Optimal iteration scheduling for intra- and inter-tile reuse in nested loop accelerators

Document status and date:
Published: 01/01/2013

Document Version:
Accepted manuscript including changes made at the peer-review stage

Please check the document version of this publication:

• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

[Link to publication](#)

Download date: 17. Nov. 2023

Optimal Iteration Scheduling for Intra- and Inter-Tile Reuse in Nested Loop Accelerators

Maurice Peemen, Bart Mesman, and Henk Corporaal

Optimal Iteration Scheduling for Intra- and Inter-Tile Reuse in Nested Loop Accelerators

Maurice Peemen, Bart Mesman, and Henk Corporaal

Department of Electrical Engineering, Electronic Systems Group

Eindhoven University of Technology, Eindhoven the Netherlands

e-mail: `m.c.j.peemen@tue.nl`, `b.mesman@tue.nl`, `h.corporaal@tue.nl`

December 29, 2013

Abstract

High Level Synthesis tools have reduced accelerator design time. However, a complex scaling problem that remains is the data transfer bottleneck. Accelerators require huge amounts of data and are often limited by interconnect resources. Local buffers can reduce communication by exploiting data reuse, but the data access order has a substantial impact on the amount of reuse that can be utilized. With loop transformations such as interchange and tiling the data access order can be modified. However, for real applications the design space is huge, finding the best set of transformations is often intractable. Therefore, we present a new methodology that minimizes the data transfer by loop interchange and tiling. In contrast to other methods we take inter-tile reuse and loop bounds into account. For real-world applications we show buffer size trade-offs that can give speedups up to 14x, alternatively these can reduce the required FPGA resources substantially.

1   Introduction

For many algorithms, especially in the domain of computer vision and image processing, the compute efficiency can be improved orders of magnitude by using specialized hardware accelerators instead of general purpose processor cores. Recently the relatively long development time of such accelerators, compared to software, is substantially reduced by High-Level Synthesis (HLS) tools. This short development time is of high importance since video coding standards change very often, and new applications are constantly introduced, e.g. Instagram. However, a very complex scaling problem that is not solved by HLS is the data transfer bottleneck of such accelerators. If for example, we consider FPGAs, there is plenty of hardware for parallel compute units, but providing these with the required high-speed data streams is a major challenge.
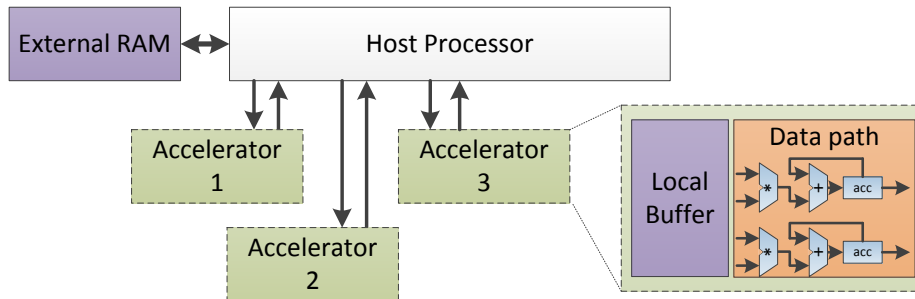
Figure 1: A host processor with accelerators to achieve high compute efficiency by heterogeneity. Data transfer is reduced with local buffers.

Especially in the computer vision and image processing domain the data transfer requirements are challenging, since the high resolution frames or images that must be processed do not fit in on-chip memories. Therefore, these frames are stored in external memory which has limited transfer throughput, and requires much more energy per access compared to on-chip memories. However, data elements that are accessed more than once can be transferred to small on-chip buffers, so a huge number of transfers can be converted into reuses of data. In Figure 1, a schematic overview is given of a heterogeneous architecture that reduces communication with local buffers in the accelerators. The dominant compute workload for accelerators is often a series of nested loops. The data access order of these loops can be altered by loop transformations such as interchange and tiling, which changes the amount of utilized data reuse. Obtaining the best set of transformations is often intractable, due to the huge design space.

Our objective is to provide methods that reduce the huge designer effort which is required to develop efficient hardware accelerators. To the best of our knowledge, this is the first work that provides an accurate and efficient model based solution for tile size optimization for static nested loop accelerators. In contrast to others, our work can perform design space exploration for data reuse and buffer size allocation in seconds instead of hours or days. More specific, we make the following contributions:

- Development of analytical models that take intra- and inter-tile reuse into account. These also include the effect of loop bounds on tile size selection.

- A method to perform quick search space exploration, to obtain the best schedules given a memory bandwidth constraint.

- We show that our technique can be used to equip a simple processor with high performance accelerators.

- We evaluate our technique on real-world applications, and show that huge speedups can be achieved with a modest amount of on-chip buffer size.

The remainder of the paper is organized as follows. Section 3 gives a motivational example for data locality optimization, and Section 4 outlines our iteration reordering models for data reuse optimization. In section 5 the scheduling exploration is discussed, and section 6 outlines the implementation of schedules into an accelerator platform. Section 8 offers an evaluation of our method on real-world applications.

2 Related work

Defining the scheduling of loop iterations which is required to optimize data reuse in hardware controlled memories such as caches is studied for decades [16]. These works often rely on a Polyhedral description [3] of the loop iterations, on which automatic transformations are applied that enhance performance, e.g. Pluto [4], and POCC [13]. These works have a strong emphasis on x86 CPU execution, which is very different from execution on an accelerator. For instance, the avoidance of conditionals in inner most loops is an important objective because these interfere with branch predictors, and prevent vectorization. In contrast, hardware accelerators can handle inner conditionals with a simple multiplexer that can improve resource sharing, which is key for FPGAs.

More related is the Data Transfer and Storage Exploration (DTSE) methodology [5], which focusses on embedded programmable processors with custom memory hierarchies. DTSE uses loop fusion and interchange to improve access locality, and regularity. In contrast to our work, loop tiling is not used because it is platform dependent, and not improving locality across loop nests. However, in deeply nested loops there is often a huge amount of data reuse that can be utilized by loop tiling. Furthermore, during accelerator development the platform is not fixed, e.g. buffer size is an open parameter that should be matched with the tiling strategy.

Recently a new tool is developed that optimizes HLS input descriptions for parallelism and locality [14]. This method uses the polyhedral framework for transformations and uses a HLS tool such as AutoESL to estimate the quality of result. The downside of this approach is its long iteration time; so testing 100 design points can take up to five hours. Secondly, the polyhedral framework generates x86 optimized code with complicated loop bounds resulting in many extra divisions, and min/max operations. In [18] the authors improve the generated output code with a HLS friendly code generator, but the fundamental problem of complex bounds remains.

The work on the Halide compiler [15] also focuses on static image processing and computer vision applications, but for x86 and GPUs. In their work Interval Analysis is used for optimization which is simpler and less expressive compared to the Polyhedral model. Their approach uses an auto tuner based upon stochastic search methods, which takes up to 2 days to converge to good solutions. In [9] the data reuse optimization problem for FPGA hardware is solved by efficient geometric programming. To use geometric programming a simplified data reuse model is used. In contrast to our method, important prop-
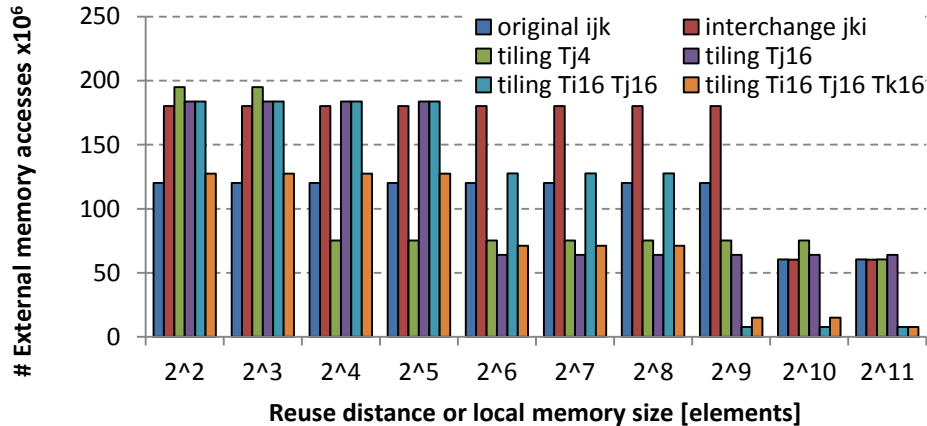
Figure 2: Data transfer histogram for matrix multiply, as given in listing 1. The effect of loop transformations such as interchange and tiling is evaluated on the data transfer.

erties such as overlap between successive tiles is not taken into account.

3 Motivation: Scheduling for data locality

For hardware controlled local memories, such as caches, the reuse distance [7] is a good metric to predict if a value can be reused given a certain chace size. Reuse distance is defined as: *the number of distinctive data elements accessed between two consecutive uses of the same element.* A modification in the iteration order of a loop nest can change the reuse distance of the enclosed array accesses. Therefore, it is possible to change data transfer requirements of an accelerator by reordering loop nest iterations. Important transformations that are used for this purpose are *loop interchange* and *tiling*.

The effect of transformations is demonstrated on matrix multiplication, the corresponding loop nest is given in Listing 1. For simplicity the result matrix C is already initialized to zero, and the sizes of the bounds are set to: Bi=500, Bj=400, Bk=300. The inner loop iterates over k, so each iteration one element of C is reused. In addition, after Bk iterations a row of A is reused. We used *Suggestions for Locality Optimizations* (SLO) [2], a reuse profiling tool, to visualize the remaining data transfers. Remaining transfers are defined as the total number of memory accesses minus the reuses of data elements. Figure 2 shows these remaining transfers for different local buffer sizes. With a very small buffer (2^2 elements) only accesses to C are reused, additionally the elements of A are reused when the buffer increases to (2^{10} elements).

```
for(i=0; i<Bi; i++){
  for(j=0; j<Bj; j++){
    for(k=0; k<Bk; k++){
      C[i][j] += A[i][k] * B[k][j];
```

Listing 1: Nested loop description of a matrix multiplication kernel as running example.

With a loop interchange transformation, loop `i` can be positioned as inner loop, hence reuse of `B` is exploited. However, figure 2 shows that an interchange does not improve but worsens the total reuse, because it also influences accesses to `B` and `C`. One could better perform tiling of loop `j` with factor `Tj=4`, as a result the reuse distance of `A` is reduced to 2^4 entries. Tiling can also be performed in other directions and with different factors. Listing 2 demonstrates tiling in all three dimensions. Further experiments with different tile factors on multiple loops, reveal that obtaining the best configuration for a buffer size is a very intricate problem. Even worse is the huge difference in data transfers for different configurations, i.e. the design space is very chaotic. For example, loop tiling with `Ti=16` and `Tj=16` gives excellent results for a buffer size of 2^9, but for 2^8 it is one of the worst schedules. If the designer could find the best schedules a huge reduction in the number of communications can be achieved, at the cost of a modest amount of buffer area.

```
for(ii=0; ii<Bi; ii+=Ti){
  for(jj=0; jj<Bj; jj+=Tj){
    for(kk=0; kk<Bk; kk+=Tk){
      for(i=ii; i<ii+Ti; i++){
        for(j=jj; j<jj+Tj; j++){
          for(k=kk; k<kk+Tk; k++){
            C[i][j] += A[i][k] * B[k][j];
```

Listing 2: Loop tiling to transfer parts of loop i,j,k, to the inner loop.

4 Modeling The Scheduling Space

To obtain the best tiling and interchange transformations for a loop nest we formulate an optimization problem. Hence, a cost function that represents the number of external transfers is used. In addition, a bounding function is used to limit the required buffer size.

4.1 Modeling intra-tile reuse

For the cost function we assume that a loop nest is split into two parts; an inner part (zero or more loops) for execution on the accelerator, and outer part that runs on a host processor. The outer part facilitates the data transfer between the external memory and the accelerator. The inner part uses this data to perform computations, which results in temporal or output results that are transferred back to the host. *Our cost function represents the number of transfers to and from the accelerator.* For the loop nest in Listing 2 the cost function is given

below:

$$N\text{tiles} * (\text{datatransfer/tile}) = \left\lceil \frac{B_i}{T_i} \right\rceil \left\lceil \frac{B_j}{T_j} \right\rceil \left\lceil \frac{B_k}{T_k} \right\rceil (2T_iT_j + T_iT_k + T_kT_j) \quad (1)$$

The first part represents the *number of tiles* that is modeled by dividing the domain of each loop by the corresponding tile factor. When modeling the number of tiles it is important to take loop bounds into account, especially for nested loops. This is mainly due to the fact that loop bounds are more often visited when they are part of a loop nest. If the tile factor is not a divisor to the loop bound; an extra check should be used on the inner loop, or dummy data values must be used that increase the bound. The first option increases the control complexity of the accelerator, but the second increases data transfer. We use the second option, and take the effect into account in the cost function by ceiling the number of tiles in each dimension. As a result, the cost function favors tile factors that are divisors of the loop bounds, which gives the best of both worlds. The second part of eq. (1) represents the *data transfer per tile*, which depends on the array references that are used for reading and writing, and the size of the tile. The term $2T_iT_j$, models the reads and writes to array C, because array A is only read it is modeled by T_iT_j.

Valid tile factors that fit in a buffer size constraint are obtained by using a buffer requirement model as a constraint. The buffer requirement is modeled as the number of distinct array elements accessed in an inner tile. For Listing 2 this results in expression (2). The selected tile factors with the array indices decide on the data content of the inner tile.

$$T_iT_j + T_iT_k + T_kT_j \leq [\text{Buffer size}] \quad (2)$$

4.2 Adding inter-tile reuse to the model

For a simple accelerator that overwrites the buffer content after processing of a tile, the model described in Section 4.1 is correct. However, we can do much better; if we want to exploit inter-tile reuse we should exploit knowledge about the data contents of the next tile. This increases complexity because the accelerator must initialize the first tile of a series (prolog), compute successive tiles with data overlap (steady state), and correctly compute the last tile of a series (epilog). Furthermore, dependencies are created between successive tiles that reduce inter-tile parallelism. Nevertheless, inter-tile reuse can substantially reduce data transfer, which is a key issue affecting the performance of many applications.

In figure 3a an example is depicted, which visualizes data transfer for matrix multiplication. Optimizing for intra-tile reuse with a buffer size constraint of 32 elements results in the tile factors Ti=3, Tj=3, Tk=3. Without inter-tile reuse the host would send 27 values (3x3 patch of A, B, C), and receive 9 values (3x3 patch of C) for every tile. On the other hand, if data overlap of successive tiles is exploited, only 18 values (3x3 patch of A and B) are transferred in the steady
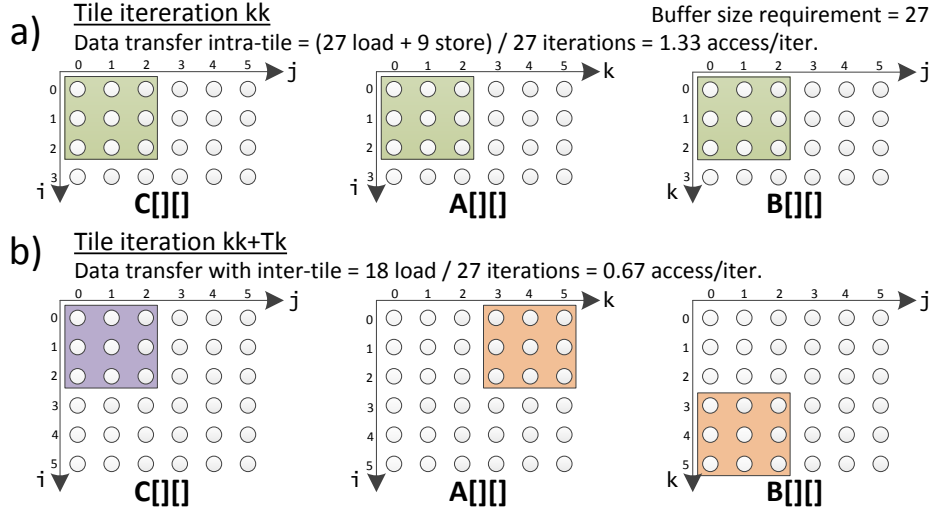
Figure 3: Data access pattern for two successive tiles for matrix multiplication, based on listing 2. a) Data transfer for intra-tile reuse requires all accessed elements. b) With inter-tile reuse elements of C can be reused, which gives a 2x reduction of data transfer.

state tiles. As depicted in figure 3b the data of C can be reused. Hence, the data transfer is reduced by 2x excluding each first and last tile of control loop kk.

The cost function of Section 4.1 does not take inter-tile reuse into account and therefore it gives suboptimal results regarding data transfer. The key observation that opens opportunities to find even better schedules is that tiling of the inner control loop has no influence on inter-tile reuse. For example, if Tk=1 in figure 3b the of reuse of C does not change, but the memory footprint reduces. As a result, not tiling the inner control loop opens opportunities in other dimensions to increase reuse. The data transfer effects with inter-tile reuse are modeled by an expression similar to (1), but considering the full range of the inner control loop as one tile. Hence, the transfers of the prolog, steady state, and epilog are included in the model. Equation (3) shows the updated data transfer model with kk as inner control loop.

$$\left\lceil \frac{B_i}{T_i} \right\rceil \left\lceil \frac{B_j}{T_j} \right\rceil (T_i T_j + T_i B_k + B_k T_j) \tag{3}$$

The key difference is that the buffer requirement constraint, as shown in eq. (2), does not change. The tile factor of the inner control loop Tk is set to 1, because it is not tiled. Hence, one dimension can get all available reuse for free.

A different inner control loop influences the amount of data overlap between successive tiles. In the example of Listing 2 the inner control loop is kk. Hence,
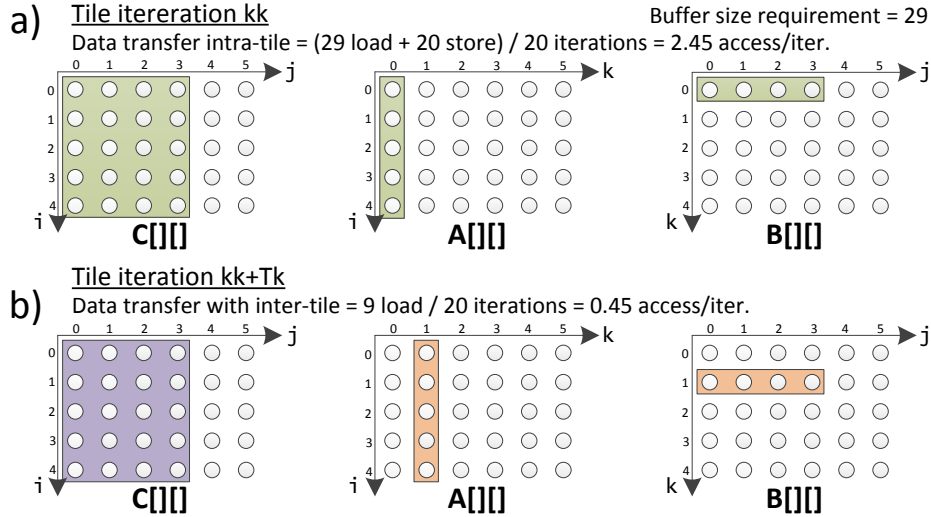
Figure 4: Data access patterns for matrix multiply, based on listing 2. Compared to figure 3 this pattern is optimized for inter-tile reuse. a) Intra-tile reuse is less compared to figure 3a. b) Inter-tile reuse is more than figure 3b; with a tile factor Tk=1 there is more reuse of C with an almost similar buffer size.

C[i][j] is reused and the send and receive transfers for array C are minimized. By loop interchange jj can be the inner control loop, and instead send transfers for array A are minimized. The corresponding model is given below:

$$\left\lceil \frac{B_i}{T_i} \right\rceil \left\lceil \frac{B_k}{T_k} \right\rceil (2T_iB_j + T_iT_k + T_kB_j) \tag{4}$$

With the improved models it is possible to find better schedules for minimal data transfer. In Figure 4a the optimal schedule for matrix multiplication with a buffer size constraint of 32 elements is visualized. The inner control loop is kk, and the tile factors are Ti=5, Tj=4, Tk=1. Similar to Figure 3a it outlines the communication requirement for only intra-tile reuse. As expected, the new schedule performs worse due to the inter-tile optimization target. However, if we compare the data transfer requirement with inter-tile reuse a reduction of 1.48x per compute iteration is achieved over the schedule of Figure 3b. As demonstrated by this matrix multiplication example, the effects of inter tile reuse must be taken into account when optimizing the iteration order. If not, a sub-optimal solution will be obtained.

5   Scheduling Space Exploration

When considering transformations such as interchange and tiling on deep nested loops the scheduling space can be huge, as shown in Section 3. We use the

models proposed in Section 4 to obtain the best schedules within seconds instead of hours or days as reported by other search methods [14, 15]. This target is achieved by using analytical models that can be evaluated quickly, and using inter-tile reuse optimization, which prunes the search space. Our approach is outlined with a simple convolution kernel, shown in Listing 3. The loop bounds of the example nesting are `Bi`=50 and `Bj`=100.

```
for(i=0; i<Bi; i++){
  for(j=0; j<Bj; j++){
    Out[i] += X[i+j] * H[j];
```

Listing 3: Nested loop description for convolution.

For *intra-tile* optimization different tiling factors should be explored e.g., combinations of `Ti` for loop `i`, and `Tj` for loop `j` that fit the constraints. The search space for this problem is depicted in Figure 5. However, for *inter-tile* optimization the search space is much smaller, so one loop e.g. `ii` is selected as inner control loop and for the other loop `j`, tile sizes are evaluated. This is also done with the other option `jj` as control loop. Essentially one dimension of the search space is removed. The corresponding cost functions are given in eq. (5), and the buffer size constraint is eq. (6).

$$\text{Cost} = \begin{cases} \left\lceil \frac{B_i}{T_i} \right\rceil (T_i + (T_i + B_j - 1) + B_j) & \text{Tj}=1 \\ \left\lceil \frac{B_j}{T_j} \right\rceil (2B_i + (B_i + T_j - 1) + T_j) & \text{Ti}=1 \end{cases} \tag{5}$$

$$T_i + (T_i + T_j - 1) + T_j \leq [\text{Buffer size}] \tag{6}$$

The search for the best configuration is performed by a bounded search through the valid solution space. Since the buffer size requirement function is monotonic, the bounds on the valid solution space can be efficiently set with the guarantee that optimal solutions are obtained. This search method is visualized in Figure 5. Important to note is the short search time that is required to obtain the best schedules for very deep nested loops. On a standard laptop the search space for our 8-level deep motion estimation kernel is explored in 4.2 seconds, with a buffer size constraint of 1024 elements.

6 Implementation

An optimized schedule must be converted to host processor code and an HLS accelerator description according to the template in figure 1. The required conversions are outlined by the matrix multiplication example with a buffer size constraint of 32 entries. The optimal schedule, derived from Listing 2 and inter-tile reuse optimization, has tiling parameters `Ti`=5, `Tj`=4, and the inner control loop is `kk`. The resulting data access pattern is depicted in figure 4b.

The host processor code executes the outer control loops, and takes care of data transfers between host and accelerator. Basically, the outer control loops of listing 2 are used. Furthermore the prolog and epilog parts are inserted, which
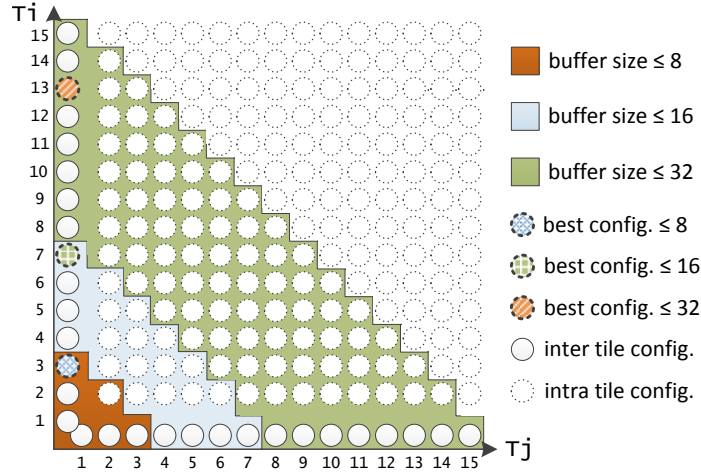
Figure 5: Design space for tiling configurations on the convolution code in listing 3. Each axis represent a possible tile factor as parameter. For inter-tile reuse optimization one of the tile factors must equal one, which prunes the space. Two options remain `Ti`=1 or `Tj`=1 the other parameter must be optimized. The best configuration given a buffer size, is not always located on the border as shown for a buffer constraint of 32.

transfer input data and output results, respectively. Finally, the inner control loop is inserted, which transfers the steady state data chunks. In Listing 4, a description of the host code is given. Note that the Send and Receive functions facilitate FIFO based communication. For the connection between the host and accelerator we use the Fast Simplex Links (FSLs) from Xilinx.

```
for(ii=0; ii<Bi; ii+=Ti){
  for(jj=0; jj<Bj; jj+=Tj){
    //prolog part nothing to send
    for(k=0; k<Bk; k++){ //steady state
      Send(A[ii:ii+Ti-1][k]);
      Send(B[k][jj:jj+Tj-1]);
    }
    //epilog part receive results
    Receive(C[ii:ii+Ti-1][jj:jj+Tj-1]);
} }
```

Listing 4: Host processor code that corresponds to the matrix multiply example of listing 2. This code performs data transfer by executing the outer control loops.

The accelerator code performs the content of the tile loops, which is the main compute workload. It has no notion of the position in the program; it just repeats execution of streams with overlapping tiles. Furthermore, it describes the read/store policy in the local buffers. The prolog and epilog parts are specified, and in addition the steady state inner control loop is inserted. This last part

10

contains a data transfer and a compute part. If desired, the compute part can be parallelized by adding HLS specific pragmas for pipelining or unrolling [17]. Listing 5 gives of the accelerator code:

```
Init(C[0:Ti-1][0:Tj-1]); //prolog
for (k=0; k<Bk; k++){ //steady state
  Receive(A[0:Ti-1]);
  Receive(B[0:Tj-1]);
  for(i=0; i<Ti; i++){
    for(j=0; j<Tj; j++){
      C[i][j]+=A[i]*B[j];
} } }
Send(C[0:Ti-1][0:Tj-1]); //epilog return results
```

Listing 5: Accelerator code, which computes on incoming data by executing inner loops of the matrix multiply example.

7 Evaluation Methodology

To evaluate our optimized accelerators, we map a representative set of real-world applications. We focus on popular embedded applications, with extensive data transfer requirements, in the image and video processing domain. As outlined these applications should contain static deep nested loops that represent the major compute workload. A short overview of the applications is given below, and in addition their sources are made available on the web: `http://parse.ele.tue.nl/research/tools`.

7.1 Benchmark Applications

7.1.1 Demosaicing

Camera processing pipelines typically require a demosaicing step, since the red, green, and blue (RGB) channels of the sensor are laid out in a Bayer [1] pattern. A demosaicing algorithm interpolates the two missing color values, at each pixel position. However, interpolation is difficult because the color channels have an inadequate sampling resolution, which causes color artifacts. We use a 5x5 position adaptive interpolation kernel based upon the Malvar-He-Cutler [10] method. Furthermore, an 8 Mpixel input image is used for realistic data transfer figures.

```
for(y=0; y<By; y++){
  for(x=0; x<Bx; x++){
    for(c=0; c<Bc; c++){
      for(k=0; k<Bk; k++){
        for(l=0; l<Bl; l++){
          Out[y][x][c] += In[y+k][x+l] *
                                W[y&1][x&1][c][k][l];
```

Listing 6: Pseudo description of the Malvar method for demosaicing

7.1.2 Motion Estimation

An important step in video coding is Motion Estimation; since it significantly improves compression, though substantially increasing complexity. In modern coding standards, such as H.264, the Integer Motion Estimation (IME) step represents 78% of the compute workload, and 78% of the memory accesses [6]. We use a full-search block matching kernel with a window of 32x32 that searches in a previous and future reference frame for the best matching block, using the Sum of Absolute Differences (SAD) cost function. Furthermore, there are four HD 720p frames between two reference frames that must be encoded by motion vectors. As outlined in Listing 7 the algorithm can be described by a very deep loop nest, with reuse opportunities in all dimensions.

```
for(f=0; f<Bf; f++){ //encoded frame nr.
  for(by=0; by<Bby; by++){ //macro block
    for(bx=0; bx<Bbx; bx++){
      for(r=0; r<Br; r++){ //reference frame
        for(sy=0; sy<Bsy; sy++){ //search window
          for(sx=0; sx<Bsx; sx++){
            for(y=0; y<By; y++){ //block difference
              for(x=0; x<Bx; x++){
                diff += abs(in[f][by,y][bx,x] -
                        ref[r][by,sy,y][bx,sx,x]);
```

Listing 7: Pseudo description of Integer Motion Estimation (IME) for a video coding application

7.1.3 Object Recognition

Presently, an increasing number of embedded devices use object detection and recognition, e.g. face detection in photo cameras, and speed sign recognition in navigation devices [12]. A flexible and robust algorithm for these tasks is a Convolutional Neural Network (CNN) [8]. In our evaluation we use a speed sign recognition application on a 720p video stream. Specifically, we used an optimized version with merged feature extraction layers for better locality, based upon [11].

7.2 Platform and tools

As a reference for evaluation fixed-point versions of the applications are mapped to three different platforms:

1. Intel Core-i7 960 CPU at 3.2GHz

2. Arm-A9 CPU at 667MHz, Xilinx Zynq SoC

3. MicroBlaze configured for performance, at 200MHz

In addition, our methodology is used to develop hardware accelerators that increase the performance of a MicroBlaze host processor. We synthesize our designs for the Xilinx ML605 board, which has one Virtex-6 FPGA (xc6vlx240t-1ffg1156). For development we use the Xilinx Vivado 2012.3 tools, including

Vivado HLS (AutoESL), which is used to create accelerators. The clock frequency of our designs is set to 200 MHz.

8   Results

To quantify the effectiveness of our inter-tile reuse optimization strategy we firstly analyze the data transfer effects for the three benchmark applications. So, for each application the cost functions and buffer size requirements are derived. With these descriptions we performed the scheduling space exploration for different buffer size constraints, as outlined in section 5.

8.1   Inter-tile reuse optimization

Figure 6 shows the data transfer requirements for the original iteration ordering, and three other optimization strategies. The amount of data transfer is specified as a factor with respect to the theoretical minimum, i.e. communicating each input and final output only once. This can always be achieved with an infinite buffer size. The communication volume is plotted against the required buffer size, which should be as small as possible.

Intra-tile optimization shows the result for an accelerator that resets the buffer contents after each tile, as discussed in section 4.1. This is a naïve optimization target; as a result it sometimes gives worse results compared to the original iteration order. If we exploit the available inter-tile reuse for schedules first optimized for intra-tile reuse the communication or the required buffer size is significantly reduced. Finally, the schedule is optimized for inter-tile reuse, which for all benchmarks results in the least amount of communication at the smallest buffer size.

Important to note is the huge data transfer reduction. E.g. for the motion estimation benchmark the best schedule can reduce data transfer up to 50x compared to the original. Furthermore, we demonstrate that a relatively small local buffer of 1024 elements can substantially reduces data transfer. For motion estimation and object recognition the remaining number of transfers is within one order of magnitude of the minimum. However, for demosaicing the minimum is already reached with a buffer of 512 elements. A designer should stop increasing buffer size after this point, because the amount of data transfer stays constant and it only increases the area footprint.

8.2   Matching compute and buffer resources

As demonstrated, there is a trade-off between buffer size and communication requirements for hardware accelerators. In addition, a designer can change the amount of compute resources, which also influences the communication requirements. For HLS descriptions this can be achieved by unrolling a loop of the algorithm. After unrolling, the data path that performs the operation is replicated resulting in extra hardware that exploits parallelism in the algorithm. To
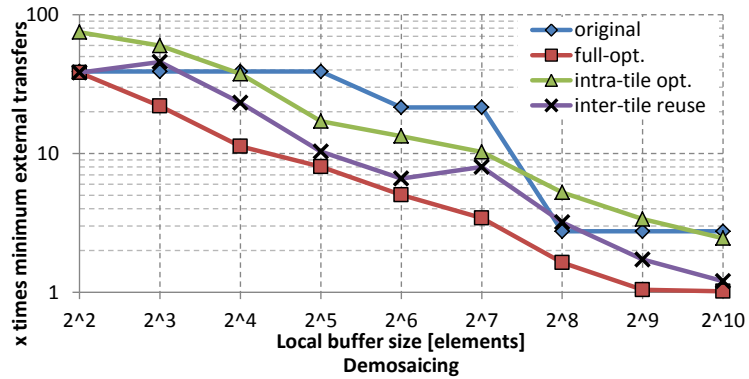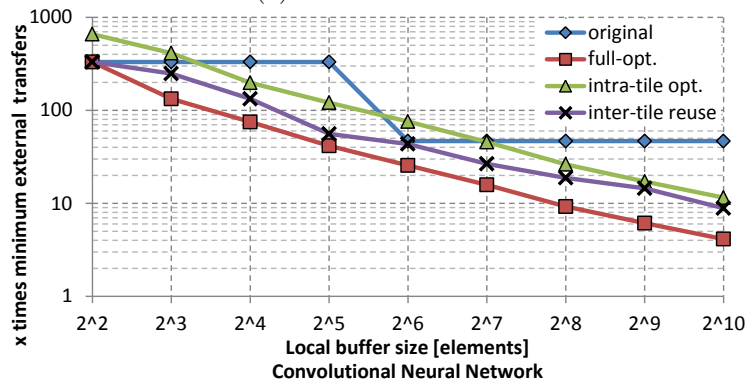
(a) Demosaicing
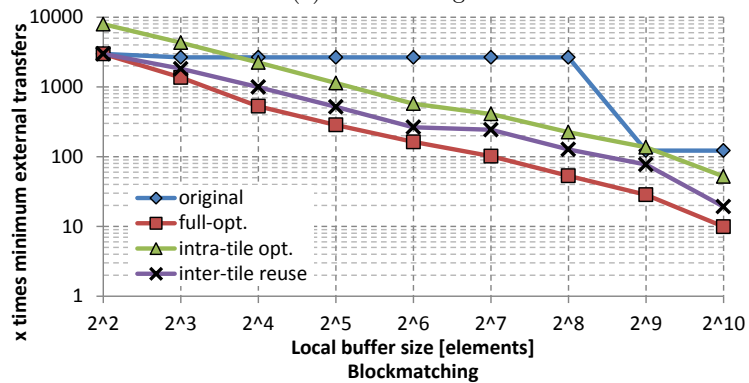
(b) Motion Estimation

(c) Object Recognition

Figure 6: External data transfers versus accelerator buffer size, for multiple schedules. These are, the original ordering, intra-tile optimization, intra-tile optimization while enabling inter-tile reuse, and inter-tile reuse optimization or full optimization.

Table 1: Data transfer and compute time for the evaluation platform

Workload	Read	Write	Demosaic	Blockmatch	CNN
Throughput [MB/s]	29.9	43.2	39.9	30.0	40.9
Compute 1PE [s]	-	-	3.2	37.9	4.3
Transfer 128entry [s]	-	-	2.8	19.3	2.0

utilize this hardware the buffer size can be increased such that more data is reused. In an efficient accelerator design, buffer and compute resources are in balance. If not, more resources are required than necessary, because performance is limited by either data transfer or compute time.

In figure 7 the measured execution time for different accelerator buffer sizes and degrees of parallelism is shown. We observe e.g. that accelerators with a buffer size of 32 elements are bandwidth limited for all three benchmark applications. These small designs do not scale if more compute resources are used. On the other hand, if only more and more buffer resources are used the execution time saturates as well. In this situation more parallelism is required because the accelerator is compute bound.

With a few simple streaming communication tests the available transfer bandwidth can be measured. This enables designers to estimate communication time that should be balanced with compute time for hardware efficient design points. By using a MicroBlaze host processor we transferred 32 MB of data to and from the accelerator to measure the transfer throughput. For each benchmark the best case transfer throughput is estimated with read/write ratio to external memory, as given in table 1. E.g. demosaicing has more writes than reads and therefore is close to the write speed. A designer can use this information to quickly discover the interesting regions of the design space. For example, a single PE design would not require a buffer size much larger than 128 entries for each of the benchmarks. A buffer size over 128 entries would only make sense if the parallelism is increased by unrolling.

8.3   Quality of results

The quality of our implemented schedules is evaluated by comparing execution time with resource usage. For FPGAs, resource usage is defined as: *MAX(%DSPs, %BRAMs, %LUTs, %flip-flops)*. In figure 8 the execution time versus resource usage is depicted by plotting the design points of figure 7. In addition, the area execution time product is used to define the efficiency of different implementations. For each benchmark the best area execution time product is extrapolated and plotted as a dotted line in the comparison.
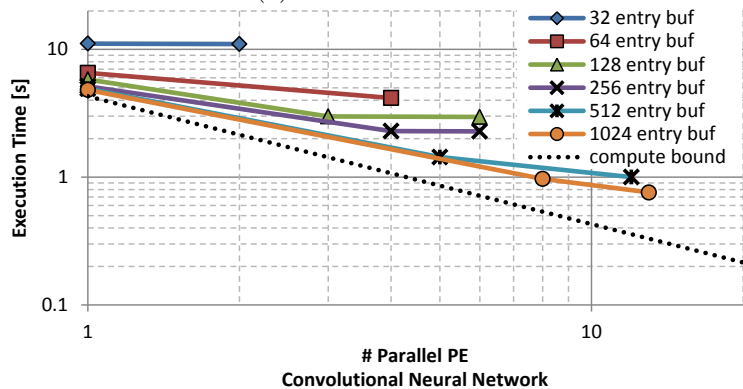
The results of demosaicing and object recognition behave intuitive. Designs that are balanced score close to the dotted line, while designs severely limited by either compute or buffer resources occur further from this line. As already predicted for demosaicing in figure 6, increasing the buffer size beyond 512
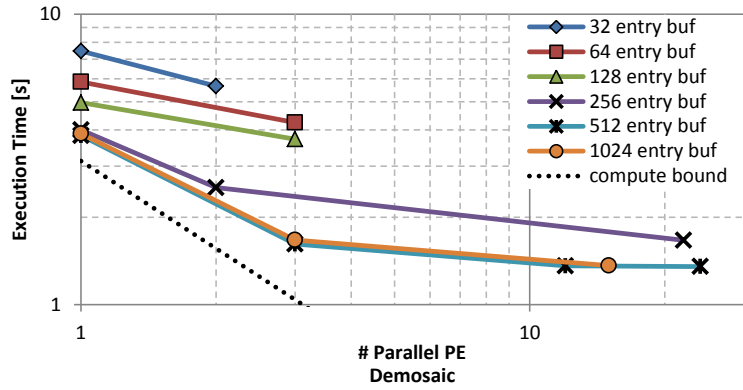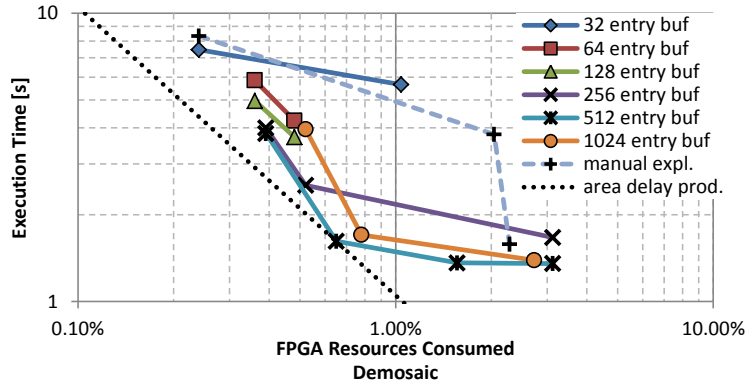
(a) Demosaicing
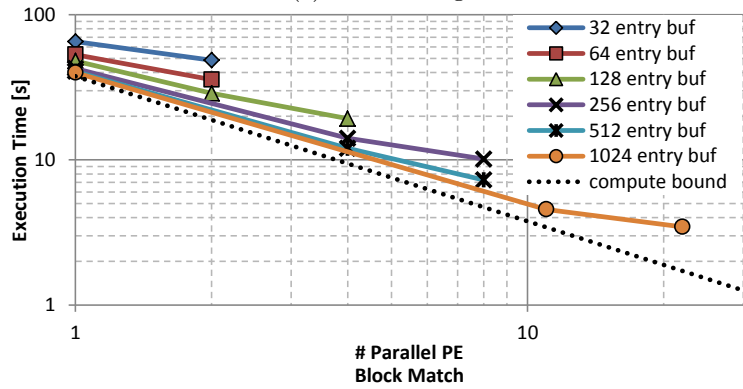

(b) Motion Estimation

(c) Object Recognition

Figure 7: Execution time of different buffer size constraints versus parallelism in the accelerator, which is created by unrolling in the HLS accelerator description. The dotted line represents the execution time without data transfer. As figure 6 showed, demosaicing exploits all reuse with a 512 entry buffer.

16

(a) Demosaicing
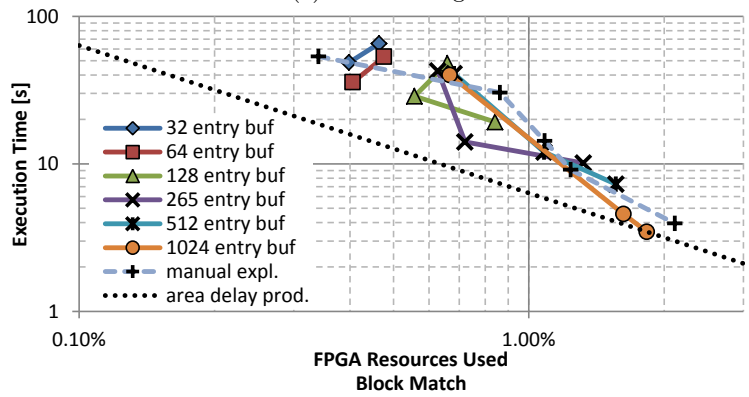
(b) Motion Estimation
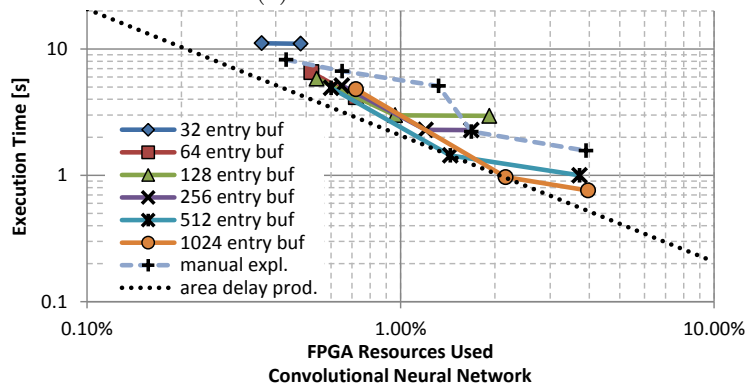
(c) Object Recognition

Figure 8: FPGA resource utilization for accelerator mappings with an optimal iteration order. The most efficient solution is extrapolated by the dotted line as the area delay product.

Table 2: Execution time comparison for multiple platforms

Platform	Demosaic [s]	Block Match [s]	CNN [s]
Intel-i7	0.54	8.12	0.63
Arm-A9	5.75	72.32	5.92
MicroBlaze	22.10	283.96	19.05
Accelerator	1.36	3.45	0.75

entries does not improve the design any further. However, for motion estimation the results behave different, e.g. increasing parallelism in a data transfer bounded designs can improve efficiency. Important to note is that Vivado HLS is a production tool, therefore a small change in the input description can trigger different optimizations. Due to unrolling with a factor two the number of required LUTs in the 32 and 64 entry designs are reduced. This reduces overall resource usage and the area delay product.

Finally, we compared the best accelerators with other platforms, as shown in table 2. The accelerators do not have a dedicated DMA controller and are therefore constraint by communication bandwidth. However, the accelerators can increase the original MicroBlaze performance by 16 to 82 times, at the cost of a very small increase of overall resource usage. As a result, a very simple and efficient embedded processor can perform on par with a high-end general purpose processor. Dedicated DMA can be added, but it will only shift the result of figure 8.

9   Conclusion

In this paper we demonstrated that efficient usage of local buffers in FPGA based accelerators can give substantial performance improvements. These improvements are driven by maximizing efficiency in the local buffers with data access optimizations for nested loops. Due to the efficiency increase only a modest amount of buffer size is required to achieve a huge reduction of external data transfer. The main improvement is achieved by optimizing for inter-tile reuse with loop transformations such as interchange and tiling. Although the design space can be huge and chaotic for very deep nested loops, we show that the best configuration of transformations can be found with an analytical model based solution. With our focus on transformations that are good for inter-tile reuse, the design space can be pruned which reduces exploration time to the order of seconds. By using the results of our optimization method with the Xilinx Vivado HLS tools, the huge designer effort to develop efficient hardware accelerators is significantly reduced. With our approach the number of required design iterations is minimized, by directly computing the best candidates before time consuming synthesis. As a result, the mapping process of static image or video processing applications to dedicated hardware accelerators is much better manageable.

References

[1] B. Bayer. Color imaging array, 1976.

[2] K. Beyls and E. D'Hollander. Refactoring for data locality. *Computer*, 42(2):62–71, feb. 2009.

[3] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. *SIGPLAN Not.*, 43, 2008.

[4] U. Bondhugula, J. Ramanujam, and P. Sadayappan. Pluto: A practical and fully automatic polyhedral program optimization system. In *ACM SIGPLAN (PLDI)*, 2008.

[5] F. Catthoor, K. Danckaert, C. Kulkarni, E. Brockmeyer, P. G. Kjeldsberg, T. Van Achteren, and T. Omnes. *Data access and storage management for embedded programmable processors*. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[6] T.-C. Chen, S.-Y. Chien, Y.-W. Huang, C.-H. Tsai, C.-Y. Chen, T.-W. Chen, and L.-G. Chen. Analysis and architecture design of an hdtv720p 30 frames/s h.264/avc encoder. *Circuits and Systems for Video Technology, IEEE Transactions on*, 16:673–688, 2006.

[7] C. Ding and Y. Zhong. Reuse distance analysis. Technical report, University of Rochester, 2001.

[8] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86:2278–2324, 1998.

[9] Q. Liu, G. Constantinides, K. Masselos, and P. Cheung. Combining data reuse with data-level parallelization for fpga-targeted hardware compilation: A geometric programming framework. *Computer-Aided Design of Integrated Circuits and Systems*, 28(3):305–315, 2009.

[10] H. Malvar, L.-W. He, and R. Cutler. High-quality linear interpolation for demosaicing of bayer-patterned color images. In *Proceedings IEEE International Conference on Acoustics, Speech, and Signal Processing,*, volume 3, pages iii–485–8, 2004.

[11] M. Peemen, B. Mesman, and H. Corporaal. Efficiency optimization of trainable feature extractors for a consumer platform. In *ACIVS*, pages 293–304, 2011.

[12] M. Peemen, B. Mesman, and H. Corporaal. Speed sign detection and recognition by convolutional neural networks. In *8th International Automotive Congress*, pages 162–170, 2011.

[13] L.-N. Pouchet, U. Bondhugula, C. Bastoul, A. Cohen, J. Ramanujam, P. Sadayappan, and N. Vasilache. Loop transformations: convexity, pruning and optimization. In *Proceedings of the 38th ACM SIGPLAN-SIGACT symposium on POPL*, pages 549–562, 2011.

[14] L.-N. Pouchet, P. Zhang, P. Sadayappan, and J. Cong. Polyhedral-based data reuse optimization for configurable computing. In *Proceedings of the ACM/SIGDA international symposium on FPGA*, pages 29–38, 2013.

[15] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN conference on (PLDI)*, pages 519–530, 2013.

[16] M. Wolf and M. Lam. A data locality optimizing algorithm. In *Proc. PLDI '91*, pages 30–44, 1991.

[17] Xilinx. *Vivado Design Suite User Guide, High-Level Synthesis, UG902*.

[18] W. Zuo, P. Li, D. Chen, L.-N. Pouchet, S. Zhong, and J. Cong. Improving polyhedral code generation for high-level synthesis. In *Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 1–10, 2013.