High-Level Synthesis Optimization for Blocked Floating-Point Matrix Multiplication

Erik H. D’Hollander
Electronics and Information Systems Department
Ghent University, Ghent, Belgium
Erik.DHollander@ugent.be

ABSTRACT

In the last decade floating-point matrix multiplication on FPGAs has been studied extensively and efficient architectures as well as detailed performance models have been developed. By design these IP cores take a fixed footprint which not necessarily optimizes the use of all available resources. Moreover, the low-level architectures are not easily amenable to a parameterized synthesis. In this paper high-level synthesis is used to fine-tune the configuration parameters in order to achieve the highest performance with maximal resource utilization. An exploration strategy is presented to optimize the use of critical resources (DSPs, memory) for any given FPGA. To account for the limited memory size on the FPGA, a block-oriented matrix multiplication is organized such that the block summation is done on the CPU while the block multiplication occurs on the logic fabric simultaneously. The communication overhead between the CPU and the FPGA is minimized by streaming the blocks in a Gray code ordering scheme which maximizes the data reuse for consecutive block matrix product calculations. Using high-level synthesis optimization, the programmable logic operates at 93% of the theoretical peak performance and the combined CPU-FPGA design achieves 76% of the available hardware processing speed for the floating-point multiplication of 2K by 2K matrices.

1. INTRODUCTION

Ever since the seminal paper by Gerald Estrin about extending a fixed CPU with a variable part [2], the idea to accelerate computations by an algorithm in hardware has been a tantalizing prospect. Nowadays, the major players in the computer industry offer silicon-on-chip solutions where a multicore ARM processor or softcore is tightly integrated with the configurable logic fabric. High-level synthesis (HLS) tools are closing the gap between the huge programming effort and the effective performance of these systems. The idea to configure hardware using a high-level language has shifted the focus from low-level design to C, C++ or OpenCL code annotated with directives and vendor supplied hardware libraries. The quality of the result largely depends on the sophistication of the compiler and the programmer's ability to generate efficient hardware using the multitude of pragmas and design options. Yet the development time is much shorter and the design exploration is guided by useful resource and timing reports.

In recent years, low level architectures for floating-point matrix multiplication have been studied extensively. Detailed analysis of the algorithm resulted in designs in which the bandwidth of the streaming data matches the speed of the execution pipeline. The maximum size of the matrices is defined by the availability of the critical resources (DSPs, Block RAMs) needed for the implementation. Despite the performance of these solutions, they do not optimize the global resource budget, e.g. by maximizing the use of both DSPs and available memory to allow larger matrices and higher calculation speeds. Moreover, the detailed descriptions of the low-level architectures are not easily amenable to a parameterized implementation using high-level synthesis tools. In this paper a matrix algorithm described in C is analyzed with respect to two design objectives: maximize the parallelism and minimize the pipeline cycle time. A stepwise refinement of the algorithm, loop directives and interface definition leads to a balanced allocation of the different resource types which maximizes the on-chip memory use and realizes a speed up to 93% of the peak performance for a single matrix multiply. In order to accommodate larger matrices, a block oriented algorithm is developed in which the block matrices are multiplied on the FPGA and the resulting blocks are added in the CPU. A Gray code block ordering scheme maximizes the data reuse. The resulting CPU-FPGA block oriented multiplication achieves 76% of the FPGA floating point performance using the ZedBoard [10], a development board based on the Xilinx Zynq-7000 SoC combining a Series 7000 programmable logic (PL) FPGA with a dual ARM-A9 processing system (PS).

2. RELATED WORK

Being one of the corner stones of linear algebra, matrix multiplication has received much attention in all kinds of accelerators, such as GPUs, systolic architectures, multi-cores, heterogeneous clusters and FPGAs. Zhuo et al. [11] use a linear array architecture and propose three algorithms which differ by the use of storage size and memory bandwidth. They obtain a performance of 2.06 GFLOPS for a 1K by 1K matrix multiply on a Cray XD1 accelerator. Kumar et al. [4] use a rank-1 update scheme to implement parallel processing elements. Sub blocks of the matrices are streamed to the architecture and intermediate results are accumulated, allowing communication and computation overlap. Theoretical analysis of an 800×800 matrix multiplication shows an execution time of 10^7 cycles. Jovanović and Milutinović [3] present an architecture of $p = 252$ processing elements with local memories to store the input matrices. Large matrices are multiplied by sending blocks to the accelerator. Simulation shows that the design matches the theoretical speedup of $2p$ flops per cycle. In the previous cases, each design required a thorough examination of the control and data paths. Although most designs are offering good performance, they are less effective in two ways. First the development time is excruciatingly long and second the design cannot easily be adapted to make full use of the available resources. The abstraction offered by high-level synthesis decreases the design effort by an order of magnitude and permits a flexible design exploration. E.g. in [1] a floating-point matrix multiplication has been synthesized using the Vivado HLS suite. The design is generated using HLS-directives and is connected to an AXI-4 streaming interface for data exchange with the processor cache of a Zynq 7000 SoC. A performance of 1.82 GFLOPS is obtained on a 32x32 square matrix multiplication with a clock period of 8.41 ns. This design uses 72% of the DSP resources and is limited to matrix sizes up to 42x42, due to exhausting the DSP budget. The approach presented in this paper balances DSP and BRAM resources to store larger matrices in the BRAM blocks. Furthermore a block oriented computation on the embedded processor using the hardware design as accelerator allows matrix sizes exceeding 2K by 2K. The global design optimization to maximize DSP and BRAM utilization, I/O overlap and data reuse is able to more than double the achievable performance on the same hardware.

3. COMPILING FOR FPGAs

In a well-organized FPGA operation, the data streams to the computing elements produce a new result in each execution cycle, multiplied by the number of parallel data streams. How is a program converted into a dataflow graph and how is this graph mapped onto the computing elements of an FPGA? For this, we will consider a program as a set of statements operating on a stream of data. Each statement is realized as a combinatorial function implemented by a LUT (Lookup Table) or a number of LUTs in sequence. The combinatorial function is computed in one clock cycle and the result is stored in a flip-flop. The computing time for n functions or statements equals therefore n clock cycles for one data element. When this pipeline is fed with one data element per cycle, we obtain one result per cycle as soon as the pipeline is filled. The challenge for the compiler is to create deep pipelines using look up tables, DSPs and Block RAM in order to avoid bubbles and gaps and to minimize the pipeline length.

Let us therefore take a look at a multiply add statement and see how it is analyzed by the compiler (Figure 1). Assume that each arithmetic operation takes one clock cycle; therefore flip-flops are needed to store the intermediate results. Now consider the same statement operating on arrays of n elements. The previous sequence of computing blocks can be reused, but the compiler has to add control logic to organize the stream of data from the memory and pipelining the results back to memory. This simple example illustrates the steps a compiler has to take in order to create an intellectual property or IP core. It has to analyze the program, create a dataflow graph subject to the dependence constraints, map the operations onto the available resources and construct the control and data paths.

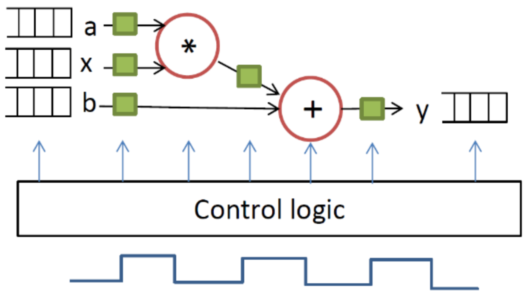

Figure 1. Control path and data path generation for loop $y[i] = a[i] * x[i] + b[i], i = 0..n$. Squares denote flip-flops.

From the same example we can also observe some factors defining the performance of an FPGA. The first is the cycle time of the clock which drives the logic fabric. A small clock cycle time limits the amount of work that can be done in one cycle and therefore more flip-flops and cycles will be needed to implement the design. This has an impact on the speed and the resource consumption. A very large cycle time will require fewer flip-flops but may leave cycles underutilized and therefore create a slower design. A compiler will adapt itself to a given clock frequency, which is part of the resource description.

The second factor is the amount of pipelining achievable from the algorithm. If the compiler is able to create pipelines with n stages, the speed may rise to n-fold the execution speed of the non-pipelined version.

And last but not least there is the parallelism. When an algorithm has n independent data streams, the compiler can organize n parallel pipelines and therefore multiply the speed up of the pipeline by the number of pipelines, creating a very fast implementation. Examples are parallel pipelined operations on the rows of a matrix or the family of systolic algorithms. As we will see the challenge there is to create parallel streams to feed the pipelines simultaneously. The bottom line is that the performance of an FPGA depends on two basic principles: cram as much as possible operations into a pipeline and create as many pipelines as possible which can be fed simultaneously.

4. HLS DESIGN OPTIMIZATION

The successful implementation of an algorithm on an FPGA involves the combined optimization at three levels:

1) pipelining and parallelizing the code to maximally use the computing resources, 2) organizing the memory in order to ensure a continuous data stream and 3) balancing the work between the CPU and the FPGA.

The feasibility of the design can be analyzed using synthesis reports, which mention the usage of the four basic resource types: lookup tables, flip-flops, DSPs and Block RAMs.

In order to focus the attention, we consider the multiplication of two square matrices of size n^2.

```
for i=0; i<n; i++
  for j= 0; j<n; j++ {
    sum=0;
    for k=0; k<n; k++
      sum+=a[i][k]*b[k][j];
    c[i][j] = sum;
  }
```

The computation requires $E = 2n^3$ floating-point operations, executable at one operation per cycle with an optimized IP-core [6]. The design goal is twofold. First we want to maximize the computation load in a single clock cycle. The objective is to obtain one scalar product $c_{ij} = \sum_k a_{ik} b_{kj}$ of a row and a column per time step. This is achieved using a D-stages deep pipeline of floating-point multiply add operations. The second objective is to create parallel streams to feed the pipeline such that one element of the product matrix is output per clock tick. The expected computation time is therefore $t_{comp} = n^2 - 1 + D$ cycles, i.e. D steps for the first element and 1 step for each of the $n^2 - 1$ remaining elements. This yields a speedup $S = O(2n^3/n^2) = 2n,$ which is proportional to the square matrix dimension n when there are unlimited resources available. E.g. with a clock cycle time $t_c = 1.e^{-8}$ and $n = 32$, this gives 3.2 GFLOPS. However the speed is limited by the number of DSPs to create the pipeline and by the memory bandwidth to feed the pipeline. Our platform is the Zedboard with a Zynq 7020 FPGA and our goal is to maximize the floating-point performance. Similar to the operation of a GPU, the data has to be brought into the FPGA, and the results have to be copied back to the memory.

4.1 Directive based optimization

4.1.1 *Bare program execution*

Without any directives or other optimizations, the performance of a 32 x 32 matrix multiply on the FPGA is 18 MFLOPS, single precision. This is close to the performance of a single floating-point multiplication, which takes 5 cycles at 10 ns per cycle. In contrast, the embedded Zynq ARM A9 Application Processor Unit (APU) realizes 253 megaflops for the same matrix size.

4.1.2 *Unroll inner loop k*

In order to increase the performance we will need to activate more DSPs. The first approach is to unroll the inner loop in order to get more parallel iterations. Unrolling the inner loop more than doubles the speed to 38 MFLOPS. The inner loop creates a loop carried dependence on the variable sum and therefore the iterations have to be carried out sequentially. Still, the compiler is able to overlap the addition in one iteration with the multiplication of the next iteration.

4.1.3 *Pipeline inner loop k*

Obviously, the next step is to examine if the loop iterations can be pipelined. The difference between unrolling and pipelining is that with unrolling each iteration is scheduled independently in parallel with the other iterations, therefore each iteration may require a duplication of resources. With pipelining the iterations are scheduled in sequence, using the same resources shifted in time. An additional advantage of pipelining is that it works well with loop carried dependencies. The single most important performance parameter of a pipelined loop is the *initiation interval*, *II*. The initiation interval is the minimum number of clock cycles between the start of two pipelined iterations. Ideally *II=1*, but the initiation interval may be larger because of delays in the data stream, dependences in the algorithm or due to lack of resources.

Let us first try to pipeline the inner loop. In this case the compiler warns that *II=4* due to a loop carried dependence because the summation value from the previous iteration is only available after 4 cycles. The performance is 48 MFLOPS which is still slightly better than unrolling the inner loop, because now the same operations (addition, multiplication) of adjacent iterations are overlapped.

4.1.4 *Pipeline loop j*

In order to create longer and independent pipelines, the next higher loop j is pipelined. This has two effects. First, pipelining an iteration containing an inner loop k requires that the whole inner loop is unrolled. This creates a long pipeline to calculate the scalar product of a row and a column. Second, all scalar products $c_{ij},\ j = 0..n-1$ can be calculated independently. Therefore we expect an initiation interval of 1 between the iterations of the pipelined loop. However, this appears not to be the case. While achieving 310 MFLOPS, the compiler warns that it cannot feed the pipeline fast enough and it ends up with an initiation interval *II=16* between the scalar product computations of $n = 32$ elements. The reason is that the Block RAM memory in the FPGA has only two ports, so only two new elements can be fetched in each cycle. Fortunately, there is a directive to distribute an array over several Block RAMs.

4.1.5 *Array partitioning*

The distribution of an array over multiple Block RAMs is transparent to the program and does not affect the code. Since matrix A is accessed by rows and matrix B is accessed by columns, the array partitioning is applied such that the row elements of A and the column elements of B are located in different memory banks which can be accessed in parallel. The combined effect of pipelining and array partitioning results in a speed of 1,382 MFLOPS with

$n = 32$ and an initiation interval *II=1*, which corresponds to one scalar product per cycle. This is the best we can achieve, since the matrix product is calculated in n^2 cycles, and this is the time to fetch the source matrices A and B in parallel.

4.1.6 *Performance parameters n and II*

In the pipelined schemes one element of the result matrix is computed in II time steps, yielding a computation time $t_{comp} = II\, n^2$. Fetching and storing the matrices requires a communication time of $t_{comm} = 3n^2$. The execution time is therefore

$$t_{exec} = t_{comp} + t_{comm} = (II + 3)n^2 \qquad (1)$$

cycles and the computation performance is

$$P(n, II) = \frac{2n^3}{(II + 3)n^2} = \frac{2n}{II + 3} \qquad (2)$$

flops/cycle. Consequently the performance increases with the matrix size and decreases with the initiation interval. The maximum performance is obtained by looking for the best combination of *n* and *II*.

4.1.7 *Maximize DSPs*

According to equation (2), the performance increases with the matrix size. Therefore one improvement is to look for the largest possible matrix until one of the resources is exhausted. E.g. a matrix multiplication with $n = 43$ maximizes the DSP use and achieves 1,956 MFLOPS. The resource utilization is given in Table 1.

Table 1. Resource budget when maximizing DSP usage

n	II	D	BRAM	DSP	LUT	FF
43	1	221	90	218	31728	18128
			32%	99%	59%	17%

4.1.8 *Maximize DSP and BRAM with II>1*

Table 1 shows that there is still plenty of Block RAM (68%) available and we know that the performance increases with the matrix size. On the other hand we did use the available DSPs with the requirement that II=1. If we would relax this requirement then we may handle larger matrices and improve the performance.

Table 2. Design exploration for II=1..5

II	n	limit	MFLOPS
1	43	DSP	1,956
2	86	DSP	3,217
3	124	BRAM	4,081
4	124	BRAM	3,393
5	124	BRAM	2,984

The result is a trade-off between DSPs and Block RAM as illustrated in Table 2, showing the maximum value of n for $II = 1..5$. The third column indicates the limiting resource. The combined optimization of n and II increases the performance up to 4,081 MFLOPS for $II = 3$ and $n = 124$.

4.2 Optimizing I/O

Is this the best we can do? The answer is yes for optimizing the computation with respect to the available resources and no for the input-output and data stream organization. Two extra steps are: increasing the I/O bandwidth and overlapping I/O streaming and computation.

4.2.1 *Improving the I/O bandwidth*

The standard interfaces to connect customized IP-cores with the ARM processor follow the AXI-protocol. The maximum data-transfer between the programmable logic (PL) and the cache of the processor uses the AXI_ACP (Accelerator Coherency Port) interface which limits the read- and write-bandwidth to 1.2 GB/s and the bus width to 64 bits [9]. Since our core uses a 100 MHz clock and 32-bit I/O, the data stream operates at 400 MB/s, i.e. only one third of the available maximum. Two modifications enable the full bandwidth use. At the PL-side, the data stream is increased from 32 to 128 bits using a resource and interface specification in the high-level language. Furthermore, the AXI-DMA bridge is customized to use a 200 MHz clock at the processor side to compensate for the 64-bit bus limit. These changes lead to a maximum bandwidth of 1.6 GB/s, which is topped by the practical bound of 1.2 GB/s imposed by the hardware characteristics, see Figure 2.

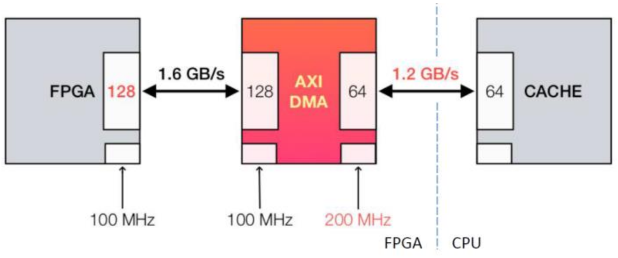

Figure 2. Increasing bus width and DMA clock speed

As a result, the communication cost is reduced by a factor of 3, i.e. $t_{comm} = n^2$ cycles.

4.2.2 *Overlapping computation and communication*

The present program separates the data movement and the computation. In order to overlap computation and communication, there are two options. Either the incoming data are sent directly to the computing elements without copying them into Block RAM, or the output data are sent directly to the CPU memory instead of buffering them in the FPGA. The first approach requires a revision of the whole algorithm by making inner loop k the outer loop, a technique used in [4], while the second approach is much simpler and has the same performance. Note that the matrix elements of the product matrix are available one by one at the end of the inner loop k, which calculates the scalar product of a row of A and a column of B. Instead of storing the result matrix C locally, the computed elements can immediately be put into a streaming buffer which is sent to the memory of the CPU. This has two advantages: first sending output matrix C creates no overhead, i.e. the communication cost is further reduced to $t_{comm} = 2/3\ n^2$ cycles. As a consequence, the optimized performance equation (2) becomes

$$P_o(n, II) = \frac{2n}{II + 2/3} \qquad (3)$$

Furthermore, extra memory is gained in the FPGA because we don't have to store matrix C. This allows to multiply larger matrices, however then the LUTs become the limiting resource. The data streaming protocol is provided by a special streaming class in the HLS language which seamlessly interacts with the DMA and IP cores connecting the logic fabric with the CPU. The efforts to optimize the I/O result in a significant performance increase to 6,295 MFLOPS for $n = 124$ and $II = 3$, i.e. 93% of the theoretical performance $P_0(124{,}3)$ using equation (**3**).

4.3 Large matrices: block computation

An FPGA has a limited amount of fast on-chip memory, typically much lower than in a GPU. Therefore large matrices are multiplied in a block oriented fashion.

4.3.1 Cooperating CPU-FPGA computation

The CPU executes a traditional matrix multiplication, but now the matrix is divided into blocks which are sent to the FPGA for multiplication and the result is added to the proper sub-matrix in the large product matrix. The block computation is described by the following equation.

$$C_{ij} = \underbrace{\boxed{\sum_{k=0}^{m-1}}}_{\text{CPU}} \underbrace{\boxed{A_{ik} \cdot B_{kj}}}_{\text{FPGA}}$$

Figure 3. Load distribution of block oriented matrix computation

Large matrix $C_{n \times n}$ is subdivided in m^2 blocks of block size $b = n/m$ rows and columns. To calculate C_{ij}, blocks A_{ik} and $B_{kj}, k = 0..m-1$ are multiplied consecutively on the FPGA and the resulting blocks are added to the proper submatrix on the CPU, as indicated in Figure 3. CPU and FPGA operate in parallel non-blocking mode. On the ZedBoard the CPU has a memory of 512 MB. While the maximum square matrix size on the IP-core is $n \leq 124$, the block-matrix computation allows a matrix size $n > 2000$. The performance of the combined CPU-FPGA block computation is 4.19 GFLOPS for $n = 2108$ and $II = 3$.

4.3.2 Data reuse using Gray code block ordering

In a naïve straightforward implementation each block multiplication $A_{ik}B_{kj}$ requires sending two block matrices, i.e. $2m^3$ block matrix transfers (see Figure 3). As a consequence both input matrices A and B are sent two times. It is possible to perform a loop interchange and an iteration reordering such that each block matrix is sent only once, a technique also used ad hoc in [3]. By generating the index tuple (k, i, j) using the $(m, 3)$-ary generalized Gray code [5], the k-loop becomes outermost and matrices A_{ik} or B_{kj} can be reused in the inner loops. The Gray code which optimizes the data reuse in the FPGA is given in Figure 4.

```
/* (m,3)-ary Gray code to optimize FPGA data reuse
 * in block-oriented matrix multiplication
 * input: block size m, block matrices $A_{ik}$, $B_{kj}$
 * output: $C$
 */
   Gmax = m*m*m; m2 = m*m;
   for (g =0; g < Gmax; g++) {
     kk = g/m2;            // iterate with stepsize $m^2$
     ii  = (g%m2)/m;       // iterate with stepsize $m$
     jj  = g%m;            // iterate with stepsize 1
     k   = kk;
     i   = (g/m2)%2   == 0 ?  ii : m-1-ii);      // $m^2$ steps
     j   = ((g/m)%2   == 0 ? jj : m-1-jj);       // $m$   steps
  /* if (i,k) changes → send $A_{ik}$
   * if (k,j) changes → send $B_{kj}$
   * FPGA multiplication
   * receive product $A_{ik}B_{kj}$
   * CPU accumulate $C_{ij}$ += $A_{ik}.B_{kj}$
   */
   }
```

Figure 4. Data reuse using Gray code ordering

The nested loops i, j, k of the original program are merged into a single loop g to avoid data stream delays due to loop overhead. The indexes i, j, k are derived from index g such that k remains constant for m^2 successive iterations. Consequently only one matrix A_{ik} or B_{kj} is sent for the iterations (k, i, j) where k is constant, and 2 matrices are sent when k changes, reducing the communication overhead from $2m^3$ to $m^3 + m$ matrix transfers, roughly halving the communication bottleneck. The Gray-code block communication for $m = 3$ is shown in Table 3.

Table 3. Optimized Gray-code ordered block matrix communication for matrix partitioning with m = 3

kij	send	kij	send	kij	send
000	$A_{00}B_{00}$	122	$A_{21}\ B_{12}$	200	$A_{02}\ B_{20}$
001	B_{01}	121	B_{11}	201	B_{21}
002	B_{02}	120	B_{10}	202	B_{22}
012	A_{10}	110	A_{11}	212	A_{12}
011	B_{01}	111	B_{11}	211	B_{21}
010	B_{00}	112	B_{12}	210	B_{20}
020	A_{20}	102	A_{01}	220	A_{22}
021	B_{01}	101	B_{11}	221	B_{21}
022	B_{02}	100	B_{10}	222	B_{22}

As a result, the data reuse in the case with $n = 2108$ and $II = 3$ raises the performance to 4.77 GFLOPS.

5. DISCUSSION

The optimization steps for accelerating the block-oriented floating-point matrix multiplication are shown in Figure 5. The performance trail shows 4 local maxima corresponding to the optimization milestones. First, the combined optimization of DSP and BRAM usage more than doubles the performance, from 1.38 to 4.08 GFLOPS. Remarkably this result is obtained by relaxing the condition on the initiation interval, $II = 1$. Second, widening the bus and streaming the results during the computations reduces the communication overhead and achieves 6.30 GFLOPS. Third, the distribution of a blocked-matrix multiply on the heterogeneous CPU-FPGA multiprocessing system allows increasing the matrix size by one order of magnitude and achieving a performance of 4.19 GFLOPS. Finally, the data reuse of 50% using the Gray-code block ordering increases the performance up to 4.77 GFLOPS for O(2Kx2K) matrices.

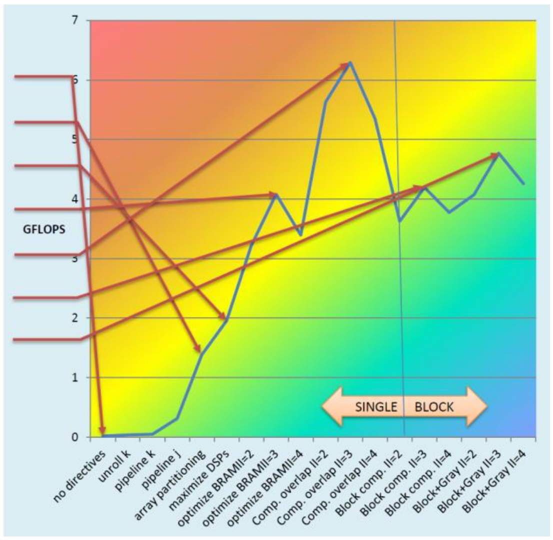

Figure 5. HLS synthesis optimization trail

The experiments were carried out on a ZedBoard incorporating a Zynq XC7Z020 running at 100 MHz. The floating multiply and add operations use 2 and 3 DSPs out of 220, yielding 44 multiply-add units. The PL accelerator operates at 93% efficiency and the CPU-FPGA is able to exploit 4.771/6.295 = 76% of the PL computation speed for 2K by 2K matrix multiplication. It is important to note that none of the optimizations involved any programming in VHDL. The software used is Vivado HLS [7] for the IP design source code with C-style programming directives, and Xilinx SDK for the ARM-processor code using NEON vector instructions and –O2 compiler optimizations. The system layout and supporting IP-cores were designed using the Vivado Design Suite [8].

6. CONCLUSION

The use of FPGAs as compute accelerator has been hampered by the complexity of the design and the lack of supporting tools. Existing low level schemes have been presented which use sophisticated streaming parallel and pipelined architectures. While efficient, these schemes are not easily parameterized to take full advantage of the available resources in a real FPGA. In this paper it is shown that high level synthesis is able to capitalize on all resources by following a simple design strategy which optimizes the combined use of compute power and memory. Furthermore the resulting optimized IP-core is integrated as hardware algorithm to maximize the matrix multiplication of large matrices in a heterogeneous CPU-FPGA SoC. Experiments show that the HLS approach is able to achieve 6.29 GFLOPS on a single PL-based matrix multiply and 4.77 GFLOPS on the combined PS-PL block oriented matrix multiply for large matrices.

7. REFERENCES

[1] Daniele Bagni, A. Di Fresco, J. Noguera and F. M. Vallina 2016. *A Zynq Accelerator for Floating Point Matrix Multiplication Designed with Vivado HLS*. Technical Report #XAPP1170. Xilinx.

[2] Estrin, G. 2002. Reconfigurable computer origins: the UCLA fixed-plus-variable (F+V) structure computer. *IEEE Annals of the History of Computing*. 4 (2002), 3–9.

[3] Jovanović, Ž. and Milutinović, V. 2012. FPGA accelerator for floating-point matrix multiplication. *IET Computers & Digital Techniques*. 6, 4 (2012), 249–256.

[4] Kumar, V.B.Y., Joshi, S., Patkar, S.B. and Narayanan, H. 2010. FPGA Based High Performance Double-Precision Matrix Multiplication. *International Journal of Parallel Programming*. 38, 3–4 (Jun. 2010), 322–338.

[5] Sankar, K.J., Pandharipande, V.M. and Moharir, P.S. 2004. Generalized Gray codes. *Proceedings of 2004 International Symposium on Intelligent Signal Processing and Communication Systems, 2004. ISPACS 2004* (Nov. 2004), 654–659.

[6] Xilinx 2014. *LogiCORE IP Floating-Point Operator v7.0, Product Guide PG060*.

[7] Xilinx 2015. *Vivado Design Suite User Guide: High-Level Synthesis (UG902 2015.1)*.

[8] Xilinx 2015. *Vivado Design Suite User Guide: Synthesis (UG901)*.

[9] Xilinx 2014. *Zynq-7000 Technical Reference Manual UG585 (v1.7)*.

[10] ZedBoard 2013. *ZedBoard Hardware User's Guide*.

[11] Zhuo, L. and Prasanna, V. 2007. Scalable and Modular Algorithms for Floating-Point Matrix Multiplication on Reconfigurable Computing Systems. *IEEE Transactions on Parallel and Distributed Systems*. 18, 4 (Apr. 2007), 433–448.