,tmp[N-1])
    C[i][j] = C[i][j] + TreeAddVal
  }
}

```

If all the components are pipelined, the data path in figure 2 can, after the pipeline latency,

- read $N+1$ data from the memories,
- perform N multiply operations,
- perform one TreeAdd operator ($N-1$ flops)
- perform one final sum and the consequent write to memory

Referring to a clock period T_{ck} , previous data path has an internal I/O bandwidth $BW = \frac{4(N+2)}{T_{ck}}$

[Byte/s] and a peak computational power $P = \frac{(2N-1)}{T_{ck}}$ [Flop/s]. The sustained performances will

be lowered by the pipeline start-up penalties.

In order to instruct HCE to produce an architecture similar to the one sketched above, we use its directives to force the spatial data distribution (the **HWST split** directive) and the pipeline behaviour (the **HWST pipeline** directive).

To clarify the ideas, we report some portions of the source code. As HCE maps each C function onto a dedicated HW module, in order to define the pipelined TreeAdd component we have to declare the following function (we are in the case $N=64$) which will be synthesized as a pipeline component which, thanks to the tree-like data dependencies, will perform the tree-based sum of the $temp_i$ ($i=0,1,\dots,63$) input values:

```

/*#HWST pipeline*/
float TreeAdder( float temp0, float temp1, ..., float temp63 )
{ //... declarations are omitted
  /* first stage*/
  sum0_0 = temp0 + temp1;
  sum0_1 = temp2 + temp3;
  ...
  sum0_31 = temp62 + temp63;
  /* second stage*/
  sum1_0 = sum0_0 + sum0_1;
  sum1_1 = sum0_2 + sum0_3;
  ...
  sum1_15 = sum0_30 + sum0_31;
  /* third stage*/
  sum2_0 = sum1_0 + sum1_1;
  sum2_1 = sum1_2 + sum1_3;
  ...
  sum2_7 = sum1_14 + sum1_15;
  /* fourth stage*/
  sum3_0 = sum2_0 + sum2_1;
  sum3_1 = sum2_2 + sum2_3;
  sum3_2 = sum2_4 + sum2_5;
  sum3_3 = sum2_6 + sum2_7;
}

```

```

    /* fifth stage*/
    sum4_0 = sum3_0 + sum3_1;
    sum4_1 = sum3_2 + sum3_3;
    /* last stage*/
    return sum4_0 + sum4_1;
}

```

The MAD function is declared below the TreeAdder function, so that in the MAD synthesis the TreeAdder module will be used whenever the TreeAdder function will be called in the MAD code. MAD has the following structure (implementation details are skipped): the `HWST split` directive is used both to split the `a_bt` matrix and to unroll the for (j) loop; the unrolling process is used to statically determine the indices used to address the split dimension, accessing the proper blockRAM bank.

```

void MAD(float a_bt[2*N][N]/*#HWST split 2 N*/, float c)
{
    //...declarations are omitted; the indices ii and jj are linear functions of i and j and
    //are used to compensate the offsets to properly access the a_bt matrix
    for (i=0; i<N; i++)
    {
        /*#HWST split */
        for (j=0; j<N; j++)
        {
            temp0= a_bt[ii][0] * a_bt[jj][0];
            temp1= a_bt[ii][1] * a_bt[jj][1];
            ...
            temp63= a_bt[ii][63] * a_bt[jj][63];

            c[i][j] = c[i][j] + TreeAdder( temp0, temp1, ..., temp63 );
        }
    }
}

```

When previous code is compiled through HCE, both the data path reported in figure 2 and a control Finite State Machine (FSM) are generated. The FSM consists (except some heading and tailing states) of a sequence of $N_S=144$ states which are repeated N times; during the i^{th} iteration ($i=0,1,\dots,N-1$), the FSM computes the $c[i][0], \dots, c[i][N-1]$ values. As the computation of each $c[i][j]$ value requires $2N-1$ flops, the system sustains a computation rate $SR = \frac{N(2N-1)}{N_S T_{ck}}$ [Flop/s]; its

efficiency with respect the theoretical peak $P = \frac{(2N-1)}{T_{ck}}$ can be quantified as

$\eta = \frac{SR}{P} = \frac{N}{N_S} = \frac{64}{144} = 0.44$, that is the pipeline start-up penalties cause a performance reduction of 56%. As $N_S T_{ck}$ is the time required to compute one row $c[i]$ of the c matrix, the time necessary to compute the whole MAD function is given by $t_{MAD} = N(N_S T_{ck})$.

Performance evaluation of the basic matrix-matrix product

Let us now analyze which are the performances that we can expect from the `mm_mu1` algorithm. We have just determined that the inner MAD module sustains a computation rate

$SR = \frac{N(2N-1)}{N_S T_{ck}}$ [Flop/s]. Such an efficiency figure is an upper bound for the whole `mm_mu1`

algorithm. In fact, without using special hardware organization to overlap communications and computation, each execution of MAD will be preceded by the download operation (from the main

memory into the FPGA) of the matrices $\mathbf{A}_{i,k}$, $\mathbf{B}_{j,k}^T$; before/after performing the inner for (k) loop, the $\mathbf{c}_{i,j}$ matrix has to be downloaded/uploaded from/into the main memory. By indicating with BW the bandwidth between the FPGA and the main memory, we can express the computing time of the `mm_mul` algorithm through the following expression:

$$t_{mm_mul} = \frac{(m \cdot n)}{N^2} \left(\underbrace{\frac{(2N^2) \cdot 4}{BW}}_{\text{Read/Write } C_{ij}} + \frac{l}{N} \left(\underbrace{\frac{(2N^2) \cdot 4}{BW}}_{\text{Read } A_{ik} \cdot B_{jk}^T} + t_{MAD} \right) \right) \quad [1]$$

Still maintaining nearly the same structure of the inner MAD operator, we can improve the performances of the `mm_mul` algorithm by introducing a further level of blocking. In fact, while having still more memory modules, we could be forced to maintain a “small” value for N due to constraints on the available hardware (for instance, the number of DSP modules). As in the blockRAM modules there is enough space to contain 4 `a_bt[2N][N]` matrices (again, we are referring to the Virtex5 blockRAM modules which can store 512 32bit words), we decided to download 4 `A[N][N]` and 4 `BT[N][N]` matrices into the new `A_BT[4*2N][N]` array; similarly, we decided to put 16 `c[N][N]` blocks into the new `C [16N*N]` matrix: in such a way for each data transfer we can compute 16 block matrix MADs, each related to a different pairing of the 4 A and the 4 B^T blocks. The signature of the MAD function has been slightly modified, adding two integer indices to determine which blocks of A, B^T and C has to be used in the current iteration. We write the `MM_blocked_MAD` function as

```
void MM_blocked_MAD(float a_bt[4*2*N][N] /*#HWST split 2 N*/, float c[16*N*N])
{
    for( i_2 = 0; i_2 < 4*N ; i_2+= N )
        for( j_2 = 0; j_2 < 4*N ; j_2+= N )
            MAD(a_bt, c, i_2, j_2);
}
```

In the previous `MM_blocked_MAD` function the basic MAD function is computed 16 times, each time accessing a different pair of blocks of A and B^T and updating one of the 16 blocks of the C blocks.

Using the `MM_blocked_MAD` function, the `mm_mul` algorithm becomes

```
void mm_mul(float a[m][l], float b_T[l][n], float c[m][n])
{
    for(int i_1 = 0; i_1 < m; i_1 += 4*N) {
        for(int j_1 = 0; j_1 < n; j_1 += 4*N) {
            download 16 N*N blocks from c[m][n] to C[16*N*N];
            for(int k = 0; k < l; k += N) {
                download 8 blocks from a[m][l] and b[l][n] to A_BT[8*N][N]
                MM_blocked_MAD(a_bt, C);
            }
            Upload 16 N*N blocks from C[16*N*N] to c[m][n];
        }
    }
}
```

Let us now investigate the advantages of this 2-level blocking scheme. It is easy to verify that the execution time of the module `MM_blocked_MAD` is $t_{MM_blocked_MAD} = 16 \cdot t_{MAD}$. Taking into account the data movements to/from the FPGA, the time necessary to compute the new `mm_mul` algorithm is given by

$$t_{mm_mul}^{new} = \frac{(m \cdot n)}{16N^2} \left(\underbrace{\frac{2 \cdot (16N^2) \cdot 4}{BW}}_{\text{Read/Write } C[16N*N]} + \frac{l}{N} \left(\underbrace{\frac{(8N^2) \cdot 4}{BW}}_{\text{Read } A_BT[8N*N]} + 16 \cdot t_{MAD} \right) \right) \quad [2]$$

Comparing [1] with [2] we see that the computing time remains unchanged ($t_{comp} = \frac{(m \cdot n \cdot l)}{N^3} t_{MAD}$), while the communication time passes from

$$t_{comm} = \frac{(m \cdot n)}{N^2} \left(\underbrace{\frac{(2N^2) \cdot 4}{BW}}_{\text{Read / Write } C_{ij}} + \frac{l}{N} \underbrace{\frac{(2N^2) \cdot 4}{BW}}_{\text{Read } A_{ik}, B_{jk}^T} \right) \quad [3]$$

to

$$t_{comm}^{new} = \frac{(m \cdot n)}{16N^2} \left(\frac{2 \cdot (16N^2) \cdot 4}{BW} + \frac{l}{N} \left(\frac{8N^2 \cdot 4}{BW} \right) \right) \quad [4]$$

We define the time to move two $N \times N$ blocks as $T_2 \equiv \frac{(2N^2) \cdot 4}{BW}$

So [3] and [4] can be written as

$$t_{comm} = \frac{(m \cdot n)}{N^2} T_2 \left(1 + \frac{l}{N} \right) \quad [5]$$

and

$$t_{comm}^{new} = \frac{(m \cdot n)}{N^2} T_2 \left(1 + \frac{l}{4N} \right) \quad [6]$$

As, for real cases, l results to be much larger than N , the new formulation allows to reduce the overhead term connected to data movement of a factor 4. To quantify the speedup on the overall computation, let us refer to “reasonable” parameters for the clock period and for the I/O bandwidth, assuming $T_{ck}=5\text{ns}$ and $BW=1\text{GB/s}$:

- in both the cases, being $t_{MAD} = N(N_S T_{ck})$, the “pure” computing time is

$$t_{comp} = \frac{(m \cdot n \cdot l)}{N^3} (64(144 \cdot 5 \times 10^{-9})) = \frac{(m \cdot n \cdot l)}{N^3} \times 46 \mu\text{s};$$

- without the 2nd level of blocking, the communication time is ($l \gg N$)

$$t_{comm} \cong \frac{(m \cdot n \cdot l)}{N^3} T_2 = \frac{(m \cdot n \cdot l)}{N^3} \times 32.8 \mu\text{s}$$

- with the 2nd level of blocking, the communication time is ($l \gg N$)

$$t_{comm}^{new} = \frac{(m \cdot n \cdot l)}{N^3} T_2 \frac{1}{4} = \frac{(m \cdot n \cdot l)}{N^3} \times 8.2 \mu\text{s}$$

Combining previous numbers, we obtain the speedup figure $S = \frac{t_{comp} + t_{comm}}{t_{comp} + t_{comm}^{new}} = \frac{46 + 32.8}{46 + 8.2} = 1.45$,

corresponding to a reduction of 31% of the global computing time.