Lithographic Aerial Image Simulation with FPGA-Based Hardware Acceleration

Jason Cong and Yi Zou
Computer Science Department
University of California, Los Angeles
Los Angeles, CA 90095, USA
{cong, zouyi}@cs.ucla.edu

ABSTRACT

Lithography simulation, as 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 co-processor acceleration is very attractive for obtaining increased computational performance with reduced power consumption. This paper describes an 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 identified the need for manual architecture-level exploration for parallel implementations.

Categories and Subject Descriptors

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

General Terms

Algorithms, Performance, Design

Keywords

Lithography simulation, Co-processor acceleration, FPGA

1. INTRODUCTION

In the semiconductor industry, optical lithography is the technology used for printing the circuit patterns onto wafers. As the technology scales down, and the feature size is even smaller than the wavelength of the light employed, 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 as 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 [8]. 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 [14] is a very accurate approach but is so expensive that it can only simulate small regions and designs.

The coherent decomposition method can balance the accuracy and running time better, 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 image is large. As the technology scales down and the accuracy requirement goes up, it will be more challenging to meet the specific turn-round time.

Leading commercial computational lithography products have already started to use special co-processor acceleration to further accelerate the computation. Brion Technologies 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 [5]. Mentor Graphics uses the computing power of Cell Broadband Engine to accelerate the computation in their nmOPC product [1].

This paper presents yet another hardware implementation for accelerating lithography imaging simulation on FPGA platform. Unlike the image-based approach Brion takes which ultimately relies on the accelerated performance of 2D-FFT, we use the polygon-based approach [13] instead. The polygon-based approach has comparable or even better performance than image-based approach in software implementation. The polygon-based approach can be implemented without any floating point operations, and can be better paralleled and accelerated by utilizing the high bandwidth of on-chip memory in FPGA.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
FPGA'08, February 24–26, 2008, Monterey, California, USA.
Copyright 2008 ACM 978-1-59593-934/08/02 ...$5.00.

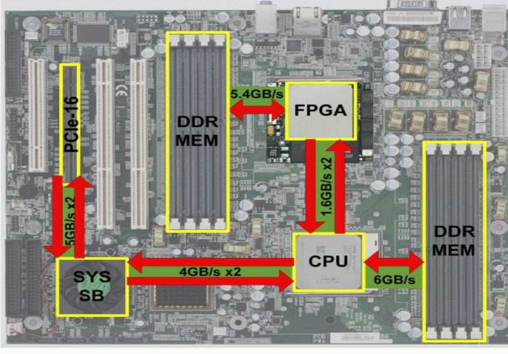


Figure 1: XD1000 system diagram

Moreover, we also leverage up-to-date C to HDL compilation tools to write all our design in C, whereas the FPGA accelerator design by Brion was based on complete manual RTL coding [5]. The challenge for this design is still to develop 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 paper is organized as follows: Section 2 describes our hardware platform. Section 3 describes the basic equations for lithography simulation and discusses the trade-offs between image-based approach and polygon-based approach. Section 4 describes our entire design, specifically on the design of memory partitioning. Section 5 discusses our experience on using C to HDL compilation tools, and the C code refinements in implementing the design. Section 6 describes our experimental results, and section 7 concludes the paper.

2. FPGA CO-PROCESSOR PLATFORMS

Figure 1 is the system diagram of the XtremeData's XD1000 development system [4], which is the hardware platform we use. This development system uses a dual Opteron motherboard and one Opteron is replaced by an XD1000 co-processor module. The XD1000 co-processor is built based on Altera's largest FPGA in Stratix II family: EP2S180, and is socket compatible with Opteron socket 940. The FPGA co-processor communicates with the host Opteron CPU via Hypertransport links. Comparing with traditional FPGA boards/cards that are loosely coupled with host processor, the socket compatible FPGA co-processor provides a low latency and high throughput solution for accelerator design.

3. BASICS FOR AERIAL IMAGE SIMULATION

3.1 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 eigen values. If we only keep K significant eigen values and eigen vectors, 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 eigen value, $O(x,y)$ is the object function (field) and $\phi_k(x,y)$ is the kth eigen vector. The symbol $\otimes$ denotes convolution (2D image convolution). For more details on the derivation of the coherent decomposition method, please refer to [6].

2D image convolution can be implemented through 2D FFT. We can first transform the padded object pattern (see section 4.1) and the eigen vector into the frequency domain via 2D FFT. (Note the FFT of the eigen vector ϕ_k only needs to be computed once and can be reused.) Then we multiply eigen vector and object pattern in the frequency domain. We can obtain the convolution result via an inverse FFT of the multiplied result. This is the image-based simulation which is used by [5].

As the actual layout is solely composed of polygons (or rectangles if we perform polygon decomposition on the layout), the convolution of different sizes of polygons/rectangles can be pre-computed and stored. 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 described via *quadrant functions*.

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

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:

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

where

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

is the convolution of the quadrant function with the kth eigen vector. This is the polygon/rectangle-based algorithm we use, and it is described in more detail in [13].

Note that for rectilinear polygons, an equation similar to Equation 2 can be written using quadrant function on each vertex of the polygons. In Figure 2, we label the + and − on the vertexes of rectilinear polygons (rectangle is simply a special type of rectilinear polygons, and + and − are just the signs for the look-up value for the vertexes, see Equation 2 and Equation 3 for the rectangle case). Using rectangles or polygons will not alter the overall algorithm and design

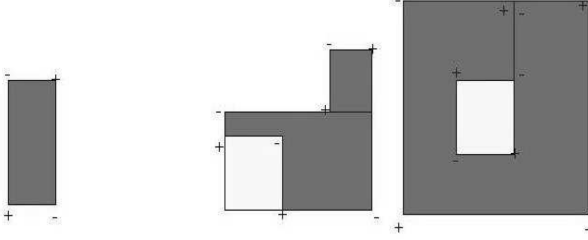


Figure 2: Rectilinear polygons can be processed similar to rectangles

presented following. For simplicity, 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 directly without the need for rectangle decomposition.

3.2 Image based simulation versus polygon based simulation

It is worthwhile to briefly discuss the trade-offs between the 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 the conversion from the GDSII to the object image is also computationally expensive. The 2D-FFT algorithm, although already 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 truncation 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 $4N*2^{-p}$ and the actual error is even much smaller than that conservative bound due to statistical cancellations.

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 among 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 via FFTW [7], and tested that on kernels with size 400 by 400, and found out the running time is comparable to a polygon-based method with a moderate density. The polygon based approach will require less computation and run faster for layers that are not very dense.

Regarding the FPGA platform, the acceleration on 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 2 to 8X speedup is seen on accelerating FFT on FPGA platforms via parallel implementation [11]. On the other hand, for the polygon-based approach, as the convolution on the quadrant function can be pre-computed and reused multiple times, the remaining computations only involve table look-up and simple addition/substraction 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.

3.3 Detailed settings for the imaging equation using the polygon-based approach

We assume the convolutions of eigen vectors and the quadrant functions are already pre-computed, 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 are our settings for our implementation and experiments, and also are a result of the practical requirements in our industry collaborator [12]. But our architecture certainly is not confined to these settings and can be extended to other settings. But we need to modify several parameters such as the number of partitions, array size, and recompile/re-synthesis the design.

4. FPGA BASED DESIGN FOR THE IMAGING SIMULATION

4.1 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 be considered due to the locality of the litho effects. For example, for a kernel ranges among a 2000nm by 2000nm area, suppose we want to compute a 1000nm by 1000nm image region, an object pattern ranges among 3000nm by 3000nm needs to be taken into computation. The reason is that the objects that are far away from the current pixel and out of the interaction range 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.

4.2 Rearranging the nested loop

Now we devote ourselves to implementing the nested loop corresponding to Equation 3, which is described in Figure 3, where c is a constant, and $rect_x$ and $rect_y$ are arrays for the coordinates of rectangle corners. Note this is just the code for simulating one region of an image and there is another outer loop over the pseudo code in Figure 3 for changing the current image region where the input N and $rect_x$ and $rect_y$ shall all be changed for different image regions. Here the *pixel_max* is the number of pixels in either X or Y direction. And we use $5*x$ and $5*y$ in the code as we use an image grid on 25nm and a kernel grid on 5nm. 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.

First, we look at the choices for the outer-most loop. As the whole nested loop will need the data in the kernel array,

```
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_x];
     }
     I[x][y]+=I_k[x][y]*I_k[x][y];
   }
  }
```

Figure 3: Pseudo-code 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_x];
      }
     //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];
 }
```

Figure 4: Pseudo-code for the rearranged nested loop

which shall be reused for the image computation with different image regions or different pixels and layout corners, we would like to pre-fetch the kernel array and store the kernel array in the on-chip RAM of the FPGA. As the total size of on-chip RAM of FPGA is limited, it is unlikely that all the kernels can be put in, but in our case at least one kernel can be put in. Thus, we would like to make the loop over different kernels the outer-most one.

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 as 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 4 is the nested loop after the loop interchange.

The illustration of the computation is shown in Figure 5. The address of kernel data depends on the value of the rectangle corner. For one specific corner, the address is just an affine mapping over the pixel index x and y. After the rearranging of the nested loop, the key problem is to design and implement the inner loop: the loop over different rectangle corners and image pixels.

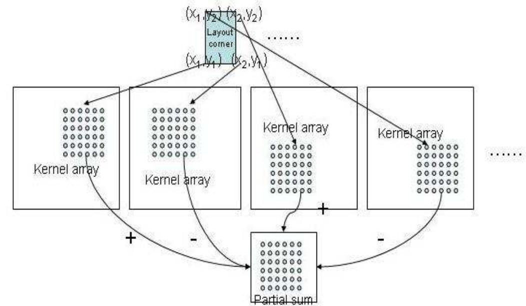


Figure 5: Computation of the inner nested loop

4.3 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 software component running on processor; the software component parses and provides input data to the hardware component and also performs the square operation and stores the results.

4.4 Exploring the parallelism via memory partitioning

Typically, the accelerator on the FPGA platform is able to explore task parallelism, data parallelism, and instruction parallelism. Instruction parallelism, is already realized as the loop pipelining optimization technique in several high level synthesis tools, e.g. Impulse C™[2] and AutoPilot™[3] which we use in this work. Task parallelism needs to explicitly write multiple processes/tasks, and needs to consider inter-process synchronization and arbitration of shared resources. Data parallelism, on the other hand, widely used in SIMD instructions or GPGPU accelerators, tries to use a same or similar program/code to cope with multiple data.

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 6 shows a 4-way naive partitioning. And each processing element (PE) contains one partition of kernel array and one partition of partial image array, and each PE is responsible for the computing of one partition of partial image array. We then allocate four PEs in the FPGA and schedule the operations which will read or write different memory blocks into different computation stages of each PE (shown in Figure 7).

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. If the whole loop make heavy use of one or several specific partitions, the benefit or speedup using partitioning might become degraded.

Later, we decided to mainly borrow the idea from data

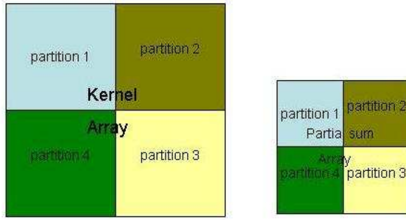


Figure 6: Naive partitioning based on geometric locations

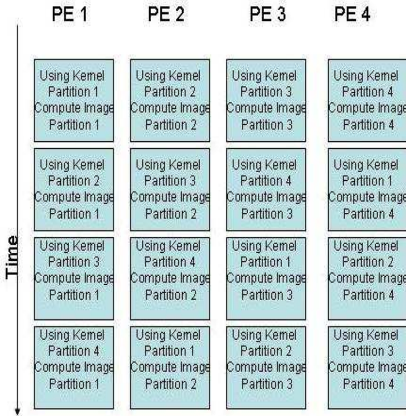


Figure 7: Block scheduling for naive partitioning

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 8, where we unroll once in x direction and once in y direction. 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 initial interval of pipelining and not contribute much for the overall latency. The goal of the memory partitioning is to make sure the correspondent memory accesses in the unrolled loop are partitioned into different memory blocks.

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 array for different layout corners. 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.

To get 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, and the corresponding addresses to get the data in the kernel array have a regular pattern (shown in Figure 5). A better partition scheme should be able to evenly distribute the memory access pattern shown in Figure 5 into multiple

```
......
  //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];
    }
......
```

Figure 8: Pseudo-code for the partially unrolled nested loop

memory blocks, regardless of the location value of specific layout corners. Therefore, we choose to use an interleaving partition scheme.

We still take 4-way partitioning as an example, and the illustration is shown in Figure 9. *The same color/texture means the data are physically stored in the same memory 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. In our case, the kernel array is on a 5nm grid and the image array on a 25nm grid. We hope memory accesses in the most inner unrolled loop in Figure 8 are always in different memory partitions (different colors in Figure 9) thus can be scheduled to execute at the same cycle. Figure 9 could be obtained via first *coloring* the bottom-left 5 * 5 corner blocks in the kernel array, and *coloring* other blocks in an interleaved fashion, to ensure that the four memory access in the inner unrolled loop should be in different memory blocks. Note the Figure 9 only shows a 4-way 2 * 2 partitioning corresponding to the partially unrolled loop in Figure 8, 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 9, so that we can write the output data into the array concurrently. Layout corner array does not need partitioning as the loop over layout corners is an outer loop.

4.5 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.

Again take the 4-way partitioning in Figure 8 and Figure 9 as an example. In Figure 10, 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,

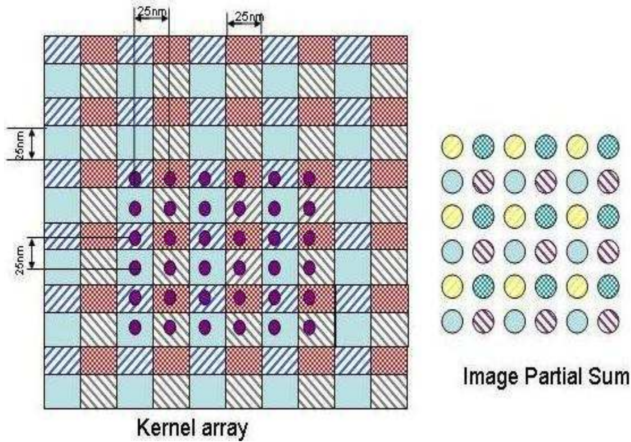


Figure 9: 4 way(2 by 2) memory partition scheme for load balancing

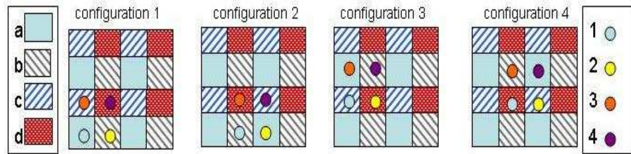


Figure 10: Address generation and output data multiplexing

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 the which configuration is among the four cases in the 2 by 2 partitioning design in Figure 10. Later for each configuration, there is a mapping function to transform the original address into the mapped address.

4.6 Output data multiplexing

The data we fetched from the partitioned memory blocks needs to go through multiplexing before it is sent to accu-

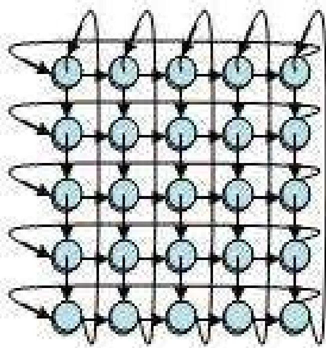


Figure 11: Ring based data multiplexing

```
//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;
                }
}
```

Figure 12: Pseudo code for 2D ring multiplexing for 5 * 5 partitioning

mulator, because still the computation of one partition of image partial sum might require data in different partitions of kernel array. For the 2 by 2 partitioning shown in Figure 10, if the data from the four memory blocks are $data_a$, $data_b$, $data_c$, $data_d$ and the data we can directly send to the accumulator are $datamux_1, datamux_2, datamux_3, datamux_4$. We denote the multiplexing logic as

$$
\begin{aligned}
& datamux_1, datamux_2, datamux_3, datamux_4 \\
&= (data_a, data_b, data_c, data_d) \quad (configuration\ 1) \\
&= (data_b, data_a, data_d, data_c) \quad (configuration\ 2) \\
&= (data_c, data_d, data_a, data_b) \quad (configuration\ 3) \\
&= (data_d, data_c, data_b, data_a) \quad (configuration\ 4)
\end{aligned}
$$

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 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 need to be shifted, and *ring_shift_x* means every data in the 2D-ring is assigned the value on its circular left side, and *ring_shift_y* means every data in the 2D-ring is assigned the value on its circular upper side. The whole shifting is done in a multi-cycle 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 multiplexing block can be implemented as a unit that have a multi-cycle latency but with a one cycle pipeline initiation interval (similar to many floating point IPs).

4.7 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 much because the whole nested loop (over different rectangles and image pixels) can be pipelined and can achieve an initial interval equal to one. 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 13) or realized as explicit control flow (Figure 14). In

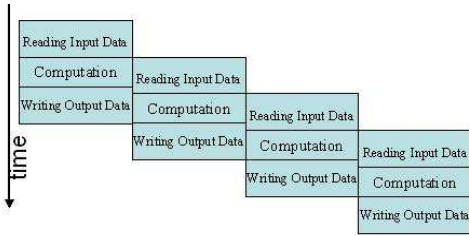


Figure 13: Block pipelining/overlaping communication and computation

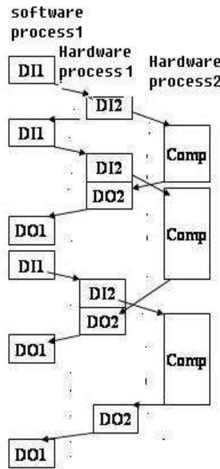


Figure 14: Explicit control flow on overlapping communication and computation

Figure 14 $DI1$ 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 serves as input data array and output data array for the computation of one region respectively. 2X storage space is used for ping-pong buffer while one is used in the current computations and another is used for sending/receiveing data. Using ping-pong buffers can ensure the overlapping will not alter the data that is needed for current computation.

4.8 Using wider bits 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 the each memory block can have two ports, we try to utilize both of the two ports of the *kernel* memory. But at the output side, the image partial sum memory needs to store a

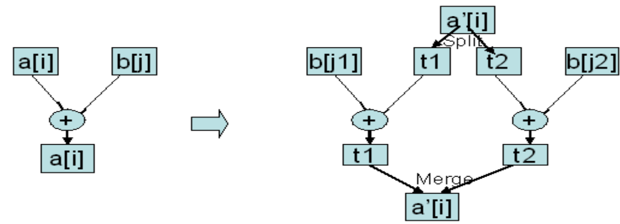


Figure 15: Using wider data to balance the port usage of the memory blocks

wider bit of data to avoid port conflict. This would provide a 2X more parallelism without further partitioning of the memory blocks. Figure 15 is the illustration of computation elements using wider data.

5. LEVERAGING C TO HDL COMPILER FOR HARDWARE GENERATION

The whole algorithm is written in C so that we can leverage up-to-date C to HDL translation tools. The tools we use are Impulse C [2] and AutoPilot [3], both of which can take ANSI C as input and generate synthesizable and optimized RTL. Impulse C has built-in Platform Support Packages (PSP) for multiple platforms and could generate the whole system including both software component and hardware component for various environments very easily. Also currently it is the only vendor that supports the XD1000 system we use. AutoPilot currently has a SOPC builder interface, while the complete platform support of XD1000 is ongoing, but it provides a wider support for ANSI C features such as structures and pointers and functional calls. It also performs a better analysis and optimization. For example, Inpulse C will need the user to explicitly specify whether specific array variables are non-recursive for pipelined loop generation (otherwise it will do a conservative estimation and generate a pipeline with large initial interval), while AutoPilot can do the analysis without explicit specification. Both tools can be used and guided to generate the hardware code meeting desired requirements.

5.1 C-based hardware generation and optimization without code refinement

Originally, the core C code might be as short as shown in Figure 3 or Figure 4. 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 much larger degree of parallelism to get speedup.

Both tools provide loop unrolling and pipelining techniques 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

Table 1: Device information of EP2S180

ALUT	M512 blocks	M4K blocks	M-RAM blocks	Total Memory Bits
143,520	930	768	9	9,383,040

ports and there is not much data reuse for the loop. Pipelining did help as it could generate a pipelined loop with a small initial 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 can not completely compensate for the low clock frequency of FPGA to get a decent speedup.

5.2 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. All these refinements are also 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 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 need less than a thousand line of code, and the generated RTL contains several tens of thousands lines of code. Thus using C based design shall have significant savings in turns of design effort comparing with pure manual RTL design.

5.3 Manual RTL tweaking

Function block pipelining, however, is not yet directly supported by either tool. The explicit control flow (Figure 14) can actually be described by Impulse C's CSP (*Communication of Sequential Process*) model, but the tool will face difficulties on the RTL generation of a shared global memory (ping-pong buffers in our design). Take the ping-pong buffer of image partial sum as an example, in one hardware process we use two ports for computation involving accumulator; in another hardware process, we use one port to write the data to the shared SRAM. The ports are not conflicted as we use explicit control flow to avoid this, but the tool is unaware of that and might assume there are still port conflicts. We instead put instances of the ping pong buffer in each hardware process in C code (declare the arrays as local arrays rather than global arrays) and generate RTL for two hardware processes. Then we manually merge the RTL of the memory access to form a correct implementation.

6. EXPERIMENTAL RESULTS

We implement the algorithm on Xtremedata's XD1000 development system [4]. Table 1 shows the device information of EP2S180.

We use Impulse C and AutoPilot to synthesize the C code and generated RTL is compiled through Quartus II 6.0. We use similar but not identical C code to feed the two tools, and do several tool-specific tuning. The performance/speedup is somewhat the same, as both of them can generate desired pipeline initiation interval and ways of data communications.

Table 2: Device utilization of the design with 5 by 5 partitioning

Tool	ALUT	Memory Bits	Fmax(MHZ)
Impulse C	25042	2,972,876	115.58
AutoPilot	24797	2,972,876	115.00

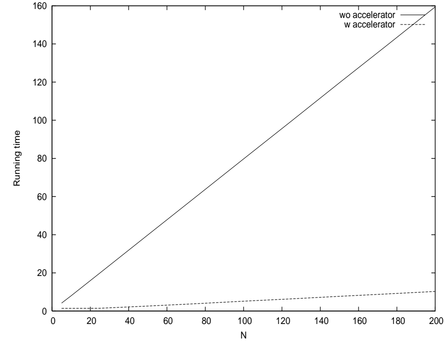


Figure 16: Running time comparison with or without accelerator, single kernel

6.1 Speedup measurement

We use a 5 by 5 partitioning scheme and it effectively drives 50 processing elements. *Kernel* array spans 2000nm by 2000nm area and is a 400*400 array containing 16-bit resolution fixed point values. The window of image region we simulate ranges from 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 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; the design runs at 100MHZ. Note that around 8% ALUT is used by the HyperTransport core and SRAM interface cores in the framework, thus the design itself consumes less than 20K ALUT.

We first conduct our experiment on a layout design with size 200um*200um with the setting shown in Section 3.2 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. Here is the running-time comparison 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 248 (2.2GHZ) 4G DDR memory and is compiled through gcc -O3. The measured speedup factor is around 15. Note for a very small N, e.g $N \leq 10$, the speedup shall be 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 3 and Figure 16 show

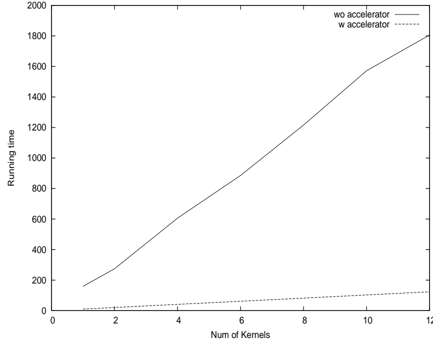
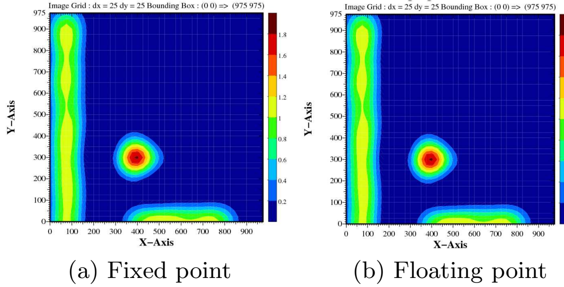


Figure 17: Running time comparison with or without accelerator, multiple kernels, N=200



(a) Fixed point (b) Floating point

Figure 18: Accuracy comparison between the fixed point and floating point computation

the measured running time and speedup with different N. Then we keep the N fixed and change the number of kernels, and the data is shown in Table 4 and Figure 17. Note the pure software version also rearrange the nested loop, the loop over layout corners is the outer loop, and the loop over image pixels is the middle loop and the loop over kernels is the inner loop. We find out this combination will give out the best run-time for software version.

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 percentage of partitioned arrays are put into the remaining M-4K and M512 blocks, which will increase the difficulty for 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.

6.2 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 within 0.3% 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 18 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.

Table 3: 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 4: Running time comparison with or without accelerator, multiple kernels, N=200

NumofKernel	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	82.29	14.77
10	1572	102.80	15.29
12	1806	123.46	14.62

6.3 Comparison with the FFT-based approach and other acceleration techniques

There is no prior published work to accelerate 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 5 shows the measured/expected performance rate on various platforms, and we assume only one kernel is used here. We use FFTW [7] in the software implementation of the 2D FFT-based convolution. Note in each column of the FFT-based simulation we give two numbers. The reason is that in the polygon-based approach, we use the 5nm grid on the kernel/eigen vectors and layout corners and the 25nm grid for the simulated image. If we use the 25nm grid for converted

Table 5: Performance rate (Mpixel/s)comparison of various algorithm and platforms

N	polygon software	polygon FPGA	2D-FFT software	2D-FFT FPGA[9]	2D-FFT GPU[10]
10	8.0	44.4	4.8 , 0.2	30.6 , 1.1	85 , 3.4
50	1.6	24.5	4.8 , 0.2	30.6 , 1.1	85 , 3.4
100	0.8	12.4	4.8 , 0.2	30.6 , 1.1	85 , 3.4

object image and sampled eigen vectors in the FFT-based approach, it will give the pixel rate of the first number. It is quite fast but it might lose some accuracy in representing the objects and might cause some accuracy loss in the simulated image. If we use 5nm grid for the object image and sampled eigen vectors, it will give the pixel rate of the second number (effective pixel rate, where we still only need the pixels on the 25nm grid).

From Table 5 we can see, for a moderate density N around 50 to 100, while the polygon-based approach is not as fast as the the FFT-based approach using 25nm in the object and eigen-vectors, it is much faster than the FFT-based approach using 5nm grid. (Note the extra overhead for FFT-based approach: conversion from the GDSII to the object image, is not included.) In turns 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 single precision and around 10X in 16-bit precision [9] using a Virtex-4 device. We also notice that recently FFT-based 2D convolution is shown to achieve very high FLOPS [10] on NVidia G80 with the help of the CUDA Toolkit and CUFFT library. We are also investigating whether the polygon-based approach can also be greatly accelerated on the GPU platforms.

7. CONCLUSIONS AND FUTURE WORK

This paper 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 balance the load better and ensure a decent speedup. A 5 by 5 partitioning design can achieve around 15X speedup over software implementation.

We need to make several 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 one kernel at least 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 showed that certain 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 which could possibly do a better job of handling the linear/affine addressing patterns in our design.

8. ACKNOWLEDGMENTS

This work is partially funded by Natural Science Foundation and also a grant from Altera Corporation and Magma Design Automation under the California MICRO program.

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 Impulse Accelerated Technologies and AutoESL Design Technologies for providing C to HDL tools Impulse C™ and AutoPilot™, respectively.

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

9. REFERENCES

[1] Datasheet of calibre nmopc. Mentor Graphics.

[2] http://impulsec.com.

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

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

[5] Y. Cao, Y.-W. Lu, L. Chen, and J. Ye. Optimized hardware and software for fast full-chip simulation. *Optical Microlithography XVIII*, 5754(1):407–414, 2004.

[6] N. B. Cobb. *Fast optical and process proximity correction algorithms for integrated circuit manufacturing*. PhD thesis, UC Berkeley, 1998.

[7] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005.

[8] C. A. Mack. Lithography simulation in semiconductor manufacturing. *Advanced Microlithography Technologies*, 5645(1):63–83, 2005.

[9] O. Mencer and R. G. Clapp. Accelerating 2d ffts and convolutions for seismic processing. Brief Notes by Maxeler Technologies.

[10] V. Podlozhnyuk. Fft based 2d convolution. NVIDIA white paper.

[11] I. Uzun, A. Amira, and A. Bouridane. Fpga implementations of fast fourier transforms for real-time signal and image processing. *IEE Proceedings - Vision, Image, and Signal Processing*, 152(3):283–296, 2005.

[12] A. K.-K. Wong. Private communication. Magma Design Automation Inc.

[13] A. K.-K. Wong. *Optical imaging in projection microlithography*. SPIE Press, Bellingham, WA, 2005.

[14] M. S. Yeung. Fast and rigorous three-dimensional mask diffraction simulation using battle-lemarie wavelet-based multiresolution time-domain method. *Optical Microlithography XVI*, 5040(1):69–77, 2003.