A Data Re-use Based Compiler Optimization for FPGAs[⋆]

Ram Subramanian[⋆⋆] and Santosh Pande

[1] Xilinx Inc., 2100 Logic Dr., San Jose, CA 95124,
ram@xilinx.com
[2] College of Computing, Georgia Institute of Technology, Atlanta, GA 30332,
santosh@cc.gatech.edu

1 Our Approach

The speed at which a design could be tested (executed) really determines the use of FPGAs for rapid prototying. FPGAs provide reasonable routing resources, and a high capacity for mapping large hardware designs. However, profitable mapping of computations onto FPGAs is a complex task due to many trade-offs involved. We present an approach to customize FPGA-based co-processors to most profitably execute loops to speed-up the the execution. Our framework specifically addresses the issues of parallelism, reducing data transfer overheads through reuse, and optimizing the safe frequency at which design can be maximally clocked.

1.1 Motivating Example

We first illustrate our approach through an example and then present the framework developed.

Consider the example of a simple matrix multiplication code as shown below.

```
for I = 1 to 100
  for J = 1 to 100
    for K = 1 to 100
      C[I,J] = C[I,J] + A[I,K] * B[K,J];
```

As seen from the loop, we can exploit self-temporal data reuse for arrays A and B, since for all iterations of index J, A[I,K] is a constant, and similarly for all iterations of index I, B[K,J] is a constant. The loop thus has a self-temporal reuse factor of 100 for both indices I and J. Moreover, only loop K carries the dependency of C[I,J], and the variable is called a *reduction* variable since the same array location is being read and written into for all iterations of loop K. Suppose we unroll the inner loop by k and expand the reduction variable. The transformed loop is given below:

[⋆] This research was supported in part by DARPA award # ARMY DABT63–97–C–0029 and by NSF grant # CCR-0073512

[⋆⋆] Work done as a part of Masters' thesis.

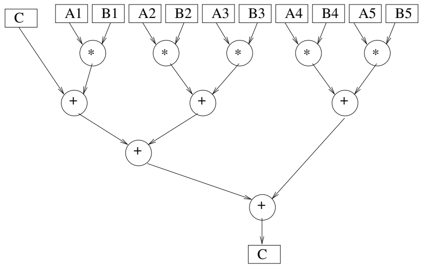

Fig. 1. Example loop unrolled five times

```
for I = 1 to 100
  for J = 1 to 100
    for K = 1 to 100 by k
      C[I,J] += A[I,K] * B[K,J]
           + ...
           + A[I,K+k-1] * B[K+k-1,J];
```

The computation graph for the statement inside the loop for $k = 5$ is given in Figure 1. Similarly for any k, a parallel computation graph is created. This graph has the nice property of directly translating to hardware, where the rectangular modules are registers and the circular modules are hardware implementations of the operator and the arrows are input and output buses connecting the operators and registers. For any other k, the graph changes in width and depth. As the depth increases, the maximum register to register latency also increases. As the width increases, so does the number of operands to be brought into hardware before the start of computations. As seen in the figure, registers $A1, A2 \ldots, A5$ and registers $B1, B2, \ldots, B5$ take values of array `A` and `B` as given by the iterations. The input register `C` takes the value of the previous `C` result, and the output register `C` is stored into the memory as soon as the computation is done in hardware.

The array variable `C` is a reduction variable, carried by loop `K`. The loop nest is found to be a *fully permutable* nest, with loop `K` carrying a *doall reduction.* A fully permutable loop nest means that the loops in the nest can be legally interchanged without affecting the data dependencies. This means that if a reduction operation is carried out, the loop can be parallelized. Here, loop `K` is partially unrolled to carry out the reduction operation partially on hardware. Thus, the loop can be transformed and mapped on the hardware as follows:

```
for I = 1 to 100
  for K = 1 to 100 by k
    for J = 1 to 100 by j in parallel
      /* Statement S1 */
      C[I,J] = C[I,J] + A[I,K] * B[K,J]
          + ...
```