FPGA-Based Hardware Acceleration of Lithographic Aerial Image Simulation

JASON CONG and YI ZOU
University of California, Los Angeles

Lithography simulation, an essential step in design for manufacturability (DFM), is still far from computationally efficient. Most leading companies use large clusters of server computers to achieve acceptable turn-around time. Thus coprocessor acceleration is very attractive for obtaining increased computational performance with a reduced power consumption. This article describes the implementation of a customized accelerator on FPGA using a polygon-based simulation model. An application-specific memory partitioning scheme is designed to meet the bandwidth requirements for a large number of processing elements. Deep loop pipelining and ping-pong buffer based function block pipelining are also implemented in our design. Initial results show a 15X speedup versus the software implementation running on a microprocessor, and more speedup is expected via further performance tuning. The implementation also leverages state-of-art C-to-RTL synthesis tools. At the same time, we also identify the need for manual architecture-level exploration for parallel implementations. Moreover, we implement the algorithm on NVIDIA GPUs using the CUDA programming environment, and provide some useful comparisons for different kinds of accelerators.

Categories and Subject Descriptors: B.7.1 [**Integrated Circuits**]: Types and Design Styles—*Algorithms implemented in hardware*

General Terms: Algorithms, Performance, Design

Additional Key Words and Phrases: Lithography simulation, coprocessor acceleration, FPGA

ACM Reference Format:
Cong, J. and Zou, Y. 2009. FPGA-based hardware acceleration of lithographic aerial image simulation. ACM Trans. Reconfig. Techn. Syst. 2, 3, Article 17 (September 2009), 29 pages.
DOI = 10.1145/1575774.1575776. http://doi.acm.org/10.1145/1575774.1575776.

This work is partially funded by Natural Science Foundation CNS-0725354, a grant from the Chinese Natural Science Foundation for International Collaboration, and also a grant from Altera Corporation and Magma Design Automation under the California MICRO Program.
J. Cong served as the chief Technology Advisor of Magma Design Automation from 2004 to 2008.
Authors' address: J. Cong and Y. Zou, Computer Science Department, University of California, Los Angeles, Los Angeles, CA 90095; email: {cong, zouyi}@cs.ucla.edu.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2009 ACM 1936-7406/2009/09-ART17 $10.00 DOI: 10.1145/1575774.1575776.
http://doi.acm.org/10.1145/1575774.1575776.

1. INTRODUCTION

Optical lithography is the technology used for printing circuit patterns onto wafers. As the technology scales down, and the feature size is even smaller than the wavelength of the light employed (e.g., 193nm lithography), significant light interference and diffraction may occur during the imaging process. Lithography simulation, which tries to simulate the imaging process or the whole lithography process—from illumination to mask to imaging to resist—is considered an essential technique for the emerging field of DFM.

Lithography simulation can be done through various methods with different accuracy. Model- or rule-based optical proximity correction (OPC) uses empirical rules and models from experimental data to perform the simulation and discover the defects caused by lithography [Mack 2005]. It is fast but not accurate enough. On the other hand, using finite difference or finite element methods to solve the corresponding electromagnetic equations directly [Yeung 2003] is a very accurate approach, but is so expensive that it can only simulate small regions and designs.

The coherent decomposition method [Pati and Kailath 1994] can better balance the accuracy and running time, and is the main method used in computational lithography for large designs. It first decomposes the whole optical imaging system into many coherent systems with decreasing importance. The image corresponding to each coherent system can be obtained via numerical image convolution, and the final image is the weighted sum of the image of each coherent system. However, the method still needs a large amount of CPU time to perform the simulation because the number of layers and the size of images are large. As the technology scales down and the accuracy requirement goes up, it will be more challenging to meet the tight requirement of design turn-round time.

Leading commercial computational lithography products have already started to use special coprocessor acceleration to further accelerate computation. Brion Technologies (now part of ASML) reports that each leaf node composed of two CPUs and four FPGAs in their Tachyon System can achieve 20X speedup over one single CPU node [Cao et al. 2004]. Mentor Graphics uses Cell Broadband Engine to accelerate the computation in their nmOPC product [Mentor 2004]. Clear Shape Technologies has filed patents for using GPUs to accelerate the computation [Wang et al. 2006].

This article presents a new hardware implementation for accelerating lithography imaging simulation on FPGA platforms. Unlike the image-based approach Brion takes, which ultimately relies on the accelerated performance of 2D-FFT, we use the polygon-based approach [Cobb and Zakhor 1995; Wong 2005] instead. The polygon based approach makes use of the fact that the actual layouts are solely composed of rectilinear shapes, and it has comparable or even better performance than an image-based approach in software implementation. Recent advances in OPC algorithms, for example, IB-OPC [Yu and Pan 2007] also employ a polygon-based approach for lithography intensity simulation. Moreover, the polygon-based approach precomputes the convolution and stores that into a look-up table, and the subsequent computation mainly just

involves some additions and subtractions on the look-up value. Thus the polygon based approach could be better approximated via fixed-point computations without sacrificing much accuracy. The algorithm can be better parallelized and accelerated by utilizing the high bandwidth of on-chip memory in FPGA.

Another unique aspect of our work is that we leverage state-of-the-art C to HDL compilation tools and write all our design in C, whereas the FPGA accelerator design by Brion [Cao et al. 2004] was based on completely manual RTL coding. A design described in higher level languages such as C/C++ is more portable to various platforms and easier to maintain. Also these tools could evaluate multiple design choices faster and perform various kinds of optimizations for improved performance against the RTL-based design, but those tools also have limitations on the supported language features. The challenge we experienced for this design is that of manually developing an efficient memory partitioning scheme, based on the observation of the memory access pattern, to provide a large data bandwidth for a larger number of processing elements. Deep loop pipelining and the overlapping of the communication and computation via ping-pong buffers are also implemented to take advantage of both instruction-level parallelism and task-level parallelism. All the design techniques are represented at algorithmic level in the code refinement/rewriting of ANSI C, and the resulting C code is further synthesized into RTL though the automatic C-to-RTL synthesis tools.

This article is organized as follows: Section 2 describes the basic equations we use in this work for lithography simulation and discusses the trade-offs between the image-based approach and the polygon-based approach. Section 3 describes our entire design, specifically the design of memory partitioning. Section 4 discusses our experience on using C to HDL compilation tools, and the C code refinements in implementing the design. Section 5 describes our experimental results, and Section 6 concludes the article. A preliminary version of this work was presented in Cong and Zou [2008]. This article includes additional discussions and results, such as a detailed derivation on address generation and data multiplexing, discussions of communication overhead, the nested loop pipelining, interconnect issues and comparisons against the GPU implementation.

2. BASICS FOR AERIAL IMAGE SIMULATION

2.1 The Imaging Equation

The coherent decomposition method first decomposes the whole optical system, into a series of coherent optical systems (using eigenvalue decomposition). The series is truncated to a finite one based on the ranking of the eigenvalues. If we keep only K significant eigenvalues and eigenvectors, the image can be computed as:

$$I(x, y) \approxeq \sum_{k=1}^{K} \lambda_k |(O \otimes \phi_k)(x, y)|^2. \quad (1)$$

Here the $I(x, y)$ is the image intensity, λ_k is the kth eigenvalue, $O(x, y)$ is the object function (field), and $\phi_k(x, y)$ is the kth eigenvector. The symbol $\otimes$ denotes convolution (2D image convolution). For more details on the derivation of the coherent decomposition method, please refer to Cobb [1998].

One way to perform the 2D image convolution is through 2D FFT. We can first transform the padded object pattern (see Section 2.3) and the eigenvector into the frequency domain via 2D FFT. (Note the FFT of the eigenvector ϕ_k only needs to be computed once and can be reused.) Then we multiply the eigenvector and object pattern in the frequency domain. Finally we can obtain the convolution result via an inverse FFT of the multiplied result. This is the image-based simulation that is also used by Cao et al. [2004].

As the actual layout of VLSI circuits is only composed of polygons (or rectangles if we perform polygon decomposition on the layout), the convolution of different sizes of polygons/rectangles can be precomputed and stored. We first consider purely rectangle cases. The convolution for an object pattern solely composed of N rectangles with vertices at $(x_1^{(n)}, y_1^{(n)})$, $(x_2^{(n)}, y_1^{(n)})$, $(x_1^{(n)}, y_2^{(n)})$,$(x_2^{(n)}, y_2^{(n)})$ can be simplified via *quadrant functions*.

The object pattern in one padded area can be written as:

$$O(x, y) = \sum_{n=1}^{N}[Q(x - x_1^{(n)}, y - y_1^{(n)}) - Q(x - x_2^{(n)}, y - y_1^{(n)}) + Q(x - x_2^{(n)}, y - y_2^{(n)}) - Q(x - x_1^{(n)}, y - y_2^{(n)})], \quad (2)$$

where *quadrant function*

$$Q(x, y) = \begin{cases} 1 & if\ x \geq 0\ and\ y \geq 0 \\ 0 & otherwise. \end{cases}$$

The convolution equation thus can be rewritten as:

$$\begin{aligned} I(x, y) &\approxeq \sum_{k=1}^{K} \lambda_k |(O \otimes \phi_k)(x, y)|^2 \\ &= \sum_{k=1}^{K} \lambda_k | \sum_{n=1}^{N} [\psi_k(x - x_1^{(n)}, y - y_1^{(n)}) - \psi_k(x - x_2^{(n)}, y - y_1^{(n)}) \\ &+ \psi_k(x - x_2^{(n)}, y - y_2^{(n)}) - \psi_k(x - x_1^{(n)}, y - y_2^{(n)})]|^2, \end{aligned} \quad (3)$$

where

$$\psi_k(x, y) = Q(x, y) \otimes \phi_k(x, y)$$

is the convolution of the quadrant function with the kth eigenvector. This is the polygon/rectangle-based algorithm we use, and it is described in more detail in Wong [2005].

For rectilinear polygons, an equation similar to Equation (2) can be written using quadrant function on each vertex of the polygons. In Figure 1, we label the + and − on the vertexes of rectilinear polygons (the rectangle is simply a special type of rectilinear polygon, and + and − are just the signs for the look-up value for the vertexes; see Equation (2) and Equation (3) for the rectangle

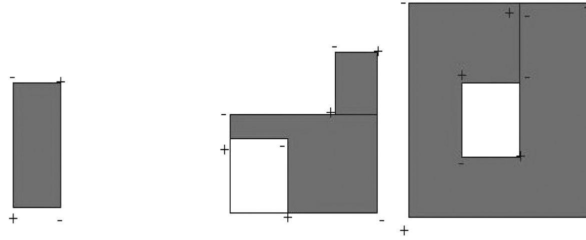


Fig. 1. Rectilinear polygons can be processed similar to rectangles.

case). We go from a vertex which is at bottom-left, and label that with + and go around the border of the polygon and label that with − and + respectively.

Using rectangles or polygons will not alter the overall algorithm and design presented in the following. For simplicity and benchmarking purposes, we assume we are given N rectangles with 4N vertexes for one region of the image in the subsequent illustration, while our litho simulation tool, which goes from GDSII to simulated image, is also capable of processing polygons from the GDSII directly without the need for rectangle decomposition.

2.2 Image-Based Simulation versus Polygon-Based Simulation

It is worthwhile to briefly discuss the trade-offs between image-based simulation and polygon-based simulation. Image-based simulation first converts the layout, which in most cases is stored in GDSII, into an object image, and then gets the image convolution via FFT and IFFT. Note that only image-based simulation needs to take this additional conversion step from the GDSII to the object image, which might also be expensive. The 2D-FFT algorithm, although well studied, still needs a large number of floating-point operations. On the other hand, the polygon-based algorithm can use fixed-point computation without losing much accuracy, and thus can be implemented without using any floating-point operations. Suppose the rounding error for the elements in ψ_k in Equation (3) is 2^{-p}, the absolute error for computing $\sum_{n=1}^{N}[\psi_k(x - x_1^{(n)}, y - y_1^{(n)}) - \psi_k(x - x_2^{(n)}, y - y_1^{(n)}) + \psi_k(x - x_2^{(n)}, y - y_2^{(n)}) - \psi_k(x - x_1^{(n)}, y - y_2^{(n)})]$ is bounded by $4N * 2^{-p}$. If we assume the error distribution for the elements in ψ_k is a uniform distribution between -2^{-p} and 2^{-p}, the error distribution of the sum $\sum_{n=1}^{N}[\psi_k(x - x_1^{(n)}, y - y_1^{(n)}) - \psi_k(x - x_2^{(n)}, y - y_1^{(n)}) + \psi_k(x - x_2^{(n)}, y - y_2^{(n)}) - \psi_k(x - x_1^{(n)}, y - y_2^{(n)})]$ follows uniform sum distribution. Its variance is proportional to N and the standard deviation is proportional to $\sqrt{N}$, thus the actual error is much smaller, statistically, than the conservative error bound which is linear with N.

Both algorithms scale linearly with the number of pixels to compute. The issue of the polygon-based approach is that the running time will also depend on the layout density, which determines the number of polygons or rectangles in a unit area within the interaction range (N in Equation (3)), while the image-based approach only depends on the chip area. We implemented the 2D-FFT based 2D-convolution using the FFTW package [Frigo and Johnson 2005], and tested that on kernels with size 400 by 400. We found that the running time

is comparable to a polygon-based method with a moderate density (see Section 5.3). The polygon based approach requires less computation and runs faster for layers that are not very dense, and the image-based approach runs faster for very dense layers. Note that the polygon-based approach also saves the step on conversion from polygons to images, as polygons are naturally stored in GDSII.

In terms of the FPGA-based acceleration, FFT is still tightly constrained by the available DSP units or logic slices, and the peak FLOPs of FPGA are at the same magnitude with the peak FLOPs of modern CPU; thus, typically only a 2 to 8X speedup is seen on accelerating FFT on FPGA platforms via parallel implementation [Uzun et al. 2005]. Fixed-point FFT core for FPGA is also available and give potentially larger speedup, but it will have a worse accuracy because multiplication can enlarge the absolute error. For the polygon-based approach, the convolution on the quadrant function can be precomputed using highly accurate floating point computations (on CPU) and reused multiple times. The time to precompute the convolution can be ignored as it is a one-step process. The remaining computations only involve table look-up and simple addition/subtraction operations, and are much more suitable for a decent speedup. Therefore, in this work we use the polygon/rectangle-based approach rather than the image-based approach for accelerator design.

2.3 Detailed Settings for the Imaging Equation Using the Polygon-Based Approach

We assume the convolutions of eigenvectors and the quadrant functions are already precomputed, and sampled into a 2D array called *kernel*. The region/range of the kernel we use is 2000nm by 2000nm; it is sampled on a 5nm grid, and thus contains 400 by 400 numbers. The image we need to compute is on a 25nm grid. Without loss of generality, we assume the layout corners (vertexes of the polygons) are also on the 5nm grid. (If the layout corner is on a much finer grid, interpolation will be used to get the kernel value.) These settings used in our algorithm and implementation were recommended by our industry collaborator from Magma Design Automation [Wong 2007], but our architecture certainly is not confined to these settings and can be extended to other settings. Some setting changes require a recompile/resynthesis while some do not. This depends on whether the change of underlying hardware is needed. For example, the design synthesized with a large N (layout density metric) can be used for a smaller N without the need of changing hardware. On the other hand, enlarging array sizes or changing the memory partitioning schemes will affect the underlying hardware and a recompile/resynthesis will be needed to generate the new hardware bitstreams.

3. FPGA-BASED ACCELERATOR FOR THE IMAGING SIMULATION

3.1 Image Padding for the Polygon-Based Approach

Both the object pattern and the simulated image are large; however, when we compute one region of the image, only one padded region of the object needs to

```
for (x=0;x<pixel_max;x++)
 for (y=0;y<pixel_max;y++)
 {
//Initialize pixel intensity
  I[x][y]=0;
  for(k=0;k<K;k++)
  {
   //Initialize partial sum
   I_k[x][y]=0;
   //Core computation
   for(n=0;n<4*N;n++)
    {
     addr_x=5*x-rect_x[n]+c;
     addr_y=5*y-rect_y[n]+c;
     I_k[x][y]+=(-1)^n*kernel[k][addr_x][addr_y];
    }
    I[x][y]+=I_k[x][y]*I_k[x][y];
   }
  }
```

Fig. 2. Pseudocode for the nested loop.

be considered due to the locality of the litho effects. For example, for a kernel ranges within a 2000nm by 2000nm area, if we want to compute a 1000nm by 1000nm image region, an object pattern within a range of 3000nm by 3000nm needs to be taken into computation. The reason for this is that some objects are far away from the current pixel and out of the interaction range, and therefore need not be considered.

The computation complexity is proportional to the number of rectangles N taken into computation, and the intensity of each pixel is determined by the rectangles within the interaction range (2000nm by 2000nm in our case) around this pixel.

3.2 Rearranging the Nested Loop

Now we devote ourselves to implementing the nested loop corresponding to Equation (3), which is described in Figure 2, where c is a constant for addressing alignment, and $rect_x$ and $rect_y$ are arrays for the coordinates of rectangle corners. Note that this is the code for simulating one region of an image, and there is another outer loop over the pseudo-code in Figure 2 for changing the current image region where the input N and $rect_x$ and $rect_y$ shall all be changed as we move to different image regions. Here the *pixel_max* is the number of pixels in either X or Y direction. We use $5 * x$ and $5 * y$ in the code as we use an image grid on 25nm and a kernel grid on 5nm. The outermost loop goes over different image pixels. Within the outer loop there is a loop that goes over different kernels. The innermost loop goes over different layout corners. This is a direct implementation of Equation (3), but it might not be suitable for generating synthesizable hardware. We apply loop interchange techniques to find a better rearrangement for the nested loop.

```
//Initialize pixel intensity
for(x=0;x<pixel_max;x++)
 for(y=0;y<pixel_max;y++)
  I[x][y]=0;
for(k=0;k<K;k++)
 { //Initialize partial sum
    for(x=0;x<pixel_max;x++)
     for(y=0;y<pixel_max;y++)
      I_k[x][y]=0;
     //The core computation
    for(n=0;n<4N;n++)
     for(x=0;x<pixel_max;x++)
      for(y=0;y<pixel_max;y++)
      {
       addr_x=5*x-rect_x[n]+c;
       addr_y=5*y-rect_y[n]+c;
       I_k[x][y]+=(-1)^n*kernel[k][addr_x][addr_y];
      }
     //Square operation
    for(x=0;x<pixel_max;x++)
     for(y=0;y<pixel_max;y++)
      I[x][y]+=I_k[x][y]*I_k[x][y];
 }
```

Fig. 3. Pseudocode for the rearranged nested loop.

First, we look at the choices for the outermost loop. Because the whole nested loop will need the data in the kernel array, which will be reused for the image computation with different image regions or different pixels and layout corners, we would like to prefetch the kernel array and store the kernel array in the on-chip RAM of the FPGA. In our setting in Section 2.3, each kernel has $16 * 400 * 400$ bits of data, which is 2.44Mb, if we use 16bit precision for kernel data. As the total size of on-chip RAM of FPGA is limited, it is unlikely that all the kernels will fit, but in our case at least one kernel can be put in. (the device we use has around 9Mb on-chip memory in total) Thus, we would like to make the loop over different kernels the outermost loop.

For the inner part, our considerations are that the loop over different layout corners is less *structured* than the loop over the image pixels. If we fix one layout corner and update the set of pixels, the memory access pattern is very regular, but if we fix one pixel and change different layout corners, it will not be that regular because we can not expect the layout corners to have some specific pattern. Thus, we would like to make the loop over different pixels the inner-most loop. Figure 3 is the nested loop after the loop interchange.

The illustration of the computation is shown in Figure 4. The address of kernel data depends on the value of the rectangle corner. For one specific corner, the address for the kernel array is just an affine mapping over the pixel index x and y. As the data of $rect_x[n], rect_y[n]$ is in the layout corner array and accessed at runtime, the data access for the kernel array is still some type

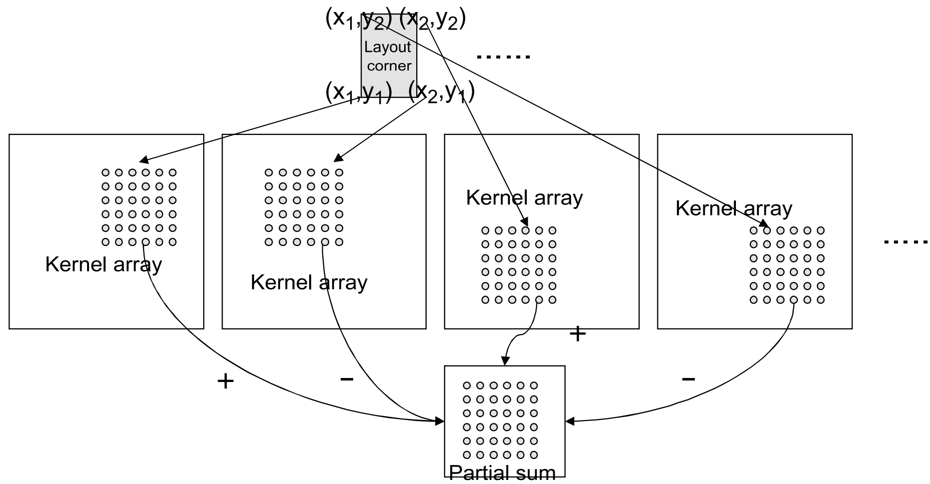


Fig. 4. Computation of the inner nested loop.

of indirect memory access. This imposes great challenges for the automation tools. After the rearranging of the nested loop, the key problem is to design and implement the inner loop—the inner loop over different rectangle corners and image pixels.

3.3 Communication Analysis and HW/SW Partitioning

In our design, the hardware component running on FPGA mainly initializes and computes the image partial sum I_k, while the result is sent back to a software component running on the processor; the software component parses and provides input data to the hardware component and also performs the square operation and stores the results.

As the computation is mainly performed on the FPGA coprocessor, input data required for computation needs to be transferred from the host CPU to the coprocessor, and the computed results need to be transferred back. Although the Hyper-Transport bus enables a low latency solution, overhead in the data transfer still exists.

The major part of data transfer from host (CPU) to coprocessor is the layout corner array ($rect_x$,$rect_y$ in the pseudo-code). Note that kernel data also needs to be transferred from the host to coprocessor, but the same kernel can be used for a larger number of padded image region, thus the communication overhead in transferring kernel data can be neglected. The major data transfer from coprocessor to CPU is the array of image partial sum I_k. Assume we use 16-bit data for the elements in $rect_x$,$rect_y$ array and 32-bit data in array I_k. For settings shown in Section 2.3, the total bytes of data transfer for computing one padded region with size 1000nm by 1000nm on a 25nm grid is $(32/8) * 4N + (32/8) * 40 * 40$, where the first term corresponds to the data transfer for the layout corner array and the second term corresponds to the transfer of partial image. The data transfer is done in a DMA-like fashion.

For a moderate density say N = 100, and a data bandwidth around 800MB/s (the peak bandwidth of the SRAM device we use as hardware/software shared memory), the data transfer needs around 10*us*. This is roughly 10% of the overall execution time of our accelerated design. In Section 3.8 we talk about the overlapping of the communication and computation, a technique that could completely resolve the overhead.

3.4 Exploring Parallelism

Typically, the accelerator on the FPGA platform is able to explore task parallelism, data parallelism, and instruction parallelism. We will use high level synthesis tools, for example, AutoPilot™[1] to implement our design. AutoPilot first parses the input description e.g., in C, and generates control data flow graph (CDFG) of the code. Then it performs scheduling and binding on the CDFG to generate the final RTL. Instruction parallelism, is directly realized by the scheduler of the tool, because scheduler might schedule multiple instances of function units (FU) at same cycle. Moreover, the loop pipelining pragma can inform the scheduler to schedule the loop in a pipelined fashion, creating more instruction parallelism. Task parallelism needs to explicitly write multiple processes or tasks, and needs to consider inter-process synchronization and arbitration of shared resources. Currently, AutoPilot can realize the task parallelism via functional block pipelining or through SystemC based description. Data parallelism, on the other hand, widely used in SIMD instructions or GPU accelerators, tries to use a same or similar program/code to cope with multiple data. AutoPilot uses loop unrolling pragma to invoke program transform for loops and the scheduler will schedule the unrolled CDFG to create the data parallelism.

In our initial scheme, we developed a design based on task-level parallelism. We first partitioned the kernel array and the partial image array into several partitions based on the geometric locations. Figure 5 shows a 4-way naive partitioning.

We then allocate four PEs in the FPGA, and each PE is responsible for the computing of one partition of partial image array. As the computing of one partition of partial image might need all the four partitions of kernel data, access conflicts might occur. We schedule the operations such that different PE will read or write different memory blocks at each computation stage (shown in Figure 6).

However, as the address of the required kernel data depends on the set of layout corners, it is likely that this approach might face load balancing problems, if the layout corner is not uniformly distributed. For example, if the whole loop makes heavy use of one or several specific partitions, the benefit or speedup using partitioning might become degraded. Another drawback is that this type of task parallelism need control flow in each PE and need additional logic to do synchronization.

[1]http://www.autoesl.com

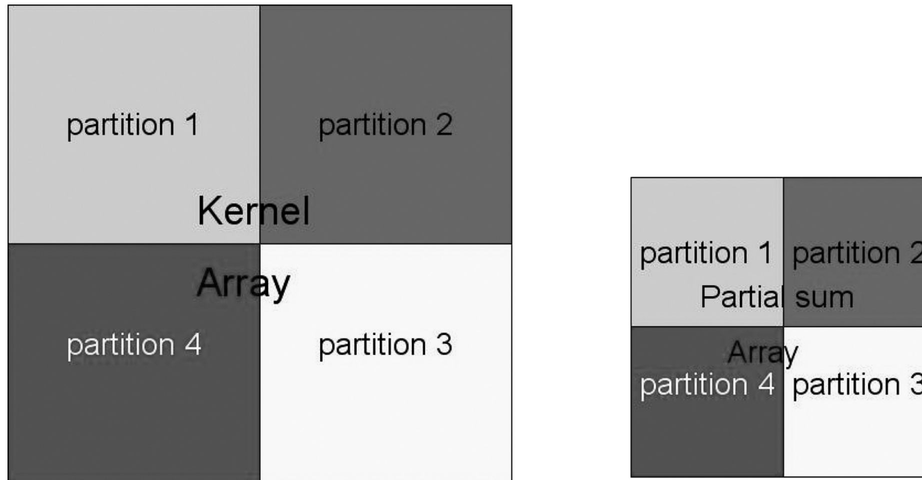


Fig. 5. Naive partitioning based on geometric locations.

Later, we decided to mainly borrow the idea from data parallelism, where we first unroll the inner nested loop (the loop over different pixels) to some degree and try to execute the multiple operations in the inner loop at exactly the same cycle. The benefit is that the control flow is more simplified and the load balancing problem no longer occurs. The 4-way unrolled code is shown in Figure 7, where we unroll once in x direction and once in y direction. (Note this unrolling doesn't need to be written explicitly, as in Figure 7, but can also be achieved by specifying unrolling pragmas in the original loop.) However, this rewriting technique does not help without further memory partitioning, as each on-chip memory block only has limited ports. When loop pipelining is further enabled, the unrolling might increase the initiation interval of pipelining and not contribute much for the overall latency. The goal of the memory partitioning is to make sure the correspondent simultaneous memory accesses in the unrolled loop are partitioned into different memory blocks.

Besides the parallelism similar to data parallelism, we also use a loop pipelining technique as an instructional parallelism technique, and the data prefetching or the overlapping of the SW/HW communication and computation using function block pipelining or task-level parallelism.

3.5 Memory Partitioning Using Modulo Addressing

This subsection discusses the memory partitioning scheme to allow multiple memory access in the inner unrolled loop to be parallelized. For each block of on-chip memory in the FPGA, typically there are only two ports available. If the kernel array and partial image (a temporary buffer for each pixel to store the inner sum) are just single memory blocks without partitioning, multiple memory accesses can not be scheduled to the same cycle due to port contention. Thus, clearly we need to partition the memory to allow for parallel processing and to achieve high bandwidth and high throughput.

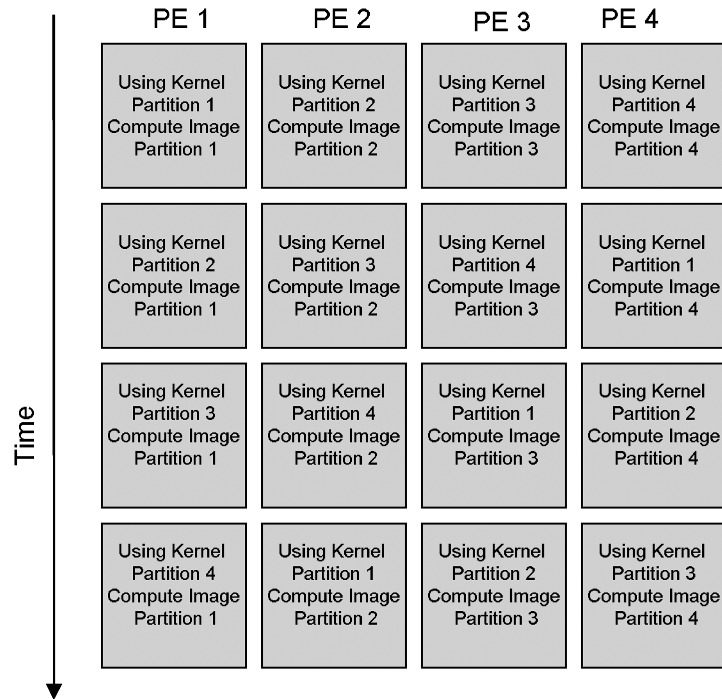


Fig. 6. Block scheduling for naive partitioning.

The idea we used in memory partitioning is a variant of modulo addressing. The modulo addressing approach with a circular buffer that could get a row of data containing n elements from n banks is presented in Tanskanen et al. [2004]. The basic idea of Modulo addressing is simple: if we want to fetch a row of data with address $x, x+1, x+2, \cdots, x+n-1$ simultaneously, we can use cyclic array partitioning where data with different modulo of address $x\%n$ are placed in different memory blocks. Cubic addressing [Doggett and Meissner 1999] provides a partitioning scheme for 3-D array which can obtain a 2*2*2 voxel neighborhood simultaneously from 8 banks, this is a special case for modulo addressing. Our case is a 2D case rather than 1D. Also, we want data with address $5x, 5(x+1), \cdots, 5(x+n-1)$ are in different blocks; thus we need some natural extensions on the simple modulo addressing. Also the aforementioned work is in the domain of multiport memory architecture design for medical image processing or video coding, while we describe the partitioning scheme for our design in behavior C for reconfigurable computing.

In our case, we mainly have three arrays: the kernel array, which serves as the look-up table for the computation; the partial image sum array, which is used to store the intermediate inner sum for different pixels; and the layout corners array. Without loss of generality, we assume that the overall size of the three arrays can be loaded into the on-chip RAM of the FPGA. This assumption holds for our test settings, but generally might not be true, and further partitioning of data and computations might be needed.

```
......
  //Core computation, 4-way unrolling
  for(n=0;n<4N;n++)
   for(x=0;x<pixel_max/2;x++)
    for(y=0;y<pixel_max/2;y++)
    {
     addr_x=5*2*x-rect_x[n]+c;
     addr_y=5*2*y-rect_y[n]+c;
     I_k[2*x][2*y]
       +=(-1)^n*kernel[k][addr_x][addr_y];
     I_k[2*x+1][2*y]
       +=(-1)^n*kernel[k][addr_x+5][addr_y]
     I_k[2*x][2*y+1]
       +=(-1)^n*kernel[k][addr_x][addr_y+5]
     I_k[2*x+1][2*y+1]
       +=(-1)^n*kernel[k][addr_x+5][addr_y+5];
    }
......
```

Fig. 7. Pseudocode for the partially unrolled nested loop.

We would like to further partition the array to take advantage of the high peak bandwidth of the on-chip RAM. Multiple Processing Elements(PE), can process multiple data concurrently if we partition the kernel array and the partial sum array effectively without access contention and serialization.

To obtain a better partitioning scheme for this specific nested loop, we need to take a look at the memory access pattern. For a specific layout corner, we need to update all the image pixels. The corresponding data access in the kernel array has a regular pattern (shown in Figure 4). A better partition scheme should be able to evenly distribute the memory access pattern shown in Figure 4 into multiple memory banks/blocks, regardless of the location value of specific layout corners. Therefore, we choose to use a modulo interleaving partition scheme.

We still take 4-way partitioning as an example, and the illustration is shown in Figure 8. *The same color/texture means the data are physically stored in the same memory bank/block.* Using this partition scheme, multiple data can be fetched concurrently without any conflicts.

The basic idea for partitioning is to follow the grid size of the access pattern in both X and Y directions so that memory accesses in the most inner unrolled loop in Figure 7 are always in different memory partitions. In our case, the kernel array is on a 5nm grid and the image array on a 25nm grid. The illustration of the partitioning is shown in Figure 8. In this figure, dots are memory accesses for kernel arrays (shown on the left) and the memory access for the image partial sum (shown on the right). We can see the concurrent accesses always lie *in different colors* in Figure 8, and thus can be scheduled to execute at the same cycle. Figure 8 could be obtained via first *coloring* (here we use *coloring* to represent the memory partitioning) the bottom-left 5 * 5

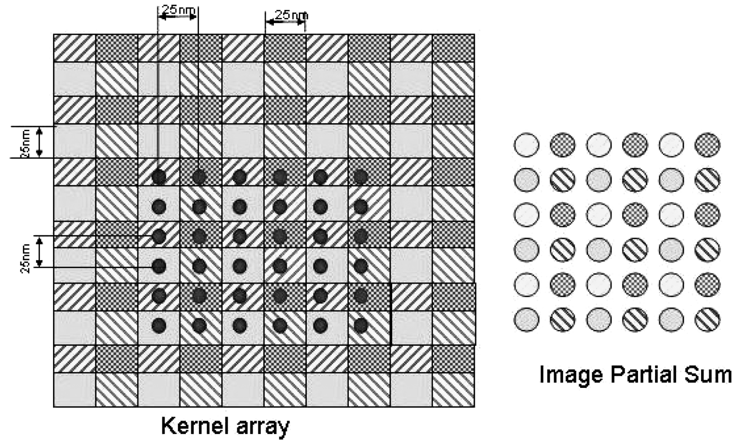


Fig. 8. 4-way (2 by 2) memory partition scheme for load balancing.

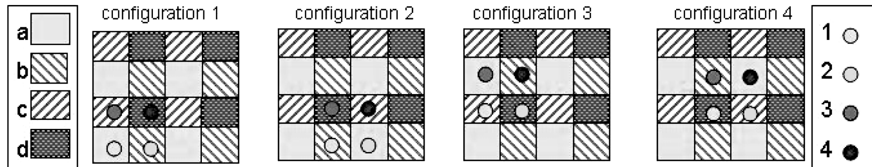


Fig. 9. Address generation and output data multiplexing.

corner blocks in the kernel array, and *coloring* other blocks in an interleaved fashion, to ensure that the four memory accesses in the inner unrolled loop are in different memory blocks. Note that Figure 8 only shows a 4-way 2 * 2 partitioning corresponding to the partially unrolled loop in Figure 7, but a 5 * 5 or 8 * 8 partitioning scheme can also be developed similarly to allow for a larger data bandwidth.

The array of image partial sum is also partitioned in a fashion shown in Figure 8, so that we can write the output data into the array concurrently. As the addresses for image partial sum are affine mapping of loop variables and do not depend on runtime data, the partitioning for the image partial sum array is somewhat simpler. The layout corner array does not need partitioning as the loop over layout corners is an outer loop.

3.6 Address Generation Logic

As we explore the memory access pattern to partition the memory to allow for concurrent access, the addresses to fetch those data also need to transformed and mapped.

The address generation logic, and the memory partitioning and the multiplexing of output data, are all implemented via rewriting the code of the original algorithm written in ANSI C.

Again take the 4-way partitioning in Figure 7 and Figure 8 as an example. In Figure 9, a, b, c, d are four memory blocks after partitioning, and 1, 2, 3, 4

are four concurrent memory accesses. There are four different configurations shown in the figure. With different address shifting determined by $rect_x$ and $rect_y$, the concurrent memory accesses have different combinations with the memory blocks they visit. In configuration 1, the four concurrent memory accesses $1, 2, 3, 4$ will need the data in memory block a, b, c, d, respectively. In configuration 2, the four accesses will need the data in memory block b, a, d, c, respectively. Different configurations will have slightly different address generation logics.

The address generation logic first looks at which configuration is among the four cases in the 2 by 2 partitioning design in Figure 9. Later, for each configuration, there is a mapping function to transform the original address into the mapped address. For each of the four configurations shown in Figure 9, the data we get from different memory partitions also needs to go through multiplexing to provide the required data for the accumulator. Let us go into more detail on the address generation. Again we take 2 by 2 partitioning cases as an example. Suppose we denote that the four addresses that the unrolled loop needs to access (before the address mapping) in Figure 7 are $[addr_x][addr_y]$, $[addr_x{+}5][addr_y]$, $[addr_x][addr_y{+}5]$ and $[addr_x{+}5][addr_y{+}5]$ respectively. We first determine which group the address lies in by looking at the quotient $addr_x/(2*5)$, $addr_y/(2*5)$; e.g., in Figure 9 $addr_x/(2*5) = 0$ and $addr_y/(2*5) = 0$. We can determine which configuration it is by looking at the modulo $addr_x\%(2*5)$ and $addr_y\%(2*5)$; for example, $addr_x\%(2*5) < 5$ *and* $addr_y\%(2*5) < 5$ means that it is the first configuration. The divisor here is $2*5$: 2 relates to 2 by 2 partitioning, 5 relates to the image grid size which is 5X larger than kernel grid size (also in the pseudo-code in Figure 2).

We denote the mapped address $[addr_x^a][addr_y^a]$, $[addr_x^b][addr_y^b]$, $[addr_x^c][addr_y^c]$, $[addr_x^d][addr_y^d]$ for the four different memory blocks shown in Figure 9. And a mapping function is

$$\begin{aligned} addrmapped_x &= f(addr_x) = (addr_x/(2*5))*5 + addr_x\%5 \\ addrmapped_y &= f(addr_y) = (addr_y/(2*5))*5 + addr_y\%5. \end{aligned}$$

The first term determines the address that corresponds to different groups of concurrent access, and the second term determines the address shifting within a 5*5 block.

If it is the first configuration, then the addresses for the four memory block are the same.

$$\begin{aligned} addr_x^a &= addr_x^b = addr_x^c = addr_x^d = addrmapped_x \\ addr_y^a &= addr_y^b = addr_y^c = addr_y^d = addrmapped_y. \end{aligned}$$

If it is the second configuration, we have

$$\begin{aligned} addr_x^b &= addr_x^d = addrmapped_x \\ addr_x^a &= addr_x^c = addrmapped_x + 5 \\ addr_y^a &= addr_y^b = addr_y^c = addr_y^d = addrmapped_y. \end{aligned}$$

We can get the address for the remaining two configurations similarly. After we get the addresses for each configuration, we can somewhat simplify the

logic by extracting the common terms and write the address generation logic as follows:

$$\begin{aligned} addr_x^a &= addrmapped_x \;\; if(addr_x\%(2*5) < 5) \\ & \quad addrmapped_x + 5 \;\; otherwise \\ addr_y^a &= addrmapped_y \;\; if(addr_y\%(2*5) < 5) \\ & \quad addrmapped_y + 5 \;\; otherwise. \end{aligned}$$

When we use other partitioning, such as 5 * 5 partitioning, similar address generation logic can also be written using the same idea and format.

Note that the equations shown above use division and modulo operations, it is well known that these operations are relatively costly on FPGA in terms of both area and latency. However, we precompute these costly operations at the CPU side and store them so that the FPGA does not need to worry about this. If we recall the address generation shown in Figure 2, we only need to convert the array data in $rect_x$ and $rect_y$ to the form of $quotient * divisor + remainder$ so that the quotient and remainder used in address computation can be obtained directly.

3.7 Output Data Multiplexing

The data we fetched from the partitioned memory blocks needs to go through multiplexing before it is sent to the accumulator, because the computation of one partition of image partial sum might still require data in different partitions of kernel array. For the 2 by 2 partitioning shown in Figure 9, if the data from the four memory blocks are $array_a[..]$, $array_b[..]$, $array_c[..]$, $array_d[..]$ and the data we can directly send to the accumulator are $Reg_1, Reg_2, Reg_3, Reg_4$. We denote the multiplexing logic as

$$\begin{aligned} & Reg_1, Reg_2, Reg_3, Reg_4 \\ &= (array_a[..], array_b[..], array_c[..], array_d[..]) \;\; (configuration\ 1) \\ &= (array_b[..], array_a[..], array_d[..], array_c[..]) \;\; (configuration\ 2) \\ &= (array_c[..], array_d[..], array_a[..], array_b[..]) \;\; (configuration\ 3) \\ &= (array_d[..], array_c[..], array_b[..], array_a[..]) \;\; (configuration\ 4). \end{aligned}$$

But this naive multiplexing might have a large routing overhead when we have a larger partitioning. We use 2D ring-based shifting to implement the multiplexing. Figure 10 is the interconnect structure for the 2 by 2 partitioning using ring-based multiplexing. For configuration 1, no shifting is needed; for configuration 2, we can shift one step in X direction; for configuration 3, we can shift one step in Y direction; and configuration 4 requires shifting one step in both X and Y directions.

A larger partitioning scheme, for example, 5 * 5 partitioning can also use a similar multiplexing scheme. Figure 11 is the interconnect structure for the 5 * 5 partitioning using ring-based multiplexing. The pseudo-code for the 2D-ring-based multiplexing for 5 * 5 partitioning is shown in Figure 12. sel_x and sel_y are values to determine how many steps the whole ring needs to be shifted, which can be obtained by modulo of the layout corner, and *ring_shift_x*

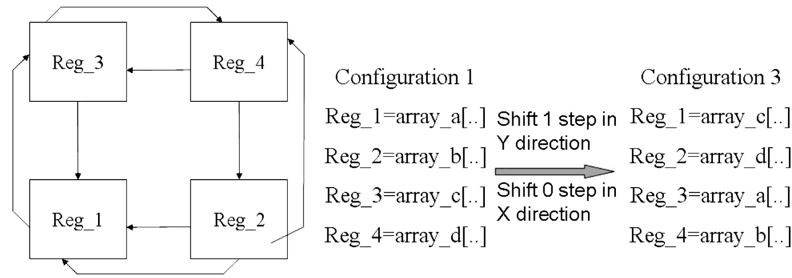


Fig. 10. 2D ring for 2 by 2 partitioning.

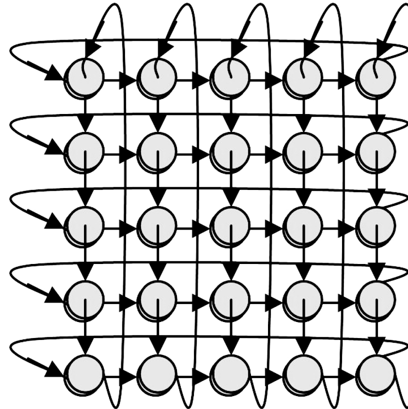


Fig. 11. Ring-based data multiplexing for 5 by 5 partitioning.

means all data in the 2D-ring is assigned the value on its circular left side, and *ring_shift_y* means all data in the 2D-ring is assigned the value on its circular upper side. Although the *if* statement inside the loop can be merged to the loop bound, we write it in the current form so that it has a constant loop bound. These loops are further unrolled/flattened to facilitate the loop pipelining of the outer loop. We wrote the pseudo-code in such a way that the cycles need to perform shifting are constant regardless of which configuration it is, otherwise a nondeterministic cycle count might bring difficulties for pipelining. The whole shifting is done in a multicycle fashion. In our 5 * 5 partitioning-based design, we will shift two steps in one clock cycle. Although the latency of the multiplexing through this 2D ring structure is long, it will not affect the overall performance due to loop pipelining, as the multiplexing block can be implemented as a unit that has a multicycle latency but with a one cycle pipeline initiation interval (similar to many floating point IPs).

3.8 Loop Pipelining and Function Block Pipelining

The whole nested loop is pipelined to increase the throughput. Although many rewritings, including the address generation and data multiplexing, complicate the logic and increase the latency, they will not affect the performance

```
//shifting/multiplexing in X direction
 for(m=0; m<5-1; m++)
 {       if(sel_x>m){
                ring_shift_x;
                }
 }
 //shifting/multiplexing in Y direction
 for(m=0; m<5-1; m++)
 {      if(sel_y>m){
                ring_shift_y;
                }
 }
```

Fig. 12. Pseudocode for 2D ring multiplexing for 5 * 5 partitioning.

```
 for(x=0;x<pixel_max;x++)
      for(y=0;y<pixel_max;y++)
          {// loop pipelining pragma
                ......
          }
```

(a) wo flattening

```
x=0;y=0;
 for(idx=0;idx<pixel_max*pixel_max;idx++)
          {// loop pipelining pragma
                ......
               y++; if(y==pixel_max) {y=0;x++;}
          }
```

(b) w flattening

Fig. 13. Loop flattening for deep loop pipelining.

much because the whole nested loop (over different rectangles and image pixels) can be pipelined and can achieve an initiation interval equal to one. Loop pipelining can be achieved by specifying loop pipelining pragmas within the nested loop. The core loop we implement is a nested loop. Specifying loop pipelining pragmas might only pipeline the inner loop. The pipelining possibility also lies between different instances of the middle loop and even the outer-most loop. We manually flatten the loop to reduce the depth of the loop-nest. In this way, we can further reduce latency by reducing the startup latency of the pipeline. The flattening process is shown in Figure 13. This is a tool-specific rewriting. Some tools might be able to pipeline the whole nested loop without manual flattening.

Moreover, we would like to overlap all the communications and computations so that the hardware component is running the computations almost all the time. This can either be viewed as function block pipelining (Figure 14) or

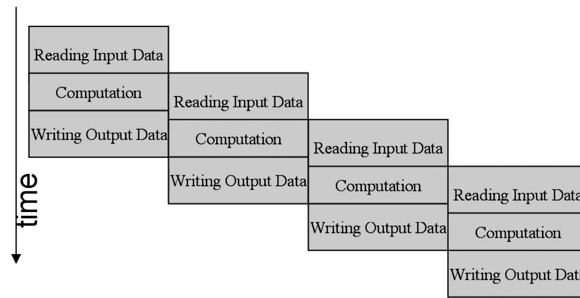


Fig. 14. Block pipelining/overlaping communication and computation.

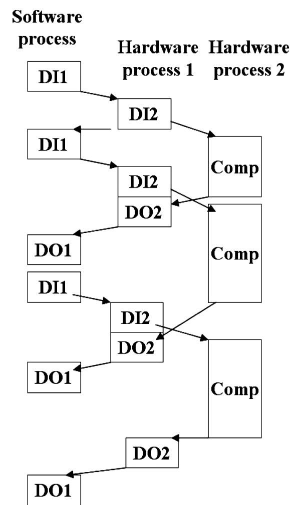


Fig. 15. Explicit control flow on overlapping communication and computation.

realized as an explicit control flow of multiple tasks (Figure 15). In Figure 15 $DI1$ means transferring data from the CPU side to the SW/HW shared SRAM, and $DI2$ means transferring data from the SRAM to the FPGA. $DO2$ means transferring data from the FPGA to the SRAM and $DO1$ means transferring data from the SRAM to the CPU. *Comp* is the computation part in FPGA. This is an explicit control flow, and two hardware processes communicate with each other via *signals*. Ping-pong buffers are used for both the layout corner array and the image partial sum array, which serve as input data array and output data array for the computation of one region respectively. 2X storage space is used for the ping-pong buffer while one is used in the current computations and the other is used for sending/receiving data. Using ping-pong buffers can ensure that the overlapping will not alter the data that is needed for current computation. In Section 3.3, we found that the communication time is not greater than the computation time, thus the overlapping can hide the overhead in the communications.

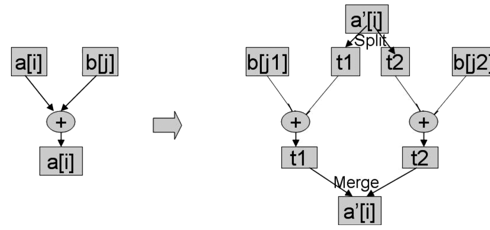


Fig. 16. Using wider data to balance the use of the memory ports.

3.9 Using Wider Memory Access to Balance the Usage of Memory Ports

Besides the techniques shown in the previous subsections, we observe that the port accesses for the *kernel* memory and the image partial sum memory are not equal. Even in the pipelined loop, at each cycle we only get one unit of data from a partition of *kernel* memory, but we use two ports for the image partial sum memory due to the accumulator. As each memory block can have two ports, we try to use both of the two ports of the *kernel* memory. But at the output side, the image partial sum memory needs to store a wider bit of data to avoid port conflict. This would provide a 2X more parallelism without further partitioning of the memory blocks. Figure 16 illustrates the computation elements using wider data.

4. LEVERAGING C TO HDL COMPILER FOR HARDWARE GENERATION

The entire algorithm is written in C so that we can leverage up-to-date C to HDL translation tools. We use AutoPilot, which is a commercial tool that can take ANSI C (within the synthesizable subset) as input and generate synthesizable and optimized RTL.

4.1 C-Based Hardware Generation and Optimization without Code Refinement

The original core C code might be as short as shown in Figure 2 or Figure 3. However, simply taking these codes into the translation tools might generate a hardware design with even poorer performance than a software implementation, as the clock frequency of FPGA is much slower than conventional CPU, and we need a much larger degree of parallelism to get speedup. Automation tools can realize the parallelism via scheduling multiple operations in the control data flow graph (CDFG) at the same cycle. Currently, these tools are not able to extract system-level parallelism automatically, but they provide a set of pragmas which work as hints or directives for invoking parallel execution. These include loop optimization techniques such as unrolling and pipelining, which can increase the performance substantially compared to a generated hardware design without using these techniques.

Loop unrolling and pipelining techniques are provided by the tool to optimize the performance of the nested loop. Loop unrolling can increase the degree of parallelism if the computation is not constrained by memory access of input data or there is some input data reuse between iterations of inner loop bodies. Loop pipelining tries to start the execution of the loop body of the next

iteration before completion of the prior iteration, and thus can greatly reduce overall latency of the nested loop. In our case, the unrolling will not help much compared to the pipelined loop as inner loop bodies will need the data from a single memory block with limited ports and there is not much data reuse for the loop. Pipelining did help as it could generate a pipelined loop with a small initiation interval (one clock cycle in our case). But as the execution of loop body without pipelining just uses around five to six clock cycles, the pipelined design still cannot completely compensate for the low clock frequency of FPGA to get a decent speedup. AutoPilot, recently added pragmas for memory partitioning, but till now they still can not handle the indirect data access in our case very well.

AutoPilot, also recently added pragmas to specify the ping-pong buffer-based IO interface. Thus, the overlapping behavior can also be generated automatically.

4.2 Code Rewriting/Refinements for the Core Nested Loop

To break up the bottleneck at code generation for the data access, we conduct a set of rewriting shown in previous subsections to increase the bandwidth and throughput for the FPGA platform. The general idea is to partition the memory blocks to allow for concurrent data access, and the access pattern needs to be exploited to develop a good partition scheme. Also the addresses for the memory access after partitioning need to be mapped or generated and data fetched from different partitions needs to be multiplexed. These are details described in Sections 3.5 to 3.7. Besides the memory partitioning we presented in the previous section, another issue is interconnect. It is very difficult for high level tools to estimate the impact of interconnect at the high level, and in most cases these tools ignore the interconnect issues, and only use the delay of the functional unit during scheduling. The code for output data multiplexing shown in Section 3.7 only performs some small bit-width comparisons and assignment operations, which does not consume much function unit delay in the modeling of the tools. Thus, tools might chain a lot of comparison and assignment into one cycle. But the interconnect delay will significantly degrade the frequency in this case. We manually add clock boundary in that code to make sure that the number of shifting steps in one cycle is fixed.

Many of these rewritings are intrinsic for hardware design. We feel that the gap between the software C code and the C code suitable for hardware generation still exists. It is unlikely that a pure software designer can master these rewriting techniques; thus, there is still much room for synthesis tools to extract the systematic parallelism and automate these refinements and rewritings, especially for users who are developing accelerators for high-performance computing who might not be very familiar with HDL and the memory hierarchy of FPGA. However, even some rewriting and code tweaking cannot be done automatically for the time being; C-based design greatly shortens the development cycle and helps maintenance of the design. Most of these refinements are specified at algorithmic level using C. We then use the C-to-HDL compilation tools to generate the RTL for the refined C code, and we also enable

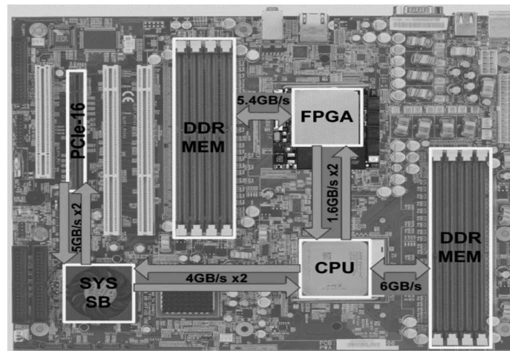


Fig. 17. XD1000 system diagram.

Table I. Device Information of EP2S180

ALUT	M512 blocks	M4K blocks	M-RAM blocks	Total Memory Bits	DSP/Multiplier	IO Pins
143,520	930	768	9	9,383,040	96 / 384	1170

the loop pipelining for the nested loop. The performance is greatly increased with the help of large parallelism in the refined code. The core C algorithm is less than one hundred lines of code, while after explicitly memory partitioning, the C code becomes around a thousand lines of code, and the generated RTL contains several tens of thousands of lines. Thus, using the C-based design shall result in a significant saving in terms of design effort, compared to a pure manual RTL design.

5. EXPERIMENTAL RESULTS

We implement the algorithm on Xtremedata's XD1000 development system.[2] Figure 17 is the system diagram of XtremeData's XD1000™ development system, which is the hardware platform we use. This development system uses a dual Opteron motherboard and one Opteron is replaced by an XD1000 coprocessor module. The XD1000 coprocessor communicates with the host Opteron CPU via Hypertransport™ links, and it is built based on Altera's largest FPGA in Stratix II family: EP2S180. Table I shows the device information of Altera Stratix II EP2S180 FPGA.

5.1 Speedup Measurement

We use a 5 by 5 partitioning scheme, and it effectively drives $25 * 2 = 50$ processing elements. *Kernel* array spans a 2000nm by 2000nm area and is a 400*400 array containing 16-bit resolution fixed point values. The window of image region we simulate has a size of 1000nm by 1000nm, thus image partial sum array is a 40*40 array containing 32-bit resolution fixed point values. Layout corner array is an array containing up to 800 32-bit values and can

[2]http://www.xtremedatainc.com

Table II. Device Utilization of the Design with 5 by 5 Partitioning

Tool	ALUT	Memory Bits	DSP/Multiplier	IO pins	Fmax(MHZ)
AutoPilot	22,457 (15.6%)	2,972,876(31.7%)	0 (0%)	485(41%)	112.30

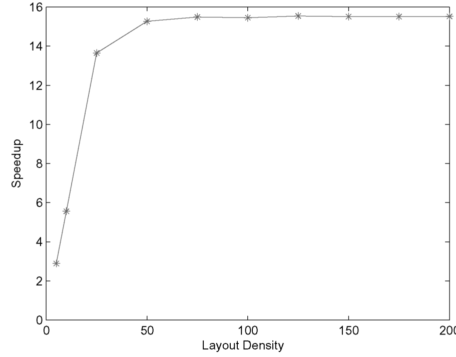


Fig. 18. Speedup plot with accelerator, single kernel.

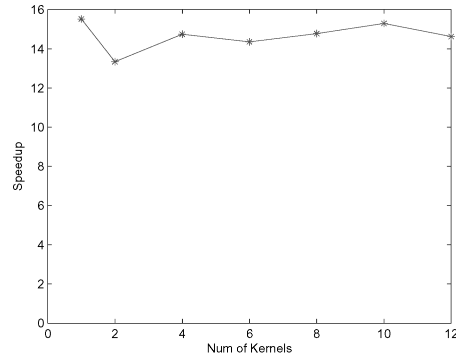


Fig. 19. Speedup plot with accelerator, multiple kernels, N=200.

store N up to 200 rectangles. All these arrays are stored in the on-chip RAM of the FPGA.

The design has only around a 20% device utilization in logic ALUT and 30% utilization of memory bits; it does not use any multiplier and DSP units. We run the design at 100MHZ. Note that around 8% ALUT is used by the Hyper-Transport core and SRAM interface cores in the framework, thus the design itself consumes less than 20K ALUT. Table II shows the device utilization of our design.

We first conduct our experiment on a layout design with size 200um*200um with the setting shown in Section 2.3 where we simulate each unit of image region of 1000nm by 1000nm, and the range of the kernel is 2000nm by 2000nm.

We generate the layout with different layout density N. Figure 18 and 19 depict the speedup curve of the FPGA accelerated version versus the pure software implementation running on the Opteron CPU. The software implementation runs on the same development box of XD1000 with AMD Opteron

Table III. Running Time Comparison with or without Accelerator, Single Kernel

N	wo accelerator	w accelerator	speedup
5	4.16	1.44	2.89
10	8.01	1.44	5.56
25	19.94	1.46	13.65
50	39.88	2.61	15.27
75	59.80	3.86	15.49
100	79.74	5.16	15.45
125	99.64	6.41	15.54
150	119.63	7.71	15.51
175	139.45	8.99	15.51
200	159.44	10.27	15.52

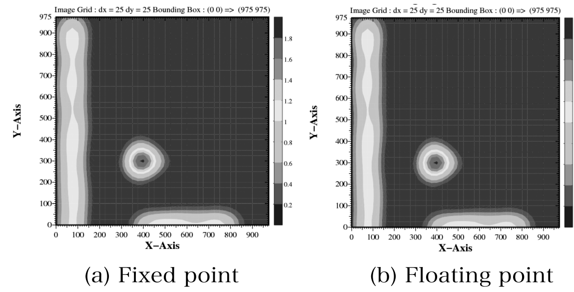
Table IV. Running Time Comparison with or without Accelerator, Multiple Kernels, N=200

No. of Kernel	wo accelerator	w accelerator	speedup
1	159.44	10.27	15.52
2	274.27	20.55	13.34
4	606.37	41.12	14.74
6	885.71	61.70	14.35
8	1216.00	82.29	14.77
10	1572.00	102.80	15.29
12	1806.00	123.46	14.62

248 (2.2GHZ) 4G DDR memory and is compiled through gcc -O3. The measured speedup factor is around 15. Note that for a very small N, for example, $N \leq 10$, the speedup we get is small due to the overhead in communications. For a moderate N, we can keep a speedup around 15, as the communication time is smaller than the computation time. We first just use one kernel for simulation. Table III shows the measured running time and speedup with different N. Then we keep the N fixed and change the number of kernels. The data is shown in Table IV.

One limitation of our current design is that we assume the three arrays used for computing using one kernel can be fitted into the on-chip RAM of the FPGA. However, we are not able to fit a larger partitioning with the current setting of input/output data size, although the overall on-chip memory bits have not exceeded the memory bits available in the device. The reason is that around 60% of the on-chip memory in the device is M-RAM, and each M-RAM can store only one memory partition. The more partitioning we use, the larger the percentage of partitioned arrays are put into the remaining M-4K and M512 blocks, which will increase the difficulty of fitting the design.

It is possible to fit the design and achieve a higher bandwidth and speedup with a larger partitioning scheme, such as an 8 by 8 partitioning or even larger partitioning scheme, if we use a smaller kernel array or decreased resolution. Another way is to further partition the kernel array and computation into multiple parts and only load one part into the on-chip RAM and only do the computation that uses that part for one time.



(a) Fixed point (b) Floating point

Fig. 20. Contour graph using fixed point or floating point computation.

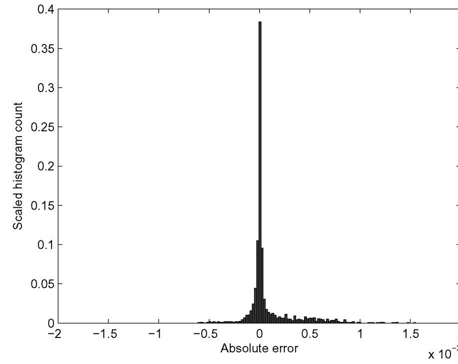


Fig. 21. Error distribution for the contour graph.

5.2 Accuracy of the Fixed-Point Computation

Using fixed-point computation will not have a big impact on accuracy. We measured the error of the approach versus the software implementation using all floating point operations, and the absolute error is usually within 1% for the pixels with bright intensity. The relative error for pixels with very small/weak intensity, on the other hand, might be larger because of the truncation error. Figure 20 shows two plotted pictures of a 1000nm by 1000nm region obtained via either software or hardware. We can see that almost no difference can be observed in the contour graph. We also plot the error distribution of the contour graph in Figure 21. From the figure, we see the maximum absolute error of all the pixels is around 10^{-3}. Besides, most of the pixel tend to have a even smaller error, which is in line with our analysis in Section 2.2. Note that the maximum intensity of that contour graph is 1.83, and the relative error is indeed very small.

5.3 Comparison with the FFT-Based Approach and Other Acceleration Techniques

There is no prior published work on accelerating the polygon based lithography image simulation. However, the 2D FFT-based image convolution has been extensively studied in various platforms. We list them here for reference. Table V shows the measured/expected performance rate on various platforms (and we

Table V. Performance Rate (Mpixel/s) Comparison of Various Algorithm and Platforms

N	polygon software	polygon FPGA	polygon GPU	2D-FFT software	2D-FFT FPGA [Mencer and Clapp 2007]	2D-FFT GPU [Podlozhnyuk 2007]
10	8.0	44.4	32.4	0.2	1.1	3.4
50	1.6	24.5	10.8	0.2	1.1	3.4
100	0.8	12.4	6.7	0.2	1.1	3.4

assume only one kernel is used here). We use FFTW [Frigo and Johnson 2005] in the software implementation of the 2D FFT-based convolution. Note our setting uses different grid size for imaging and kernel, say a 5nm grid on the kernel/eigenvectors and layout corners and a 25nm grid for the simulated image, but FFT needs the same grid size for computation. If we use the finer grid size among the two(the kernel grid), it will report the effective performance rate shown in the column of the table. If we use the coarser grid, the effective performance rate will be 25X to 30X larger, but it might lose some accuracy in representing the objects and might cause some accuracy loss in the simulated image. It might be more fair to compare all implementations with a same grid size for kernel and image, yet we do not have data of the FPGA implementation for this setting.

From Table V we can see, for a moderate density N around 50 to 100, while the polygon-based approach is not as fast as the FFT-based approach using 25nm in the object and eigenvectors, it is much faster than the FFT-based approach using a 5nm grid. (Note the extra overhead for the FFT-based approach: conversion from the GDSII to the object image, is not included.) In terms of accelerated simulation, our implementation can achieve up to 15X speedup over the software implementation, while the FPGA accelerated FFT-based 2D convolution only reported around 5X in the single precision and around 10X in the 16-bit precision [Mencer and Clapp 2007] using a Virtex-4 device. We also notice that recently FFT-based 2D convolution is shown to achieve very high FLOPS [Podlozhnyuk 2007] on NVidia G80 with the help of the CUDA Toolkit and CUFFT library; if we use the coarser grid size, it can achieve 90Mpixels/s.

As the algorithm in Figure 2 is naturally data parallel, we also implement the algorithm using CUDA. CUDA mainly uses the SPMD(single program multiple data) model. Each thread has a thread ID and each thread can use the ID to access different data and perform subsequent computation using the data. Figure 22 shows the part of the pseudo CUDA code. The parameter $blockDim.x$ and $blockDim.y$ define the number of the threads in one dimension and *gridDim.x* defines the number of thread blocks. This code will launch $gridDim.x * blockDim.x * blockDim.y$ threads. Further optimizations need to determine the locations for the array data, which affects the effective internal bandwidth for the application. We place the image partial sum array in the shared RAM in the threading block, place the kernel array in texture memory, and place layout corner array in global memory. The overall size of the kernel array is too big to be put in the shared memory. Accessing kernel data via texture caching might not give as large a bandwidth as the carefully partitioned on-chip memory of FPGA, but the massive threading offsets the possible latency in memory access.

```
......
  //Core computation using CUDA
  for(n=blockIdx.x;n<4N;n+=gridDim.x)
   for(x=theadIdx.x;x<pixel_max;x+=blockDim.x)
    for(y=theadIdx.y;y<pixel_max;y+=blockDim.y)
    {
     addr_x=5*x-rect_x[n]+c;
     addr_y=5*y-rect_y[n]+c;
     I_k[x][y]
        +=(-1)^n*kernel[k][addr_x][addr_y];
    }
......
```

Fig. 22. Pseudocode for the core computation using CUDA.

We test the performance on a 8800GTS video card. The current measured speedup we get for the design is around $8X$. Consider the usage of a higher end card, for example, 8800GTX, GTX280 and more tuning possibilities, we expect the speedup for our litho design using NVIDIA GPUs should be somewhat the same against our accelerator design using FPGA. Note that GPUs also have their advantage on floating-point, while FPGA design usually needs to use fixed-point for area efficiency.

External IO bandwidth is critical for certain applications. In this application, Section 3.3 gives the estimate on the communication for the FPGA design. The communication time is less than the computation time and the overlapping scheme removes the communication from the critical path. For the GPU design using CUDA, the (external IO) communications are not overlapped with the computation, but the peak bandwidth of PCI-e x16 is 4GB/s and is sufficiently fast for this application. This makes the communication time only consist of less than 1/10 of the total execution time.

In terms of power consumption, the power for this FPGA design reported by Quartus Power Analyzer tool is 6.2W and the peak power for this FPGA device is roughly 25W. The TDP(Thermal Design Power) of the Opteron 248 CPU is 95W, and the TDP of the 8800GTS GPU is 147W. (It is difficult for us to measure the actual power for the CPU and GPU at runtime, thus we put the TDP here.) We can see the power consumption of either GPU or CPU is much larger than the FPGA device. However, the FPGA coprocessor is plugged in a dual CPU motherboard. If we count the power consumption of other part in the system, for example, chipset, hard drive in the comparison, the gap on power consumption is not very large. If we consider the possibility of using multiple devices(more than one FPGAs or more than one GPUs), the gap will again become significant. In our design, FPGA platform could deliver similar performance with much less power consumption.

In terms of ease of use, the CUDA toolkit, as a C development environment for NVIDIA GPU, is very friendly to use; this makes GPGPU platform very attractive. Users need to rewrite their code to the SPMD form and tune the locations of array data for performance. This tuning needs the deep knowledge

of GPU hardware. Traditionally, using FPGA for computing has been much more difficult than GPU and CPU. C to RTL automation tools help to bridge the gap, and will also make the FPGA-based computing platform increasingly attractive. We present our paper describing our litho design on FPGA using C to RTL automation tools. We also point out that currently it also needs many hardware-oriented refinements or tuning.

In terms of cost for reaching similar performance, high-end FPGAs for HPC markets are still relatively expensive, thus FPGA designs need to demonstrate a larger speedups to justify a more competitive position. This is often the case for examples with high degree of bit-level parallelism and data/task parallelism. (Note that the design we present does not have bit-level parallelism.)

6. CONCLUSIONS AND FUTURE WORK

This article presents a design for accelerating lithography aerial image simulation using a polygon-based simulation model. The adequate memory banking scheme for the on-chip memory can improve the load balance and ensure a decent speedup. A 5 by 5 partitioning design can achieve around 15X speedup over software implementation. We also compare against other algorithms and implementations on GPU.

We see several opportunities for making improvements over the current design. One is that the 2D-ring structure might have a large interconnect delay in the feed-back path; thus buffers need to be explicitly inserted, especially when there is a larger partitioning. Another improvement concerns the assumption that at least one kernel can be loaded into the on-chip RAM. This might not be always true for different settings. Thus, further partitioning of the kernel and computation should also be implemented.

Current C to HDL code translation and synthesis tools have already enabled the designer to write and maintain the algorithm and logic in high level, purely in C, and help reduce the development cycle. However, our experience shows that a certain amount of effort is still needed to find a larger parallelism through manual refinement of the C code. More automation is needed for the extraction of systematic parallelism. Also for the mapping of memory models, the compilation tool should not simply convert one array in C into a memory block in HDL, but should provide more flexibility and optimizations on memory models that could possibly do a better job of handling the specific addressing patterns in our design.

ACKNOWLEDGMENTS

The XD1000 development system is obtained through Xtremedata university program with the joint effort by Altera Corporation, AMD, Sun Microsystems and Xtremedata Inc. We also thank AutoESL Design Technologies for providing the C to HDL tool AutoPilot^{TM}.

The authors would like to thank Alfred Wong from Magma Design Automation for giving us sample kernels data and engaging in valuable discussions. Also, we would like to thank Zhiru Zhang from AutoESL Design Technologies for his very useful suggestions concerning the tool use and code refinements.

REFERENCES

CAO, Y., LU, Y.-W., CHEN, L., AND YE, J. 2004. Optimized hardware and software for fast full-chip simulation. In *Proceedings of SPIE: Optical Microlithography XVIII*. Vol. 5754, 407–414.

COBB, N. B. 1998. Fast optical and process proximity correction algorithms for integrated circuit manufacturing. Ph.D. thesis, University of California, Berkeley.

COBB, N. B. AND ZAKHOR, A. 1995. Fast, low-complexity mask design. In *Proceedings of SPIE: Optical/Laser Microlithography VIII*. Vol. 2440, T. A. Brunner, Ed. 313–327.

CONG, J. AND ZOU, Y. 2008. Lithographic aerial image simulation with FPGA-based hardware acceleration. In *Proceedings of the 16th International ACM/SIGDA Symposium on Field Programmable Gate Arrays (FPGA'08)*. ACM, 67–76.

DOGGETT, M. AND MEISSNER, M. 1999. A memory addressing and access design for real time volume rendering. In *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS'99)*. 344–347.

FRIGO, M. AND JOHNSON, S. G. 2005. The design and implementation of FFTW3. *Proc. IEEE 93*, 2, 216–231.

MACK, C. A. 2005. Lithography simulation in semiconductor manufacturing. In *Proceedings of SPIE: Advanced Microlithography Technologies*. Vol. 5645, 63–83.

MENCER, O. AND CLAPP, R. G. 2007. Accelerating 2D FFTs and convolutions for seismic processing. Brief notes, Maxeler Technologies.

MENTOR. 2004. Datasheet of Calibre nmOPC. Mentor Graphics Corporation.

PATI, Y. C. AND KAILATH, T. 1994. Phase-shifting masks for microlithography: Automated design and mask requirements. *J. Opt. Soc. Am. A 11*, 9, 2438.

PODLOZHNYUK, V. 2007. FFT-based 2D convolution. NVIDIA white paper.

TANSKANEN, J. K., SIHVO, T., AND NIITTYLAHTI, J. 2004. Byte and modulo addressable parallel memory architecture for video coding. *IEEE Trans. Circ. Syst. Video Technol. 14*, 11, 1270–1276.

UZUN, I., AMIRA, A., AND BOURIDANE, A. 2005. FPGA implementations of fast fourier transforms for real-time signal and image processing. *IEEE Proc. Vision, Image, Signal Process. 152*, 3, 283–296.

WANG, Y.-T., TSAI, C.-M., AND CHANG, F.-C. 2006. Lithographic simulations using graphical processing units. United States Patent Application 20060242618.

WONG, A. K.-K. 2005. *Optical Imaging in Projection Microlithography*. SPIE Press, Bellingham, WA.

WONG, A. K.-K. 2007. Private communication. Magma Design Automation Inc.

YEUNG, M. S. 2003. Fast and rigorous three-dimensional mask diffraction simulation using battle-lemarie wavelet-based multiresolution time-domain method. In *Proceedings of SPIE: Optical Microlithography XVI*. Vol. 5040, 69–77.

YU, P. AND PAN, D. Z. 2007. A novel intensity based optical proximity correction algorithm with speedup in lithography simulation. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD'07)*. 854–859.

Received June 2008; revised September 2008, November 2008; accepted December 2008