Accelerating aerial image simulation using improved CPU/GPU collaborative computing

Fan Zhang a,b,*, Chen Hu a, Pei-Ci Wu b, Hongbo Zhang c, Martin D.F. Wong b

a College of Information Science & Technology, Beijing University of Chemical Technology, Beijing, PR China
b Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, Urbana, IL, USA
c Synopsys, Inc., Mountain View, CA, USA

ABSTRACT

Aerial image simulation is a fundamental problem in advanced lithography for chip fabrication. Since it requires a huge number of mathematical computations, an efficient yet accurate implementation becomes a necessity. In the literature, graphic processing unit (GPU) or multi-core single instruction multiple data (SIMD) CPU has demonstrated its potential for accelerating simulation. However, the combination of GPU and multi-core SIMD CPU was not exploited thoroughly. In this paper, we present and discuss collaborative computing algorithms for the aerial image simulation on multi-core SIMD CPU and GPU. Our improved method achieves up to 160/C2 speedup over the baseline serial approach and outperforms the state-of-the-art GPU-based approach by up to 4/C2 speedup with a hex-core SIMD CPU and Tesla K10 GPU. We show that the performance on the collaborative computing is promising, and the medium-grained task scheduling is suitable for improving the collaborative efficiency.

© 2015 Elsevier Ltd. All rights reserved.

1. Introduction

In the modern integrated circuit (IC) manufacturing, optical lithography is the key technology used for transferring circuit patterns from mask onto wafers. Due to diffraction and interference of light waves, the imprint on the wafer is not completely identical with the mask pattern, commonly called optical proximity effect (OPE). As the semiconductor devices shrink, the OPE issue becomes more and more serious, and to be an important factor affecting the chip yield. Therefore, aerial image simulation that simulates the process of optical lithography is used to predict these distortions for design correction and considered as an essential step in the design for manufacturability (DFM). Aerial image can be used in mask pattern synthesis, printability analysis, optical proximity correction (OPC) to reduce yield loses and to improve printability.

The classical aerial image simulation algorithm originated from the Hopkins partially coherent equations [1]. Then, the coherent decomposition method by Pati et al. [2] decreased the simulation to a serious of image convolution and weighted merge. Compared with other simulation algorithms, e.g., experimental model method [3] and finite element method [4], it can achieve better efficiency while maintaining acceptable accuracy, and is widely used nowadays. With the expansion of the IC scale and the decrease of device size, the volume of representing data matrix rapidly increases, and makes aerial image simulation a challenge.
simulation a time-consuming task. Basically, the core of aerial image simulation is two-dimensional (2D) convolution, which involves a huge number of numerical computations. Therefore, various methods have been proposed to accelerate the 2D convolution, including lookup table based approach [5], fast fourier transformation (FFT) based approach [6].

Although the simulation efficiency is improved, it is far from industry requirements. Due to the regularity of 2D convolution, parallelization is the most straightforward idea, and can greatly shorten the simulation time with the state-of-the-art high performance computing technologies. Hence the corresponding parallel algorithms are proposed to reach tens of acceleration, such as field programmable gate array (FPGA) based approach [7], GPU based approach [8,9] and multi-core SIMD CPU based approach [10]. FPGA has been proved to be a good solution to accelerate the aerial image simulation, but the harder programmability, refinement and integration with other EDA tools make us concentrate on the CPU and GPU parallel algorithms study. The traditional CPU parallel methods include open multiple processing (OpenMP) with multi-cores, message passing interface (MPI) with multi-CPUs, and cluster computing with multi-processors. These three parallel computing technologies are based on CPU platform, which is designed for the control and logic processing. The accelerating effects of these methods are proportional to the number of CPUs. In general, these methods can improve the computational efficiency by an order of magnitude. Nevertheless, the design pattern is not suitable when aiming to increase by more than two orders of magnitude.

Due to the outstanding capacity of GPU parallel processing, the general-purpose computation on GPU (GPGPU) technology has drawn an increasing attention. GPGPU has been applied to remote sensing processing [11], image processing [12], mathematics [13] and other scientific areas. In our previous work [8], parallel simulation in one GPU achieves $50 \times$ to $60 \times$ speedup over the CPU based serial algorithm. Meanwhile the error of the approach is on the order of $10^{-7}$, which can meet the industry requirement. However, the CPU, as one kind of computing resource, was ignored in the GPU-based aerial image simulation. Therefore, we further dig up the potential of CPU parallel computing power, and introduce the multi-core single instruction multiple data (SIMD) extensions instructions method to accelerate the simulation. In recent years, the SIMD instructions method has been applied to the classical topics, such as fast fourier transform (FFT) [14], finite-difference time domain (FDTD) [15], image processing [16] and graphics application [17], and achieves $4 \times$ to $8 \times$ speedups. Through deep optimization, our method [10] reaches an amazing $73 \times$ speedup over the serial approach with six-core and streaming SIMD extensions (SSE), which is almost the same level of GPU parallel acceleration. Dr. Hwu believes that the multicore CPU SIMD parallelism is an important trend in parallel computing, and can be effectively covered by manycore architectures [18]. Inspired by the words, we try to develop a new method to make multi-core SIMD CPU and manycore GPU work together to simulate the aerial image more faster.

In recent years, the hybrid CPU–GPU programming is attracting the attention of many researchers. There is a growing trend towards the institutional use of multiple computing resources (usually heterogeneous) as a sole computing resource [19]. Although the CPU–GPU collaborative computing has been well studied [20–24], most of these research ignore the SIMD method in CPU parallel. With introducing SIMD instructions, the computing power gap between CPU and GPU will be tremendously narrowed. Therefore, the classical heterogeneous computing algorithm should be restudied to fit the new condition. Moreover, this is the first time to apply heterogeneous computing to the aerial image simulation, not only deeply exploits the computer resources to accelerate simulation, but also lays the foundation for the distributed computing and cloud computing. Therefore, we propose an aerial image heterogeneous simulation approach based on multi-core SIMD CPU and GPU. Compared to previous work [8,10], we make the following contributions.

- Introducing the AVX instructions method to further optimize the aerial image parallel simulation algorithm.
- Proposing the collaborative computing based image simulation and applying the dynamic task partitioning and scheduling ideas for load balancing to heterogeneous multi-core SIMD CPU and GPU system.
- According to different simulation scales and hardware configuration, starting different heterogeneous computing strategies, such as multi-core SIMD CPU and one GPU, and two GPU.

The rest of the paper is organized as follows. Section 2 briefly introduces the aerial image simulation algorithm and the parallelization with SIMD CPU and single instruction multiple thread (SIMT) GPU. Section 3 presents the proposed multi-core SIMD CPU and GPU collaborative computing algorithm for aerial image simulation. Then, the experimental results and analysis are discussed in Section 4. Finally, the conclusions are drawn in Section 5.

2. Related work

In this section, we will briefly introduce some background knowledge on aerial image simulation and its GPU parallel implementation, multi-core SIMD CPU implementation, respectively.

2.1. Aerial image simulation

Aerial image simulation plays an important role in the normal manufacturability analysis and lithography related design-manufacturing co-optimization. Its task is to compute the light intensity distribution on the wafer, namely the aerial image, when the lighting and mask information are provided. The aerial image simulation, as illustrated in Fig. 1, is derived
from Hopkins partially coherent imaging equation, which is approximately decomposed to a set of linear subsystems by Singular Value Decomposition (SVD) algorithm [25–27]. In effect, the Hopkins transmission cross coefficients (TCCs) in imaging equation are generated as a matrix by the SPLAT program [28, 5]. After the decomposition on TCCs, the aerial image intensity \( I(x, y) \) at wafer \((x, y)\) can be simplified as:

\[
I(x, y) = \sum_{i=1}^{k} \lambda_i |\phi_i(x, y) \otimes f(x, y)|^2
\]

where \( I(x, y) \) is the aerial image intensity at location \((x, y)\), \( k \) is the order of approximation, \( \lambda_i \) is the \( i \)th eigenvalue from TCCs’ SVD decomposition, \( \phi_i(x, y) \) is the \( i \)th eigenvector of TCCs’ SVD decomposition and \( f(x, y) \) represents the transparency of the mask at \((x, y)\) (0 means opaque and 1 means transparent). \( \otimes \) deNotes 2D convolution and \(|.|\) indicates the modulus of a complex number. Normally for each illumination process, the optical systems are kept constant, hence the convolution kernel \( \phi_i \) and the weight factor \( \lambda_i \) are constant and prepared in advance. Once the optical models are solidly built, the remain variable for the aerial image simulation is the mask \( f(x, y) \), which is a list of rectangles representing the transparent areas. Therefore, the aerial image simulation equation can be written as:

\[
I(x, y) = \sum_{i=1}^{k} \lambda_i \left| \sum_{m=1}^{n} (\phi_i \otimes r_m(x, y)) \right|^2
\]

where \( n \) is the number of rectangles in the input mask, \( r \) indicates the rectangle. So the simulated aerial image can be calculated by \( k \cdot n \) times 2D convolutions between the fixed \( \phi_i \) and input mask \( f(x, y) \), and \( k \) times multiplications between the fixed \( \lambda_i \) and the corresponding summation of 2D convolutions.

Due to 2D convolution is a rather expensive computing task, it is the key optimization object in the aerial image simulation. Two categories methods have been explored to improve the efficiency: the polygon-based approach [5] and the FFT-based approach [7]. Compared with the polygon-based approach, the FFT-based approach requires more memory space for the aerial image simulation. The idea of this method originated in that the mask can be decomposed into many basic rectangles. If pre-computing and storing the convolution of certain basic rectangles into a lookup table, the impact of any rectangle on a pixel could be calculated by only four times table lookups, as illustrated in Fig. 2.

Firstly, a lookup table \( T \) of the same size as the convolution kernel is constructed. The value at location \((x, y)\) in \( T \) is the impact of a rectangle \( R((0, 0) - (x, y)) \), where \((0, 0)\) is the left-bottom corner and \((x, y)\) is the right-top corner on the central pixel. Since the convolution operation is linear, the impact of any rectangle \( R((x_1, y_1) - (x_2, y_2)) \) on the center pixel can be obtained by an addition and two subtractions:

\[
\text{impact} = T(x_2, y_2) - T(x_1, y_2) - T(x_2, y_1) + T(x_1, y_1)
\]

Fig. 2. Rectangle decomposition. The impact of \( R((x_1, y_1) - (x_2, y_2)) \) on the central pixel can be calculated by looking up the impacts of the four shaped rectangles together.

Please cite this article in press as: Zhang F et al. Accelerating aerial image simulation using improved CPU/GPU collaborative computing. Comput Electr Eng (2015), http://dx.doi.org/10.1016/j.compeleceng.2015.05.018
In order to compute the impact of a rectangle on any pixel, we need to compute the coordinates of the four corners of the rectangle relative to the lookup table, as if the pixel were located at the center of the lookup table. Then, through four table lookups and three floating point additions/subtractions, the impact on the pixel can be obtained. This approach substantially reduces the runtime by avoiding the repetitive computation of convolutions. Details of this approach can be found in [29].

Algorithm 1. Polygon-based simulation: CPU serial version

\begin{algorithm}
\caption{Polygon-based simulation: CPU serial version}
\begin{algorithmic}[1]
\State \textbf{Input:} $2k$ lookup tables $T_i$ with size $W_T \times W_T$, $2k$ weights $\lambda_i$ and $n$ rectangles $R_m$, represented by the coordinates of its top, bottom, left and right boundaries.
\State \textbf{Output:} Aerial image: an 2D array $I[x][y]$ with size $W_i \times H_i$
\State \For {each $i \in [0, 2k-1]$}
\State \For {each $m \in [0, n-1]$}
\State \For {each $y \in [0, H_i - 1]$}
\State \For {each $x \in [0, W_i - 1]$}
\State \quad $x_{\text{orig}} = x - W_T / 2$
\State \quad $y_{\text{orig}} = y - W_T / 2$
\State \quad $R_{\text{rel}} \leftarrow \text{compute relative position of } R_m \text{ to } (x_{\text{orig}}, y_{\text{orig}})$
\State \quad $R_t \leftarrow \text{trunc}\_R_{\text{rel}} \text{ at four sides of the lookup table}$
\State \quad $I_{\text{part}}[i][y][x] = T_i[R_t \_right][R_t \_top] - T_i[R_t \_left][R_t \_top] - T_i[R_t \_right][R_t \_bottom] + T_i[R_t \_left][R_t \_bottom]$
\State \EndFor
\State \EndFor
\State \EndFor
\State \EndFor
\end{algorithmic}
\end{algorithm}

Otherwise, there are two detailed processes should be explained. One is truncation processing at the boundary of the lookup table. The reason is that the part outside the lookup table would also sit outside the convolution kernel, which means that this part will not contribute to the light intensity at the pixel. The other thing is that the convolution kernel is complex and the modulus operation in Eq. (1) is nonlinear. Therefore, the polygon-based approach can only be applied to compute the impact of the real or imaginary component of one convolution kernel separately. In order to compute the total impact of $k$ convolution kernels, we need $2k$ lookup tables, including $k$ corresponding real tables and $k$ corresponding imaginary tables. Algorithm 1 shows the serial algorithm of the polygon-based approach. It serves as a base of our study.

2.2. Implementation method on many-core GPU

Since most of the GPU chip’s surface is placed with arithmetic logic unit (ALU) and its in-chip memory, it provides a powerful parallel computing capabilities. It has been successful introduced into many research areas, and achieves acceleration of a dozen to several hundred times. So we try to accelerate the CPU serial version of aerial image simulation by GPU, and finally get $50 \times$ to $60 \times$ speedup [8].

The first step of parallelization design is to determine the suitable parallel task granularity, so does the GPU parallel. From the previous description of CPU serial method in Algorithm 1, the image simulation code includes triple loops, respectively are lookup tables, rectangles and image pixels. Therefore, the parallel task can be designed as the partial image simulation by some rectangles inside one lookup table, or the partial image simulation by one rectangle and one image pixel simulation by corresponding contributed rectangles. Compared with the CPU parallel, the GPU parallel wins in the number of parallel threads. However, in terms of computing power on a single thread, it is weaker than CPU thread, and is suitable for fine-grained parallelism. So the GPU-based parallelization design has two directions: rectangle-based and pixel-based parallel simulation, as shown in Fig. 3. The former method takes the impact of one rectangle on all image pixels as one basic parallel computing task. Thereby, different rectangles may contribute to the light intensity at the same pixel $(x, y)$. Due to parallel working, this issue will lead to serious memory access conflicts. The access conflicts of parallel computing will lead to miscalculation or an incorrect cumulative result. To avoid such problems, the straightforward solution is thread synchronization lock, such as atomic operation of compute unified device architecture (CUDA). The essence of atomic operation is to ensure a single-threaded access to resources, while to leave the other threads in a waiting state. So, the parallel computing efficiency is reduced around several times. The latter method is another solution to memory access conflict. It takes the impact of rectangles on a image pixel as basic parallel computing task. The number of pixels is equal to the number of
threads in one GPU. The experimental results also prove that the pixel-based method is superior to the rectangle-based method. So we mainly introduce the method here.

The pixel per thread (PPT) approach can be described as follows:

1. Each block computes the impact of all the rectangles on all the pixels in one tile, which is the partition unit for matching the threads number in one thread block.
2. Each thread goes through all the rectangles and computes their impact on one pixel in the tile.

The algorithm of the PPT approach is shown in Algorithm 2. Note that we have applied several optimizations. The first optimization is that we moved the iteration of $k$ to the most inner loop so that the computed rectangles’ location relative to the pixel can be shared among all lookup tables. Furthermore, we do not need two GPU kernel calls because when the computation of partial images is completed, the same thread can be used to combine the partial images into one. We do not need any thread synchronization before combining the image.

**Algorithm 2.** Pixel per thread approach: GPU version

```plaintext
1: compute the pixel location $(x, y)$ that current thread is responsible for
2: while there is unprocessed rectangles do
3:    load one rectangle $R_m$ to shared memory
4:    for each rectangle in shared memory do
5:        if rectangle does not impact pixel $(x, y)$ then
6:            skip the rectangle
7:        else
8:            for each $i \in [0, 2k - 1]$ do
9:                accumulate the impact of the rectangle to $I_{part}(i|y|x)$
10:           end for
11:        end if
12:    end for
13: end while
14: for each $i \in [0, 2k - 1]$ do
15:    $I[y|x] = I[y|x] + \lambda i \cdot (I_{part}(i|y|x))^2$
16: end for
```

Fig. 3. GPU-based aerial image parallel simulation approaches.
Another optimization is to load the rectangles into the shared memory in parallel, as shown in line 3, where each thread loads one rectangle into the shared memory. Then each thread will go through all the loaded rectangles to compute their contributions. This improvement significantly cuts down the global memory access. With this technique, we only need $block_{\text{number}} \times rectangle_{\text{number}}$ reads from the global memory. However, if we do not load the rectangles into the shared memory and let each thread reads directly from the global memory instead, we need $block_{\text{number}} \times thread_{\text{number}}$ in global memory reads.

The third optimization is to skip some rectangles that do not impact the pixel. Before we go into the loop that goes through all the lookup tables and accumulate the impact, we check if the rectangle is actually located outside the lookup table. If it is, we can skip the iteration completely and save many computations. Since the lookup table is usually quite small compared to the whole layout, a majority of the rectangles will be skipped for each pixel.

The fourth optimization is to use the GPU memory hierarchy. Because the lookup table $T_i$ and weight $\omega_i$ are kept constant, we utilize the texture memory and constant memory to store them and get a significant runtime saving. The advantage of the PPT approach is apparent: it avoids the expensive atomic operation on global memory. However, we need to read each rectangle multiple times because we partition the layout into tiles and each tile needs to load all the rectangles. Overall, we believe the advantage exceeds the disadvantage because updating the impact on each pixel consumes the most significant portion of the runtime. Improving the speed of pixel updating by avoiding atomic operation yields a huge speedup. Our experiments also verify this point.

2.3. Implementation method on multi-core SIMD CPU

Compare to the widely application of GPU, the computing power of CPU is usually underestimated. Furthermore, the general CPU parallel application still stuck in the old way of multi-core through OpenMP and multi-CPU through MPI. In order to further exploit the CPU parallel capability, the optimal solution should be the single instruction multiple data computing model, which is implemented by the Streaming SIMD Extensions (SSE) instructions. The SSE instructions can perform four basic operations by only one instruction, that is, SSE gives us an extra $4\times$ speedup on the basis of multi-core acceleration. Our previous work prove that the lookup table vectorization method achieves $73\times$ speedup over the serial CPU version on a hex-core SIMD CPU [10].

In terms of the aforementioned three parallel task level, namely lookup tables, rectangles and pixels, all are suitable for CPU thread, which can compute more coarse granularity task compared with GPU thread. Due to the basis of SIMD parallel is the continuous memory access, the optimized directions are respectively the single instruction multiple lookup tables data and multiple pixels data. Just as the intermediate result expression $I_{[i][j][x]}$ in Algorithm 1, we can get continuous memory access in $x$ dimension (pixel), or $i$ dimension after rearranging the matrix (lookup table). According to the analysis, we present two multi-core SIMD CPU based parallel approaches: lookup tables vectorization and pixels vectorization, as shown in Fig. 4.

**Fig. 4.** Multi-core SIMD CPU-based aerial image parallel simulation approaches.
Firstly, the basic theorem of the two vectorization approaches will be introduced. In the serial algorithm, we always have the following operations:

\[ I[i][j][k] = I[i][j][k] + T[i][k][k] \quad \forall i = 0, \ldots, 2k - 1 \]  
(4)

where \( I[i] \) represents the image for the \( i \)th lookup table \( T_i \). The algorithmic operation can be either addition or subtraction. With the SSE instruction set, we can pack images \( i, i + 1, i + 2, \) and \( i + 3 \) at pixel \((x, y)\) to one register, and then pack lookup tables \( i, i + 1, i + 2, \) and \( i + 3 \) at location \((x', y')\) to the other register. Then we can perform one SSE addition/subtraction on these two registers, and write the final result back to the image \((i, i + 1, i + 2, \) and \(i + 3\)) at pixel \((x, y)\). Such a process can be done for \( i = 0, 4, 8, \ldots \), so the lookup tables are vectorized to boost the performance.

Secondly, we explain the pixels vectorization approach. Suppose the coordinate of the pixel \( a \) is \((x, y)\), the relative position of the rectangle for \( a \) is \( R((x_1, y_1) - (x_2, y_2)) \) in the \( i \)th lookup table. For its neighboring pixel \( b \) \((x + 1, y)\), the relative position of the rectangle changes to \( R((x_1 - 1, y_1) - (x_2 - 1, y_2)) \). With SSE instructions, we can pack the image pixels \((x, y), (x + 1, y), (x + 2, y), \) and \((x + 3, y)\) to one register, pack the \( i \)th lookup table at locations \((x_1', y_1'), (x_1' - 1, y_1'), (x_2' - 1, y_2'), \) \((x_2' - 3, y_2')\) to the other two registers, perform several SSE algorithmic operations on them, and then write the result back to the image at pixels \((x, y), (x + 1, y), (x + 2, y), \) and \((x + 3, y)\) to get the intermediate image result. It is the vectorization of continuous pixels and corresponding lookup table elements as seen in Fig. 4, which provide the parallel acceleration.

**Algorithm 3.** Lookup tables vectorization approach: multi-core SIMD CPU version.

```plaintext
1: for each \( i \in [0, 2k - 1], i + 4 \) do
2:     Read four tables once time and perform vectorization for SSE access
3:     for each rectangle \( R_m \) do
4:         Compute \( Z_1, Z_2, Z_3, Z_4 \) and its impact region \( G \)
5:         \( R_{rel} \) ← compute the relative positions of \( R_m \) to the start pixel in \( G \)
6:         \( R_t \) ← truncate \( R_{rel} \) at four sides of the belonging four sub-regions
7:     for each pixel \((x, y)\) inside \( G \) do
8:         Multi-core: each thread separately implements for one row of pixels
9:         SIMD: \( R^4 \) indicates the vector operations under SSE instructions
10:        if \((x, y)\) inside \( Z_1 \) then
11:            \( I_{part}[y][x][R^4] = T_p[R_t.bottom][R.t.left] - T_p[R_t.bottom][R.t.right] - T_p[R_t.top][R.t.left] + T_p[R_t.top][R.t.right] \)
12:        else if \((x, y)\) inside \( Z_2 \) then
13:            \( I_{part}[y][x][R^4] = -T_p[R_t.top][R.t.left] + T_p[R_t.top][R.t.right] \)
14:        else if \((x, y)\) inside \( Z_3 \) then
15:            \( I_{part}[y][x][R^4] = T_p[R_t.top][R.t.right] \)
16:        else
17:            \( I_{part}[y][x][R^4] = T_p[R_t.top][R.t.right] \)
18:     end if
19:     continuously move \( R_t \) four corner in row directions
20: end for
21: end for
22: for each pixel \((x, y)\) in the image do
23:     \( I[y][x] = I[y][x] + \sum_{i} T_{part}[y][x][R^4]^2 \)
24: end for
25: end for
```

The above introduction is only the basic parallel mechanism. In both methods, there are some detailed optimizations, which are exposed in our previous publication [10], such as computing decomposition and aligned memory access. Whether parallel efficiency or potential of task partition, the lookup table vectorization method in Algorithm 3 is the better solution, which has been proved by our previous work. So we try to discuss it more specifically as follows:

1. Computing decomposition
   The values obtained from the table lookups based on some rectangle corners relative to some pixels are always 0. Calculating the coordinates of those corners relative to the lookup table is redundant and should be avoided. For a rectangle, its impact region can be further decomposed into four small regions as shown in Fig. 5, pixels in Z1 will be impacted by the four corners of the rectangle, pixels in Z2 will be impacted by the right-bottom corner and the
right-top corner, pixels in Z3 will be impacted by the left-top corner and the right-top corner, and pixels in Z4 will be only impacted by the right-top corner. So, according to the region that the pixel is located in, only the corresponding corners have to be calculated.

(2) Multi-threading
The approach is to let one thread responsible for one row of pixels when updating the impact region. Since the updating of each row is independent, every thread is updating the same image but in different rows. So we do not need to worry about any synchronization issue. The final image summation can also be paralleled by the same philosophy that is each thread handles one row of the image.

3. Aerial image simulation based on improved CPU/GPU heterogeneous computing

With the increasingly development of high performance computing devices, heterogeneous and collaborative computing are the future trend in solving compute-intensive problems. Our previous work only focuses on one device once simulation, and actually need to be expanded for higher efficiency electronic design automation (EDA). The research on heterogeneous computing normally includes four aspects, i.e., heterogeneous parallel algorithm, heterogeneous collaborative mechanism, scalability and reliability. We mainly consider the former three aspects in the aerial image simulation, and try to improve the independent computing power for each devices, collaborative efficiency and the adaptability for different hardware architecture.

3.1. AVX based high performance vectorization in SIMD CPU parallel

The advanced vector extensions (AVX), supported by the new generation CPU of Intel and AMD in 2011 and later, expands the SIMD register from 128 bits to 256 bits. It means that one AVX instruction can operate 8 single precision floating point data. In the aerial image simulation, the number of lookup tables or pixels that processed by one instruction will increase from 4 elements to 8 elements. Compare with the previous SSE parallel, we can expect a $2^8$ speedup by applying AVX.

Basically, the main work by expanding SSE to AVX is the programming level, such as the replacement of data type and instruction application programming interface (API). However, there are still some algorithmic modifications should be considered to fit the higher vectorization, e.g., data padding and aligned memory access. In the AVX version of aerial image simulation, we make the modifications as follow:

1. Updating the compiler to the new version for supporting AVX instructions.
2. Processing 8 lookup tables or pixels by each AVX instruction. Accordingly, the required memory for data loading and aligned access processing will be doubled to each CPU thread.
3. Modifying the corresponding data type and instruction APIs.

Through the SIMD algorithm updating, some new issues are worth studying. First, the storage overhead and aligned access preprocess increase with the vectorization scale. Second, with the increase of vectorization scale, for example 128 bits in SSE, 256 bits in AVX and 512 bits in AVX2, the parallel granularity changes more coarse. For normal aerial image simulation, the number of lookup tables is only a dozen to tens, which are not enough to occupy the whole CPU power (64 lookup tables are required if computing on the platform of octa-core with AVX). If utilizing the pixels vectorization approach, the problem will be solved by fine-grained task. But the computing efficiency will be slow down greatly according to our previous experimental results. This is a tradeoff between hardware efficiency and algorithm efficiency, and should be studied in our future work.

3.2. Dynamic task scheduling algorithm

When talking about the dynamic task scheduling, the first key factor is task partition. Different parallel task granularity fits the different application, so does the aerial image simulation. The choice of task granularity that can balance efficiency,
accuracy, memory and complexity, is a challenging work, especially in the heterogeneous computing case. In the aerial image simulation, there are four task partition objects, respectively are the input, processing, output and their hybrid, as illustrated in Fig. 1.

(1) Input-oriented task partition

The input of aerial image simulation is the layout of mask, which consists of many decomposed rectangles. Each rectangle can contribute to the final image, and can be taken as the parallel task unit. The method can also be explained by the follow equation:

\[ I(x, y) = \sum_{p=1}^{q} \left\{ \sum_{i=1}^{k} \lambda_i \times \left( \sum_{m=1}^{p n/q} (\phi_i \odot r_m(x, y)) \right)^2 \right\} \]  

(5)

where \( p \) is the order of parallel task, \( q \) is the number of total simulation task. After the mask partition, each parallel task can simulate one partial intensity image. Then merging them together, the final aerial image will be acquired.

(2) Processing-oriented task partition

According to the decomposition of Hopkins partially coherent imaging equation, the imaging system is a set of linear system with convolution, square, scale and sum operations, and the mutual coupling of different linear systems are small. Therefore, we can take each linear system process as the task unit, namely one lookup table process, which will be dispatched to CPU and GPU, as illustrated in the follow equation:

\[ I(x, y) = \sum_{p=1}^{q} \left\{ \sum_{i=1}^{k} \lambda_i \times \left( \sum_{m=1}^{n} (\phi_i \odot r_m(x, y)) \right)^2 \right\} \]  

(6)

(3) Output-oriented task partition

The output-oriented task partition takes one row pixels in the final image as the parallel unit, and distribute to heterogeneous computing devices, as illustrated in the follow equation:

\[ I(x, y) = \sum_{p=1}^{q} \left\{ \sum_{i=1}^{k} \lambda_i \times \left( \sum_{m=1}^{n} (\phi_i \odot r_m(x, y)) \right)^2 \right\} \]  

(7)

where \( Y \) is the total number of rows of final image.

(4) Hybrid task partition

The hybrid method combines the former methods to complete a fine-grained task partition. The follow method takes part of rectangles and part of lookup tables as parallel task, and simulates the partial intensity image with several rectangles’ convolution, as illustrated in the follows:

\[ I(x, y) = \sum_{p=1}^{q_1} \sum_{p_1=1}^{p_1} \left\{ \sum_{i=1}^{k_1} \lambda_i \times \left( \sum_{m=1}^{n_1} (\phi_i \odot r_m(x, y)) \right)^2 \right\} \]  

(8)

where \( p_1 \) is the order of parallel task in the side of lookup table, \( p_2 \) is the order of parallel task in the side of rectangle, and \( q_1, q_2 \) are respectively the number of parallel task partition in the two sides.

The choice of parallel task partition should be connected with its application. In terms of aerial image simulation, it is the essential and frequently used module in the EDA design, and obviously not a big compute-intensive problem. Even a large layout image simulation will be computed in less than a few minutes. So the heterogeneous computing in single computer is enough to process the issue. The computing resources may be several CPUs and GPUs, which determine the task granularity should be medium. If granularity is coarse, the total task will not enough for partition. If granularity is fine, the powerful parallel computing capability of SIMD CPU and GPU will be wasted by a lower resource occupancy rate. From this sense, the latter two partition methods is not suitable here for their fine-grained partition. On the contrary, the second method are not suitable for its coarse-grained partition, which has been discussed in the AVX section. As for the input-oriented partition, each computing terminals are distributed with different number of input rectangles for simulation, which is a medium-grained task. Therefore, it is used for CPU/GPU collaborative computing in single computer. Notice that the GPU and CPU can still use the lookup tables vectorization method, pixel per thread method, respectively, despite of the simulation task is partitioned by rectangle number or lookup tables.

Based on the rectangle-oriented task partition, we propose the straightforward dynamic scheduling approach, as shown in Fig. 6. The initial state is the initial approximate task distribution, e.g. \( a \) and \( b \) indicate the number of rectangles for GPU and CPU. The strategy tends to distribute more task to GPU, not for its outstanding computing power but its flexible and cheap cost of threads execution. So GPU is almost in finish state when CPU completes all its job, and this situation can make the next redistribution more averaged.
The hungry state means that one computing terminal completes all its job and want to get more tasks. The steal state means that the completed terminal steals some tasks from the current working terminal, namely, the redistribution of current unfinished tasks. Due to the number of simulated rectangles are clear for each computing terminals, this realtime simulation velocity can be used for the second distribution. In Fig. 6, the current task of CPU and GPU in first hungry state are respectively 0 and c. So the realtime simulation ratio of GPU to CPU updates to \( \frac{a}{c} \) from \( \frac{a}{b} \). Then the realtime simulation velocity of GPU and CPU are \( \frac{a-c}{a-c+b} \) for GPU and \( bc/(a-c+b) \) for CPU in the steal state. After several two state transitions, the simulation is in finish state while the current extra task c is zero.

3.3. Heterogeneous scalability

Due to the independence and medium-grained of parallel task, the collaborative computing is easy to expand to multi-GPUs computing. Compared with one CPU and one GPU version, the extra work will include three parts. First, extra CPU threads should be allocated for the new added GPU, and in charge of data transfer, launching the new GPU kernels and task scheduling. Second, the steal operation is only performed between the finished terminal and the most spare task terminal. This strategy can keep other computing terminal working more efficient. Third, extra partial intensity images will be collected from GPU memory and merged in CPU memory, which will increase the simulation time. In the experiment section, CPU/multi-GPUs collaborative computing will be deployed for quick simulation.

4. Experimental results and analysis

In this section, we compare the three approaches, namely GPU-based, Multi-core SIMD CPU-based and their collaborative methods, introduced in the previous section through experiments. The experiments are performed on a machine with a hex-core 2.6 GHz Intel Xeon processor, 16 GB of memory, and a NVIDIA Tesla K10 GPU. As for the input data, we use the data set presented in [8]. There are a total of 26 lookup tables, each of which has size 257 x 257. Three data sets are used in our experiment: a small set, a medium set, and a large set. Each set contains four data, and the details of each data such as the number of rectangles and the number of pixels in the image, are shown in Table 1.

To compare the collaborative computing and previous GPU or CPU computing, we implement 6 methods, each of which has its own configuration. The comparison of runtime and speedup for these methods are shown in Tables 2 and 3, where 1 GPU and 2 GPU are the one GPU and two GPU parallel version, M-SSE is the multi-core SSE version, M-AVX is the multi-core AVX version, the last two columns are the corresponding collaborative computing version. Note that the methods all use single-precision floating points. The maximum absolute error among the pixels of the image in these methods are measured as well. All of these methods has the maximum absolute error on the order of \( 10^{-6} \). The aerial image simulation input and result in small data set are shown in Figs. 7 and 8, respectively.

Firstly, we discuss the acceleration of AVX method. The speedup of AVX to SSE is less than 2 in majority cases. It explains the task granularity issue that we discuss at the AVX method section. Due to coarse-grained task, the lookup tables vectorization is the fast method for SSE acceleration, but not for AVX acceleration. The AVX method needs more padding lookup tables to meet aligned memory access. Thus the acceleration is slowed down. Because the pixels vectorization efficiency is
slower than lookup tables vectorization, no better result is expected if we select pixels vectorization for AVX acceleration. In a word, the AVX can achieve acceleration over SSE implementation, and is better applied in the big scale problems.

Secondly, the CPU/GPU collaborative computing method shows a better speedup as we expect. Due to the independent execution among GPUs and CPU threads, the collaborative computing and their independent computing are almost the linear accumulative relationship, which also illustrates that our dynamic scheduling algorithm is simple and effective, achieves high collaborative efficiency with the least communication and control overhead. The speedup comparison of different methods on large scale layout are shown in Fig. 9. In large scale layout case, the speedup reaches more than 100, which definitely accelerates the process of IC design.

In addition to the image simulation, that is the parallel part, the other serial parts are reserved in the final collaborative computing version. To make the parallel optimization more clearly, the comparison of whole computing overheads are shown in Table 4. The whole aerial image simulation process is summarized as three operations, including input/output (IO) operation, pre-process operation and image simulation. The IO operation indicates the input of necessary data files, such as

Table 1
Data set for our experiments.

<table>
<thead>
<tr>
<th>Data</th>
<th>Rectangles number</th>
<th>Layout size (nm)</th>
<th>Pixels number</th>
</tr>
</thead>
<tbody>
<tr>
<td>small-1</td>
<td>209</td>
<td>1850 x 1100</td>
<td>463 x 275</td>
</tr>
<tr>
<td>small-2</td>
<td>212</td>
<td>1750 x 1100</td>
<td>438 x 275</td>
</tr>
<tr>
<td>small-3</td>
<td>217</td>
<td>1650 x 1100</td>
<td>413 x 275</td>
</tr>
<tr>
<td>small-4</td>
<td>225</td>
<td>1850 x 1150</td>
<td>463 x 275</td>
</tr>
<tr>
<td>medium-1</td>
<td>1100</td>
<td>4100 x 3100</td>
<td>1025 x 775</td>
</tr>
<tr>
<td>medium-2</td>
<td>1297</td>
<td>4220 x 3100</td>
<td>1025 x 775</td>
</tr>
<tr>
<td>medium-3</td>
<td>1214</td>
<td>4100 x 3100</td>
<td>1025 x 775</td>
</tr>
<tr>
<td>medium-4</td>
<td>1292</td>
<td>4100 x 3100</td>
<td>1025 x 775</td>
</tr>
<tr>
<td>large-1</td>
<td>4904</td>
<td>8100 x 6100</td>
<td>2025 x 1525</td>
</tr>
<tr>
<td>large-2</td>
<td>4897</td>
<td>8320 x 6100</td>
<td>2080 x 1525</td>
</tr>
<tr>
<td>large-3</td>
<td>4892</td>
<td>8100 x 6100</td>
<td>2025 x 1525</td>
</tr>
<tr>
<td>large-4</td>
<td>4814</td>
<td>8100 x 6100</td>
<td>2025 x 1525</td>
</tr>
</tbody>
</table>

Table 2
Runtime comparison of image simulation (unit is second).

<table>
<thead>
<tr>
<th>Data</th>
<th>1 GPU</th>
<th>2 GPU</th>
<th>M-SSE</th>
<th>M-AVX</th>
<th>M-AVX + 1 GPU</th>
<th>M-AVX + 2 GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>small-1</td>
<td>0.07</td>
<td>0.03</td>
<td>0.07</td>
<td>0.03</td>
<td>0.02</td>
<td>0.02</td>
</tr>
<tr>
<td>small-2</td>
<td>0.07</td>
<td>0.04</td>
<td>0.07</td>
<td>0.04</td>
<td>0.03</td>
<td>0.02</td>
</tr>
<tr>
<td>small-3</td>
<td>0.08</td>
<td>0.04</td>
<td>0.07</td>
<td>0.04</td>
<td>0.03</td>
<td>0.02</td>
</tr>
<tr>
<td>small-4</td>
<td>0.08</td>
<td>0.04</td>
<td>0.08</td>
<td>0.04</td>
<td>0.03</td>
<td>0.03</td>
</tr>
<tr>
<td>medium-1</td>
<td>0.40</td>
<td>0.19</td>
<td>0.47</td>
<td>0.27</td>
<td>0.20</td>
<td>0.15</td>
</tr>
<tr>
<td>medium-2</td>
<td>0.49</td>
<td>0.22</td>
<td>0.51</td>
<td>0.29</td>
<td>0.19</td>
<td>0.16</td>
</tr>
<tr>
<td>medium-3</td>
<td>0.45</td>
<td>0.21</td>
<td>0.49</td>
<td>0.27</td>
<td>0.19</td>
<td>0.14</td>
</tr>
<tr>
<td>medium-4</td>
<td>0.46</td>
<td>0.22</td>
<td>0.60</td>
<td>0.28</td>
<td>0.22</td>
<td>0.13</td>
</tr>
<tr>
<td>large-1</td>
<td>4.12</td>
<td>2.01</td>
<td>2.34</td>
<td>1.62</td>
<td>1.22</td>
<td>0.92</td>
</tr>
<tr>
<td>large-2</td>
<td>4.53</td>
<td>2.23</td>
<td>2.44</td>
<td>1.70</td>
<td>1.30</td>
<td>0.95</td>
</tr>
<tr>
<td>large-3</td>
<td>4.49</td>
<td>2.20</td>
<td>2.41</td>
<td>1.68</td>
<td>1.28</td>
<td>0.95</td>
</tr>
<tr>
<td>large-4</td>
<td>4.41</td>
<td>2.18</td>
<td>2.38</td>
<td>1.66</td>
<td>1.26</td>
<td>0.93</td>
</tr>
</tbody>
</table>

Table 3
Speedup comparison of image simulation.

<table>
<thead>
<tr>
<th>Data</th>
<th>1 GPU</th>
<th>2 GPU</th>
<th>M-SSE</th>
<th>M-AVX</th>
<th>M-AVX + 1 GPU</th>
<th>M-AVX + 2 GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>small-1</td>
<td>46</td>
<td>107</td>
<td>46</td>
<td>107</td>
<td>161</td>
<td>161</td>
</tr>
<tr>
<td>small-2</td>
<td>46</td>
<td>80</td>
<td>46</td>
<td>80</td>
<td>107</td>
<td>161</td>
</tr>
<tr>
<td>small-3</td>
<td>40</td>
<td>80</td>
<td>40</td>
<td>80</td>
<td>107</td>
<td>161</td>
</tr>
<tr>
<td>small-4</td>
<td>40</td>
<td>80</td>
<td>40</td>
<td>80</td>
<td>107</td>
<td>161</td>
</tr>
<tr>
<td>medium-1</td>
<td>26</td>
<td>60</td>
<td>28</td>
<td>48</td>
<td>65</td>
<td>87</td>
</tr>
<tr>
<td>medium-2</td>
<td>27</td>
<td>59</td>
<td>25</td>
<td>46</td>
<td>67</td>
<td>81</td>
</tr>
<tr>
<td>medium-3</td>
<td>29</td>
<td>58</td>
<td>27</td>
<td>48</td>
<td>67</td>
<td>93</td>
</tr>
<tr>
<td>medium-4</td>
<td>28</td>
<td>59</td>
<td>22</td>
<td>47</td>
<td>60</td>
<td>98</td>
</tr>
<tr>
<td>large-1</td>
<td>26</td>
<td>53</td>
<td>45</td>
<td>65</td>
<td>86</td>
<td>115</td>
</tr>
<tr>
<td>large-2</td>
<td>25</td>
<td>47</td>
<td>43</td>
<td>62</td>
<td>81</td>
<td>111</td>
</tr>
<tr>
<td>large-3</td>
<td>25</td>
<td>47</td>
<td>43</td>
<td>62</td>
<td>83</td>
<td>111</td>
</tr>
<tr>
<td>large-4</td>
<td>25</td>
<td>48</td>
<td>44</td>
<td>63</td>
<td>84</td>
<td>113</td>
</tr>
</tbody>
</table>
as layout data and lookup tables, and the output of the simulated aerial image. The pre-process operation indicates the rest serial operations like memory allocations. From the overheads comparison, it can be seen that the proportion of the serial part, i.e., IO and pre-process operations, is significantly increased from 1.2% to 57% with the collaborative computing optimizations. Therefore, the serial part lowers the whole computing efficiency. Although the serial part is difficult to be optimized, the serial execution can be hidden in batch image simulation. For example, the input of next image simulation is

![Fig. 7. The input layout of small-1.](image)

![Fig. 8. The output aerial image of small-1.](image)

![Fig. 9. The speedup comparison of aerial image simulation on large data sets.](image)
executed at the same time of current image simulation. Through the concurrent execution of serial and parallel parts in batch image simulation, the efficiency is expected to further improve.

Finally, we can see that AVX and CPU/GPU collaborative computing are all effective method to boost the aerial image simulation on single computer. The advantage of collaborative computing is obviously and should be expanded to other EDA algorithm to shorten the design process.

5. Conclusion and future work

We have investigated the aerial image parallel simulation based on the SIMD CPU/GPU collaborative computing. The AVX instructions are introduced to the SIMD CPU parallel simulation for higher vector extension. We analyze the task partition approaches and design the rectangle-based dynamic task scheduling strategy in the aerial image collaborative computing system, which consists of multi-CPUs and multi-GPUs. The simulation experiments confirm the advantages of the method. With the increase of vector extension, CPU is more powerful than GPU in this algorithm. The load of the collaborative computing system is dynamically balanced using the proposed strategy. The method achieves a speedup up to 160× over the baseline serial approach and 4× over the single GPU parallel approach. Therefore, this kind of collaborative computing method can fully exploit the computing power of single computer, and can be used to other EDA algorithms. We plan for the near future to extend the approach to cloud computing for extensive application.

Acknowledgements

This work was supported by the Beijing Higher Education Young Elite Teacher Project under Grant YETP0500, the Fundamental Research Funds for the Central Universities under Grant YS 1404, and the Interdisciplinary Research Project in Beijing University of Chemical Technology.

References


Table 4

The whole computing overheads comparison (unit is second).

<table>
<thead>
<tr>
<th>Operation</th>
<th>CPU</th>
<th>1 GPU</th>
<th>2 GPU</th>
<th>M-SSE</th>
<th>M-AVX</th>
<th>M-AVX + 1 GPU</th>
<th>M-AVX + 2 GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input/output</td>
<td>1.23</td>
<td>1.19</td>
<td>1.22</td>
<td>1.21</td>
<td>1.12</td>
<td>1.15</td>
<td>1.23</td>
</tr>
<tr>
<td>Pre-process</td>
<td>0.02</td>
<td>0.01</td>
<td>0.02</td>
<td>0.02</td>
<td>0.01</td>
<td>0.02</td>
<td>0.01</td>
</tr>
<tr>
<td>Image simulation</td>
<td>108.22</td>
<td>4.12</td>
<td>2.01</td>
<td>2.34</td>
<td>1.62</td>
<td>1.22</td>
<td>0.92</td>
</tr>
<tr>
<td>Total</td>
<td>109.47</td>
<td>5.32</td>
<td>3.24</td>
<td>3.56</td>
<td>2.75</td>
<td>2.39</td>
<td>2.16</td>
</tr>
</tbody>
</table>

Please cite this article in press as: Zhang F et al. Accelerating aerial image simulation using improved CPU/GPU collaborative computing. Comput Electr Eng (2015), http://dx.doi.org/10.1016/j.compeleceng.2015.05.018.
Fan Zhang received the Ph.D. degree in signal and information processing from Institute of Electronics, Chinese Academy of Science, Beijing, China, in 2008. He is currently an associate professor of electronic and information engineering at the Beijing University of Chemical Technology. His current research interests include signal processing, high performance computing and scientific visualization.

Chen Hu received the B.E. degree in electronic and information engineering from Beijing University of Chemical Technology, Beijing, China, in 2013. He is currently pursuing his M.S. degree in the field of high performance computing at Beijing University of Chemical Technology.

Pei-Ci Wu received the B.S. degree and M.S. degree in Computer Science from National Tsing Hua University Taiwan, in 2005 and 2007, respectively, the Ph.D. degree in Electrical and Computer Engineering from University of Illinois at Urbana-Champaign, in 2015. Currently, she works as Research and Development Engineer in Mentor Graphics Inc., USA.

Hongbo Zhang received the B.S. degree in electronic and information engineering from Zhejiang University (ZJU), China in 2005, the M.Eng. degree in electrical engineering from ZJU in 2007, and the Ph.D. degree in Electrical and Computer Engineering from University of Illinois at Urbana-Champaign, in 2012. Since 2012, he has been working as Research and Development Engineer in Synopsys Inc., USA.

Martin D.F. Wong received the B.S. degree from the University of Toronto, Toronto, ON, Canada, and the M.S. degree from the University of Illinois at Urbana-Champaign (UIUC), Urbana, both in mathematics, and the Ph.D. degree in computer science from UIUC in January 1987. He is currently a professor of electrical and computer engineering at UIUC.