ABSTRACT

ACCELERATING DATA-INTENSIVE SCIENTIFIC VISUALIZATION AND COMPUTING THROUGH PARALLELIZATION

by
Dongliang Chu

Many extreme-scale scientific applications generate colossal amounts of data that require an increasing number of processors for parallel processing. The research in this dissertation is focused on optimizing the performance of data-intensive parallel scientific visualization and computing.

In parallel scientific visualization, there exist three well-known parallel architectures, i.e., sort-first/middle/last. The research in this dissertation studies the composition stage of the sort-last architecture for scientific visualization and proposes a generalized method, namely, Grouping More and Pairing Less (GMPL), for order-independent image composition workflow scheduling in sort-last parallel rendering. The technical merits of GMPL are two-fold: i) it takes a prime factorization-based approach for processor grouping, which not only obviates the common restriction in existing methods on the total number of processors to fully utilize computing resources, but also breaks down processors to the lowest level with a minimum number of peers in each group to achieve high concurrency and save communication cost; ii) within each group, it employs an improved direct send method to narrow down each processor's pairing scope to further reduce communication overhead and increase composition efficiency. The performance superiority of GMPL over existing methods is evaluated through rigorous theoretical analysis and further verified by extensive experimental results on a high-performance visualization cluster.

The research in this dissertation also parallelizes the *over* operator, which is commonly used for α-blending in various visualization techniques. Compared with its predecessor, the fully generalized *over* operator is n-operator compatible.

To demonstrate the advantages of the proposed operator, the proposed operator is applied to the asynchronous and order-dependent image composition problem in parallel visualization.

In addition, the dissertation research also proposes a very-high-speed pipeline-based architecture for parallel sort-last visualization of big data by developing and integrating three component techniques: i) a fully parallelized per-ray integration method that significantly reduces the number of iterations required for image rendering; ii) a real-time *over* operator that not only eliminates the restriction of pre-sorting and order-dependency, but also facilitates a high degree of parallelization for image composition.

In parallel scientific computing, the research goal is to optimize QR decomposition, which is one primary algebraic decomposition procedure and plays an important role in scientific computing. QR decomposition produces orthogonal bases, i.e.,"core" bases for a given matrix, and oftentimes can be leveraged to build a complete solution to many fundamental scientific computing problems including Least Squares Problem, Linear Equations Problem, Eigenvalue Problem. A new matrix decomposition method is proposed to improve time efficiency of parallel computing and provide a rigorous proof of its numerical stability.

The proposed solutions demonstrate significant performance improvement over existing methods for data-intensive parallel scientific visualization and computing. Considering the ever-increasing data volume in various science domains, the research in this dissertation have a great impact on the success of next-generation large-scale scientific applications.

ACCELERATING DATA-INTENSIVE SCIENTIFIC VISUALIZATION AND COMPUTING THROUGH PARALLELIZATION

by
Dongliang Chu

A Dissertation
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy in Computer Sciences

Department of Computer Science

August 2016

APPROVAL PAGE

ACCELERATING DATA-INTENSIVE SCIENTIFIC VISUALIZATION AND COMPUTING THROUGH PARALLELIZATION

Dongliang Chu

Dr. Chase Wu, Dissertation Advisor — Date
Associate Professor of Computer Science, New Jersey Institute of Technology

---

Dr. Cristian M. Borcea, Committee Member — Date
Associate Professor of Computer Science, New Jersey Institute of Technology

---

Dr. Alexandros Gerbessiotis, Committee Member — Date
Associate Professor of Computer Science, New Jersey Institute of Technology

---

Dr. Xiaoning Ding, Committee Member — Date
Assistant Professor of Computer Science, New Jersey Institute of Technology

---

Dr. Yi Chen, Committee Member — Date
Associate Professor of School of Management, New Jersey Institute of Technology

BIOGRAPHICAL SKETCH

Author:	Dongliang Chu
Degree:	Doctor of Philosophy
Date:	August 2016

Undergraduate and Graduate Education:

- Doctor of Philosophy in Computer Science
  New Jersey Institute of Technology, Newark, NJ, 2016
- Master of Science in Computer Engineering
  Zhengzhou University, Zhengzhou, Henan, 2012
- Bachelor of Science in Mathematics
  North China University of Water Resources and Electric Power (NCWU), Zhengzhou, Henan, 2009

Major: Computer Science

Presentations and Publications:

D. Chu, C. Q. Wu, J. Gao, L. Wang. On a Generalized Scheme for Order-Independent Image Composition in Parallel Visualization. International Performance Computing and Communications Conference 2013.

D. Chu, C. Q. Wu, Z. Wang, Y. Wang. A Fully Generalized *over* Operator with Applications to Image Composition in Parallel Visualization for Big Data Science. International Conference on Parallel and Distributed Systems 2014.

D. Chu, C. Q. Wu.On a Pipeline-based Architecture for Parallel Volume Visualization of Big Data. International Conference for Parallel Processing 2016.

D. Chu, C. Q. Wu. The Order-dependence eliminated and efficiency optimized parallel *over* Operator. In preparation for Journal of Parallel and Distributed Computing.

D. Chu, C. Q. Wu. V4BD: A Very High-speed Value-added Volume Visualization Architecture. In preparation for Transactions on Visualization and Computer Graphics.

Dedicated to my beloved parents:
(谨此献给我挚爱的双亲:)
Faying Chu(褚发营)，father(父亲)
Lamei Shen(沈腊梅)，mother(母亲)

“No pain, No gain”
“梅花香自苦寒来”

ACKNOWLEDGMENT

Longing and longing, waiting and waiting, finally it comes to the conclusion of my Ph.D. student life, which is definitely impossible without appearances of so many amazing features in my life.

The top-ranked and valued-most character is my advisor, Dr. Chase Q. Wu. There is so much appreciation I want to convey such that I can't figure out which is the most suitable for expressing here. I will just enumerate a few that are most fresh in my mind currently. Honestly speaking, without Dr. Wu, I would have had no opportunity to conduct or start Ph.D. study in U.S.. He guided my entire process of application, first entry into the U.S., settlement at a totally strange location, etc. As I soundly got into the role of a Ph.D. student under his advisement, it constantly made me feel lucky and grateful for the precious opportunity that I was granted. He always patiently helped me formulate the research problem, discussed the possible solutions and polished the according draft again and again. I was improved and benefitted from this significantly. Of course, there are other forms of help that I want to share. However, I have to stop here and keep all of them in my heart forever.

Committee members of my Ph.D. dissertation instructed and benefited my Ph.D. a lot as well. Each of them devoted significant amounts of time to my dissertation, provided huge amounts of valued comments, and always kept me on the right track. I want to express sincere appreciation to Dr. Cristian M. Borcea, Dr. Alexandros Gerbessiotis, Dr. Xiaoning Ding, and Dr. Yi Chen.

I also want to express hearted thanks to the considerable help that I received from the department of Graduate Studies. Special thanks to Dr. Sotirios G. Ziavras, Ms. Clarisa González-Lenahan, Ms. Lillian Quiles and Mr. David Tress, Dr. George Olsen, Dr. Ali Mili, Ms. Angel Butler, .

I want to thank all of friends for the helpful hands they lent to me.

TABLE OF CONTENTS

TABLE OF CONTENTS
(Continued)

Chapter **Page**

TABLE OF CONTENTS
(Continued)

LIST OF TABLES

LIST OF FIGURES

LIST OF FIGURES
(Continued)

LIST OF FIGURES
(Continued)

Figure **Page**

CHAPTER 1

INTRODUCTION

The scale of data generated by modern scientific applications is rapidly increasing, ranging from terabyes at present to petabytes or exabytes down the road in the predictable future. Such data, now commonly termed as “big data”, can be resulted from various sources including simulations, experiments, or observations, and must be processed and analyzed in a timely manner for scientific exploration and knowledge discovery. Among many existing methods for data processing/analytics, scientific visualization and computing have been well recognized and widely used in broad science communities to make sense of the data. However, the sheer volume of today’s scientific data has posed a daunting challenge on traditional visualization and computing technologies. Parallel scientific visualization and computing have proven to be promising solutions to handle data of such scales.

1.1 Parallel Scientific Visualization and Image Composition

Many parallel visualization architectures have emerged in the last decade [48, 49, 30, 43, 1, 60]. As put forth by Molnar *et al.*, according to the time when primitives in the raw data are sorted on their corresponding processors, parallel visualization architectures can be classified into three categories, i.e., sort-first, sort-middle, and sort-last [47]. Although the instances in each category may differ in their particular implementations, they generally share some common features. In sort-first, each primitive in the raw data is assigned to a certain processor, which is responsible for a predivided region of the display screen. This scheme could save network communications between the processors due to the predetermined locality of each primitive, but may result in unbalanced workload among the processors due to the static screen partitioning. In sort-last, s are not assigned as strictly. Instead, they are rendered into pixels by the processors they are initially assigned to, and these distributed pixels must be composited into a final image. This scheme may achieve a high level of workload balance at the cost of increased network communications. In sort-middle, the entire visualization process is naturally divided into two distinctive stages: geometry processing and rasterization. Such a partitioning of visualization groups together the operations with similar purposes and distinguishes from others. This scheme may consume more bandwidth and incur a longer processing time, and hence is investigated in more theoretical than practical settings.

Among the above three well-known parallel architectures, sort-last is often preferred in many applications due to its adaptability to load balancing. In general, the sort-last architecture comprises of two stages: rendering and composition [67]. We mainly focus on composition stage here, which encompasses huge number of research topics that are of practical importance while still not satisfactorily resolved in times of "Big Data", including the composition operator, the communication overheads among participating computing units and so on.

1.2 Parallel Scientific Computing and QR Decomposition

QR decomposition is a long existing and widely used algebraic decomposition procedure in scientific computing. Given a non-singular matrix $A_{n\times n}$, its QR decomposition is defined as

$$A = Q \cdot R,$$

where Q is an orthogonal matrix, i.e., $Q^T \cdot Q = I$, and R is an upper-triangular matrix. Such decomposition reveals the contained orthogonality among vectors within the input matrix and maps the given matrix to orthogonal bases, hence simplifying the representation of the given matrix and facilitating some of its computations with others. In addition, such decomposition's role of importance to literature is also established by providing competitive solutions to a large number of basic scientific computing problems including Least Squares Problem [35, 7], Linear Equations Problem [21], Eigenvalue Problem [31], and Numerical Optimization Problem [72].

During its continuous development in the past two centuries, there emerges tons of potential solutions, achieving the same purpose through distinct methods. Similar to the visualization problem, these methods can also be roughly distinguished as parallel or sequential, each of which calls for suitable evaluation criteria. In the sequential category, we consider stability [29, 25, 6, 65, 64, 33], i.e., deviation between the calculated and expected results, and time efficiency [15, 52], i.e., number of floating-point operations to execute on a dedicated computing unit (flops). In the parallel category, the criterion of stability remains the same as in the sequential counterpart, but the criterion of time efficiency differs. It considers the number of flops on the critical path of parallel execution and the communication overhead between multiple parallel computing units.

Considering the practical facts of “Big Data” and continuously increasing data size, we focus our research on parallel QR decomposition. In the literature of parallel QR decomposition, both stability and time efficiency issues are still of concern, as time-efficient methods need more stability while stable methods need higher time efficiency. Our research goal here is to fully parallelize the decomposition procedure so as to minimize the number of flops on the critical path meanwhile maintaining a high level of accuracy.

CHAPTER 2

IMAGE COMPOSITION OPERATOR

As one key step in a sort-last parallel rendering system, image composition has received a great deal of attention from many researchers. Existing efforts mainly focus on the following two aspects of research: i) design blending methods to determine how input pixels should be combined, and ii) schedule composition workflows for higher composition efficiency with less resource consumption. We focus mainly on the blending methods in this chapter.

2.1 *over* Composition Operator

Blending methods can be divided into two categories: i) order-independent methods and ii) order-dependent methods. Among the order-independent methods, the most widely used one is the Z-buffer algorithm proposed by Catmull and others [10, 56]. The blending result of this method is only determined by the two input pixels' distances to the view point, i.e., the one closer to the view point is taken as the resultant pixel. It also means that the input order does not affect the blending result, hence facilitating the design of more flexible algorithms. Among the order-dependent methods, the most representative one is the Porter-Duff *over* operator [56]. In this operator, two input pixels are combined with a ratio that depends on the first input pixel's α-channel value, which makes the operator non-commutable and imposes a limitation on the design of potential algorithms.

The *over* operator [68] is a traditional and ubiquitous operator in the world of graphics. Due to its simplicity and satisfactory blending performance, this operator has been adopted in numerous visualization practices. For example, in the image composition stage of sort-last parallel visualization, the *over* operator is used to composite final images [55, 41, 42]; in semi-transparent surface rendering, the *over* operator is used to determine the transparency of overlapping surfaces [45, 4]; in volume rendering based on ray tracing/casting, the *over* operator is also used to blend the points sampled along a ray's forward direction [59, 39, 40, 3]. Considering its significant and indispensable role in the visualization field, this operator has even been integrated into operating systems and visualization frameworks/languages by default [2].

Although widely employed, the performance of the traditional *over* operator [68] is largely limited by the following two facts. i) It is a binary operator handling only two RGBA-formatted operands at a time. That is, n RGBA-formatted operands require $n-1$ iterations to blend, hence posing a challenge on its scalability when n is large. ii) It blends a pair of channels (either a color component or the transparency) in two input operands according to their relative positions, thus making itself order sensitive. That is, the input operands need to be first sorted according to a certain criterion (e.g., the depth of a pixel in image composition) and then blended in the sorted order. Such order-dependency may halt the blending process at some point when two consecutive operands are not simultaneously available, regardless of the availability of their downstream operands.

As we step into the big data era, the aforementioned performance issues associated with the traditional *over* operator have become even more severe. For example, today's extreme-scale e-science applications produce colossal amounts of data on the order of terabytes or even petabytes, which must be processed and analyzed in a timely manner for scientific discovery. In many scientific domains, visualization is considered as one of the most important methods for data analytics. As a fundamental unit of these methods, the *over* operator has a significant impact on the overall performance of scientific visualization, especially for asynchronous parallel visualization. Unfortunately, the inherent limitations of the traditional *over* operator cause a critical bottleneck in handling ever-increasing data volumes.

Various research efforts have been made to address the above performance limitations. One commonly considered strategy is to generalize the original operator. Meshkin [45] approximates the *over* composition result for n input pixels by ignoring the order-sensitive parts in their corresponding extended *over* composition formula. Bavoil and Myers [4] approximate n-pixel *over* composition by calculating the average of all input pixels and substituting it for each pixel, which is a special case of the extended n-pixel *over* composition formula with all the input pixels being identical. Patney *et al.* [54] propose a generalized formula for the color components without considering the $\alpha-$channel of transparency. In our work, we generalize the *over* operator for all the channels of the input operands, explore its parallelization feasibility and study its application in both order dependent and order independent compositions.

2.2 A Fully Generalized *over* Image Composition Operator

We denote n RGBA-formatted and order-predefined operands as, $P_1, P_2, \cdots, P_n$. To facilitate our explanation of the generalized *over* operator, we introduce another notation $P_{i,j}$, $1 \leq i \leq j \leq n$, which represents the blending result of P_i *over* P_{i+1} $\cdots$ *over* P_j and is a uniform representation for any possible (raw, intermediate, or final) blending results in the entire operating process. For example, when $i = j$, it refers to the raw operand P_i or P_j; when $i = 1$ and $j = n$, it refers to the final blending result from all the n raw operands.

2.2.1 Extension from Two Operands to Multiple Ones

We propose a fully generalized *over* operator as follows:

$$\alpha_{1,n} = \sum_{\substack{\text{for any } I \subseteq \{\alpha_1 \cdot \alpha_2 \cdots \alpha_n\}, \\ I \neq \emptyset}} \left((-1)^{|I|+1} \cdot \left(\prod_{\text{for any } \alpha_i \in I} \alpha_i \right) \right), \tag{2.1}$$

$$c_{1,n} = \sum_{i=1}^{n} \prod_{1 \le j < i} (1 - \alpha_j) \cdot \alpha_i \cdot C_i \cdot 1, \tag{2.2}$$

where $c_i = [c_{R_i}, c_{G_i}, c_{B_i}]^T$ represents the operand's color component values with premultiplication by α_i, i.e., $c_i = \alpha_i \cdot C_i$. Note that (2.2) was presented in [54]. We provide a brief proof for (2.1) by means of induction.

First, we know that

$$\alpha_{1,1} = \sum_{\substack{I \subseteq \{\alpha_1\} \\ I \neq \emptyset}} (-1)^{|I|+1} |\prod_{\alpha_i \in I} \alpha_i| = (-1)^2 \cdot \alpha_1 = \alpha_1.$$

Then, we assume that (2.1) holds for $n = k-1$, i.e.,

$$\alpha_{1,k-1} = \sum_{\substack{I \subseteq \{\alpha_1 \cdot \alpha_2 \cdots \alpha_{k-1}\} \\ I \neq \emptyset}} (-1)^{|I|+1} |\prod_{\alpha_i \in I} \alpha_i|. \tag{2.3}$$

For $n = k$, we have

$$\begin{aligned}
\alpha_{1,k} =& \alpha_k + \alpha_{1,k-1} - \alpha_{1,k-1} \cdot \alpha_k \\
=& \alpha_k + \sum_{\substack{I \subseteq \{\alpha_1 \cdot \alpha_2 \cdots \alpha_{k-1}\} \\ I \neq \emptyset}} (-1)^{|I|+1} |\prod_{\alpha_i \in I} \alpha_i| \\
&- \sum_{\substack{I \subseteq \{\alpha_1 \cdot \alpha_2 \cdots \alpha_{k-1}\} \\ I \neq \emptyset}} (-1)^{|I|+1} |\prod_{\alpha_i \in I} \alpha_i \cdot \alpha_k| \\
=& \sum_{\substack{I \subseteq \{\alpha_1 \cdot \alpha_2 \cdots \alpha_k\} \\ I \neq \emptyset}} (-1)^{|I|+1} |\prod_{\alpha_i \in I} \alpha_i|.
\end{aligned} \tag{2.4}$$

Hence, (2.1) holds according to the principle of proof-by-induction.

To represent $\alpha_{1,n}$ more concisely, we derive another formula for $\alpha_{1,n}$, i.e.,

$$1 - \alpha_{1,n} = \prod_{j=1}^{n} (1 - \alpha_j), \tag{2.5}$$

which could be proved as follows:

$$\begin{aligned} 1 - \alpha_{1,n} &= 1 - (\alpha_{1,n-1} + \alpha_n - \alpha_{1,n-1} \cdot \alpha_n) \\ &= (1 - \alpha_{1,n-1}) \cdot (1 - \alpha_n) \\ &= (1 - \alpha_{1,n-2}) \cdot (1 - \alpha_{n-1}) \cdot (1 - \alpha_n) \\ &= \cdots = \prod_{j=1}^{n} (1 - \alpha_j). \end{aligned} \tag{2.6}$$

Based on (2.1) and (2.2), we summarize two significant properties of this generalized operator:

- It provides a complete and accurate form of the composited result from n input operands and specifies the exact contribution of each operand to the final result according to its relative position on the to-be-blended list.
- For n available order-predefined operands, it reduces the number of blending steps from $n - 1$ to 1, by plugging each operand into its corresponding position and finishing all computations within one single step.

$$\begin{aligned} c_{1,n} &= c_{1,n-1} + (1 - \alpha_{1,n-1}) \cdot c_n \\ &= c_{1,n-1} + (1 - \alpha_{1,n-1}) \cdot \alpha_n \cdot C_n \\ &= \sum_{i=1}^{n-1} \prod_{1 \le j < i} (1 - \alpha_j) \cdot \alpha_i \cdot C_i + \prod_{j=1}^{n-1} (1 - \alpha_j) \cdot \alpha_n \cdot C_n \\ &= \sum_{i=1}^{n} \prod_{1 \le j < i} (1 - \alpha_j) \cdot \alpha_i \cdot C_i \cdot 1 \end{aligned} \tag{2.7}$$

2.2.2 An In-depth Illustration of the New *over* Operator

To justify the consistency with its predecessor, we illustrate the generalized operator in an image composition scenario as [68], where the α_i, $1 \leq i \leq n$, channel value is interpreted as the area of the sub-pixel region covered by P_i's sub-pixel geometry G_i, $1 \leq i \leq n$. We extend the mutual division assumption from two geometries to n geometries as follows: given n pixels $P_1, P_2, \cdots, P_n$, whose sub-pixel geometries and corresponding covered areas are $(G_1, \alpha_1), (G_2, \alpha_2), \cdots, (G_n, \alpha_n)$, respectively, in the composited pixel $P_{1,n}$, for any $i, j \in [1, n]$ and $i \neq j$, G_i divides both G_j and $P_{1,n}$ into two areas of the same ratio, i.e., $\frac{\alpha_i}{1-\alpha_i}$, and G_j also divides G_i and $P_{1,n}$ into two areas of the same ratio, i.e., $\frac{\alpha_j}{1-\alpha_j}$.

Following the generalized assumption, each geometry G_i, $1 \leq i \leq n$, in the composited pixel $P_{1,n}$ intersects with every other geometry, hence dividing $P_{1,n}$ into 2^n non-overlapping subregions (divisions), as shown in Figure 2.1. Each of these subregions is uniquely identifiable by a set of geometries that cover this subregion, denoted by $G_1^{'} \cap G_2^{'} \cap G_3^{'} \cdots \cap G_i^{'} \cdots \cap G_{n-1}^{'} \cap G_n^{'}$, where $G_i^{'}$ is either G_i or $\overline{G_i}$: G_i means that the sub-region is covered by G_i, and $\overline{G_i}$ means not. To be more specific, we denote the 2^n subregions as $R_1, R_2, \cdots, R_{2^n-1}, R_{2^n}$, and assign each of them a unique set of covering geometries (with 0, 1, ..., n-1, and n covering geometries) as follows

- $\overline{G_1} \cap \overline{G_2} \cap \overline{G_3} \cdots \cap \overline{G_n}$;
- $G_1 \cap \overline{G_2} \cap \overline{G_3} \cdots \cap \overline{G_n}$, $\overline{G_1} \cap G_2 \cap \overline{G_3} \cdots \cap \overline{G_n}$, $\overline{G_1} \cap \overline{G_2} \cap G_3 \cdots \cap \overline{G_n}, \cdots, \overline{G_1} \cap \overline{G_2} \cap \overline{G_3} \cdots \cap G_n$;
- $G_1 \cap G_2 \cap \overline{G_3} \cdots \cap \overline{G_n}$,$G_1 \cap \overline{G_2} \cap G_3 \cdots \cap \overline{G_n}, \cdots, \overline{G_1} \cap \overline{G_2} \cap \overline{G_3} \cdots \cap G_{n-1} \cap G_n$;
- $\cdots$
- $G_1 \cap G_2 \cap G_3 \cap G_4 \cdots \cap G_n$.

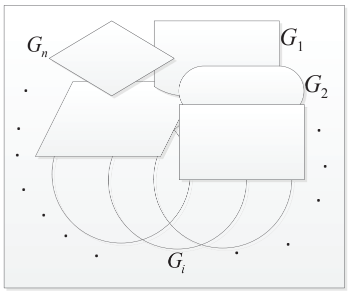


Figure 2.1 A general illustration of dividing the pixel area by n geometries.

Based on the 2^n-subregion division of a pixel, we generalize the calculation of the color components as

$$c_{1,n} = \sum_{k=1}^{2^n} \alpha(R_k) \cdot C(R_k), \tag{2.8}$$

where $\alpha(R_k)$ is the area of sub-region R_k and $C(R_k)$ is the color component value assigned to sub-region R_k. According to the definition of the *over* operator, the value of $C(R_k)$ is determined by the first covering geometry as follows: if there exists any i $(1 \le i \le n)$ in R_k's covering (intersection) expression such that $G_i^{'} = G_i$ and for any existing j $(1 \le j < i), G_j^{'} = \overline{G_j}$, then $C(R_k) = C_i$; otherwise $C(R_k) = 0$.

To facilitate the summation over the 2^n subregions, we further categorize them into $n+1$ groups $g_1, g_2. \cdots, g_{n+1}$, where all the subregions in the same group have the same color component value from the same geometry. The covering expression of each group is as follows:

g_1: $G_1 \cap G_2^{'} \cap G_3^{'} \cdots \cap G_i^{'} \cdots \cap G_{n-1}^{'} \cap G_n^{'}$;

g_2: $\overline{G_1} \cap G_2 \cap G_3^{'} \cdots \cap G_i^{'} \cdots \cap G_{n-1}^{'} \cap G_n^{'}$;

$\cdots$

g_i: $\overline{G_1} \cap \overline{G_2} \cdots \cap \overline{G_{i-1}} \cap G_i \cap G_{i+1}^{'} \cdots \cap G_{n-1}^{'} \cap G_n^{'}$;

$\cdots$

g_n: $\overline{G_1} \cap \overline{G_2} \cap \overline{G_3} \cdots \cap \overline{G_{n-1}} \cap G_n$;

g_{n+1}: $\overline{G_1} \cap \overline{G_2} \cap \overline{G_3} \cdots \cap \overline{G_{n-1}} \cap \overline{G_n}$.

Also, we denote the total area summed over all the subregions in group g_i as α_{g_i}. Under such grouping, we can rewrite (2.8) as

$$c_{1,n} = \sum_{i=1}^{n+1} \alpha_{g_i} \cdot C_i. \tag{2.9}$$

Note that $C(g_{n+1}) = C_{n+1} = 0$ as there is no geometry covering g_{n+1}. For α_{g_i} in (2.9), we have

$$\alpha_{g_i} = \left(\prod_{k=1}^{i-1} \alpha(\overline{G_k})\right) \cdot \alpha(G_i) \cdot \left(\sum_{\substack{G'_j \in \{G_j, \overline{G_j}\} \\ i+1 \leq j \leq n}} \left(\prod_{j=i+1}^{n} \alpha(G'_j) \right) \right), \tag{2.10}$$

where $\alpha(G_i) = \alpha_i$ and $\alpha(\overline{G_i}) = 1 - \alpha_i$, for $1 \leq k < i$. We also have

$$\begin{aligned}
&\sum_{\substack{G'_j \in \{G_j, \overline{G_j}\} \\ k \leq j \leq n}} \prod_{j'=k}^{n} \alpha(G'_{j'}) = \sum_{\substack{G'_j \in \{G_j, \overline{G_j}\} \\ k+1 \leq j \leq n}} \alpha(G_k) \cdot \prod_{j'=k+1}^{n} \alpha(G'_{j'}) \\
&+ \sum_{\substack{G'_j \in \{G_j, \overline{G_j}\} \\ k+1 \leq j \leq n}} \alpha(\overline{G_k}) \cdot \prod_{j'=k+1}^{n} \alpha(G'_{j'}) \\
&= (\alpha(G_k) + \alpha(\overline{G_k})) \cdot \sum_{\substack{G'_j \in \{G_j, \overline{G_j}\} \\ k+1 \leq j \leq n}} \alpha(G_k) \cdot \prod_{j'=k+1}^{n} \alpha(G'_{j'}) \\
&= \sum_{\substack{G'_j \in \{G_j, \overline{G_j}\} \\ k+1 \leq j \leq n}} \alpha(G_k) \cdot \prod_{j'=k+1}^{n} \alpha(G'_{j'}) \\
&= \cdots = \sum_{\substack{G'_j \subseteq \{G_j, \overline{G_j}\} \\ n \leq j \leq n}} \prod_{j'=n}^{n} \alpha(G'_{j'}) = \alpha(G_n) + \alpha(\overline{G_n}) = 1.
\end{aligned} \tag{2.11}$$

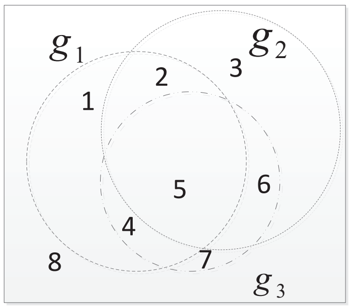


Figure 2.2 A specific illustration of dividing the pixel area by 3 geometries.

Thus, we can rewrite α_{g_i} as $\alpha_{g_i} = \alpha(\overline{G_1}) \cdots \ \alpha(\overline{G_{i-1}}) \cdot \alpha(G_i)$ and expand (2.9) as

$$c_{1,n} = \sum_{i=1}^{n+1} (1-\alpha_1) \cdots (1-\alpha_{i-1}) \cdot \alpha_i \cdot C_i, \tag{2.12}$$

which is exactly the same as (2.2). The above analysis shows that the generalized *over* operator is extended from the original binary operator to work with multiple operands simultaneously.

To make the above analysis more concrete, we consider a specific blending case containing 3 geometries, G_1, G_2, and G_3 in Figure 2.2, where the entire pixel area is divided into $2^3 = 8$ different subregions labeled from 1 to 8. According to the grouping criteria, we further classify these 8 areas into $3+1=4$ different groups, as follows:

g_1: $G_1 \cap \overline{G_2} \cap \overline{G_3}, G_1 \cap G_2 \cap \overline{G_3}, G_1 \cap \overline{G_2} \cap G_3,$
$G_1 \cap G_2 \cap G_3,$ which are labeled as 1, 2, 4, and 5
in Figure 2.2, respectively;

g_2: $\overline{G_1} \cap G_2 \cap \overline{G_3}, \overline{G_1} \cap G_2 \cap G_3,$ which are labeled
as 3 and 6, respectively;

g_3: $\overline{G_1} \cap \overline{G_2} \cap G_3,$ which is labeled as 7;

g_4: $\overline{G_1} \cap \overline{G_2} \cap \overline{G_3}$ which is labeled as 8.

2.3 Asynchronous, Order-known Image Composition

The asynchronous, order-dependent image composition problem has the following features:

- n operands $P_{1,1}, \cdots, P_{i,i}, \cdots, P_{n,n}$ on a certain blending order arrive to the operator asynchronously in an arbitrary order.
- Two available operands $P_{i,j}$ and $P_{i^{'},j^{'}}$ are directly blendable, if their sub-indexes satisfy either $j = i^{'} - 1$ or $j^{'} = i - 1$.
- In each blending, two directly blendable operands $P_{i,j}$ and $P_{j+1,k}$, $1 \leq i \leq j < k \leq n$, are blended into $P_{i,k}$.

Once an operand becomes available, if there is any other operand that fits in the blending order, the operator proceeds with the blending process; otherwise, it waits for the next available operand.

The existing *over* operator is a binary operator, working on two operands at a time. Given a sequence of order-known operands that arrive asynchronously, there exist the following two performance issues worth attentions during application of the original operator: 1) it treats only operands which are "neighbors" to each other, when no such operands available, it needs to halt, nn some extreme cases (depending on the operands' arrival order), the blending process may be delayed until over half of the operands become available; 2) given n "neighboring" operands, it takes $n - 1$ steps to process, which becomes a noticeable bottleneck in the scenario of real-time visualization/composition.

The proposed *over* operator well address the above issues: as soon as an operand becomes available, based on its blending order among all the existing operands, we directly plug it into Equations (2.2) and (2.5) for concurrent blending with all of its upstream operands and further explore the parallelization possibility within (2.2) and (2.5).

2.3.1 Algorithm Design, Analysis, and Optimization

Our designed algorithm based on the generalized *over* operator comes in two versions: a sequential version, referred to as Sequential Generalized *over* Operator based Order-known Composition (SGOKC), and a parallel version, referred to as Parallel Generalized *over* Operator based Order-known Composition (PGOKC).

SGOKC The pseudocode of SGOKC is provided in Algorithm 5, which consists of three parts:

- Part 1 (lines 1 to 6): Initialize the arrays and variables for storing the intermediate or final blending results.
- Part 2 (lines 7 to 20): Use an n-iteration "while" loop, where n is the number of operands for blending, to update the corresponding global $\alpha^{'}$ and color components related to a given operand $P_{j'}$.
- Part 3 (lines 21 to 26): Transform the above intermediate results into the final ones.

Note that Algor. 5 may not perform well when the number of input operands is large. Part 2 is an n-iteration "while" loop with one "for" loop embedded, and is more time-consuming than the other two parts. For the j-th "while" loop, its corresponding "for" loop runs in $O(j)$. Hence, the time complexity for the n-iteration "while" loop is of $O(n^2)$, which also sets the time complexity for the entire algorithm.

PGOKC SGOKC could be further improved through parallelization because i) when an operand $P_{j'}$ arrives, updating the corresponding elements in each color component array is independent of each other; ii) while summing up n elements in each array, instead of adding them up sequentially, a binary-tree based summation scheme can be used to exploit the parallelism.

Following the above analysis, we parallelize Algorithm 5 as PGOKC as shown in Algorithm 23, where all the multiplications upon the arrival of a pixel take place

Algorithm 1 SGOKC($P_{1'}, P_{2'}, \cdots, P_{n'}$)
Input: n RGBA-formatted operands $P_{1'}, P_{2'}, \cdots, P_{n'}$, which arrive in a sequence for blending by the order-dependent n-tuple *over* operator
Output: a blended RGBA-formatted operand $P^{'}$ from the n input operands using the *over* operator

```
 1: Array $a[3][n] = [1, \cdots, 1; 1, \cdots, 1; 1, \cdots, 1]$
 2: Array $P^{'}[4] = [0, 0, 0, 0]$;
 3: $j = 1$;
 4: $\alpha^{'} = 1$;
 5: while $j \leq n$ do
 6:     Receive an operand $P_{j'}$;
 7:     $k = P_{j'}$'s position in the $n$-tuple over operator;
 8:     $\alpha^{'} = \alpha^{'} \cdot (1 - P_{j'}[4])$;
 9:     for $t = 1$ to 3  do
10:         $a[t][k] = a[t][k] \cdot (P_{j'}[t])$;
11:     for $m = k + 1$ to $n$  do
12:         $P_{j'}[1:3][m] = P_{j'}[1:3] \cdot (1 - P_{j'}[4])$;
13:     $j = j - 1$;
14: $P^{'}[4] = 1 - \alpha^{'}$;
15: for $m = 1$ to $n$ do
16:     $P^{'}[1:3] = P^{'}[1:3] + a[1:3][m]$;
17: return  $P^{'}$;
```

Algorithm 2 PGOKC($P_{1'}, P_{2'}, \cdots, P_{n'}$)
Input: n RGBA-formatted operands $P_{1'}, P_{2'}, \cdots, P_{n'}$, which arrive in a sequence for blending by the order-dependent n-tuple *over* operator
Output: a blended RGBA-formatted operand $P^{'}$ from the n input operands using the *over* operator

```
 1: Array $a[3][n] = [1, 1, \cdots, 1; 1, 1, \cdots, 1; 1, 1, \cdots, 1]$;
 2: Array $P^{'}[4] = [0, 0, 0, 0]$;
 3: $j = 1$;
 4: $\alpha^{'} = 1$;
 5: while $j \leq n$ do
 6:     Receive an operand $P_{j'}$;
 7:     $k = P_{j'}$'s position in the $n$-tuple over operator;
 8:     $\alpha' = \alpha' \cdot (1 - P_{j'}[4])$;
 9:     MIMD_MULTI $\lll 3, (n - k + 1) \ggg (a, k, P_{j'})$;
10:     $j = j - 1$;
11: $P^{'}[4] = 1 - \alpha'$;
12: for $m = 1$ to $\lceil \log n \rceil$ do
13:     MIMD_SUM $\lll 3, \lceil \frac{n}{2^m} \rceil \ggg (a, m, n)$;
14: $P^{'}[1] = a[1][1]$;
15: $P^{'}[2] = a[1][2]$;
16: $P^{'}[3] = a[1][3]$;
17: return  $P^{'}$;
```

Function 3 MIMD_MULTI($a[\,], k, P_{j'}$)
Input: a newly available operand $P_{j'}$, its blending order k, the partial color component result matrix a without $P_{j'}$
Output: the partial color component result matrix a with $P_{j'}$ integrated

```
1: int $bid = blockIdx.x$;
2: int $gid = gridIdx.x$;
3: if $bid = 0$ then
4:     $a[gid][k + bid] = a[gid][k + bid] \cdot P_{j'}[4] \cdot P_{j'}[gid]$;
5: else
6:     $a[gid][k + bid] = a[gid][k + bid] \cdot (1 - P_{j'}[4])$;
```

Function 4 MIMD_SUM($a[\,], m, n$)
Input: the partial color component result matrix a, the total round of summation n, the current round of summation n
Output: the final color component result matrix a

```
1: int $bid = blockIdx.x$;
2: int $gid = gridIdx.x$;
3: int $k = \lfloor \log_2 n \rfloor$;
4: int $interval = 2^{k-m}$;
5: if $n \geq bid + interval$ then
6:     $a[gid][bid] = a[gid][bid] + a[bid + interval]$;
```

simultaneously by using the MIMD-featured function, and the final-stage summation employs a binary-tree scheme. The time complexity of Algorithm 23 is reduced to $O(n)$, and the number of steps needed for the final summation is reduced to $\lceil \log n \rceil$. Functions 7 and 22 detail the parallelization strategy in PGOKC.

2.3.2 Theoretical Performance Analysis

To facilitate theoretical analysis of the proposed *over* operator, we consider the situation where the operands become available to the operator one by one at a fixed time interval of $t_a \geq 0$. Especially, when $t_a = 0$, it means that all the operands are available to the operator simultaneously. We conduct theoretical analysis of time cost for both the original and proposed *over* operators.

The Original *over* Operator It is not straightforward to determine the time cost range for the original *over* operator because i) the number of all possible availability orders is $n!$ for n asynchronously available operands, and ii) the blending process highly depends on the arriving or availability order. For example, if the operands arrive in the order of $P_1, P_2, \cdots, P_n$, then the idle time for blending is relatively short; if the operands arrive in the order of $P_1, P_3, \cdots, P_{2i+1}$, $P_2, P_4, \cdots, P_{2i}$, then the blending process would halt till P_2 arrives, hence resulting in $i + 1$ idle time intervals.

The n-operand blending procedure can be viewed as a pipeline of 2 steps, i.e., waiting and blending, and one operand gets blended during one repetition of the pipeline. To be more specific, in Step 1, the procedure would wait for a newly available operand and buffer them before moving onto Step 2, which incurs a constant time cost of t_a. In Step 2, the procedure would look for blendable operands for the newly available one for blending; if not found, it would skip the current step. Depending on the number of blendable operands that have been found, the time cost for Step 2 contains some uncertainties, which need to be addressed.

We consider the following conditions for blending:

1. For any instance of Step 2 in the pipeline, all found blendable operands are blended.
2. The blending of each channel in two operands is performed sequentially.
3. There are 18 basic arithmetic operations for blending and they are treated equally in terms of time cost.

We provide the following lemmas for time cost analysis.

Lemma 2.3.1. *The number of blendable operands for the newly-available operand in each blending step is no more than two.*

Lemma 2.3.2. *All n arriving operands are blended into one after n repetitions of the blending step.*

Both of the above lemmas are straightforward. Their proofs can be established by contradiction, and are omitted due to the page restriction.

To quantify the time cost of Step 1, we further investigate the n-repetition pipeline. Within each repetition of the pipeline, the number of blended operands could be 0, 2 or 3, according to which, we divide the n repetitions into 3 groups: i) 0-blending repetition, ii) 1-blending repetition, and iii) 3-blending repetition, denoted by G_0, G_2, and G_3, respectively. In addition, we denote the size of each group as N_0, N_2, and N_3, respectively. Obviously,

$$N_0 + N_2 + N_3 = n. \tag{2.13}$$

The pipeline is empty initially without any operand. As the repetition proceeds, one repetition in G_0 or G_2 increases the number of to-be-blended operands in the pipeline by 1 or 0, and one repetition in G_3 decreases this number by 1. When n repetitions finish, n input operands are blended into one. Considering the repetitions and the way they change the number of to-be-blended operands in the pipeline, we derive the following equation

$$0 + 1 \cdot N_0 + 0 \cdot N_2 + (-1) \cdot N_3 = 1, \tag{2.14}$$

which is equivalent to

$$N_0 - N_3 = 1. \tag{2.15}$$

In (2.13) and (2.15), there are 3 unknowns, so one unknown needs to serve as a free variable. For simplicity, we choose N_0 as the free variable and represent the other two variables as follows

$$N_2 = n + 1 - 2N_0, \tag{2.16}$$

$$N_3 = N_0 - 1. \tag{2.17}$$

Also, $N_2 \geq 0, N_3 \geq 0$, and it follows that

$$N_2 = n + 1 - 2N_0 \geq 0, \tag{2.18}$$

$$N_3 = N_0 - 1 \geq 0. \tag{2.19}$$

Combining (2.18), (2.19) with (2.16), and (2.17), we have

$$1 \leq N_0 \leq \lceil \frac{n}{2} \rceil, \tag{2.20}$$

$$0 \leq N_2 \leq n - 1, \tag{2.21}$$

$$0 \leq N_3 \leq \lceil \frac{n}{2} \rceil - 1. \tag{2.22}$$

(2.20), (2.21), and (2.22) specify the possible size of each group.

Given n operands, there are total $n!$ different availability orders, each of which uniquely leads to one blending order or procedure. According to the number N_0 of 0-blending repetitions in each blending procedure, we divide $n!$ blending procedures into $\lceil \frac{n}{2} \rceil$ groups, and use B_i, $1 \leq i \leq \lceil \frac{n}{2} \rceil$ to represent the group whose corresponding N_0 is i. In addition to the above grouping, we introduce the following notations:

- $o_j (1 \leq j \leq n!)$ – a specific availability order;
- $b_j (1 \leq j \leq n!)$ – o_j's corresponding blending procedure;
- $T_{b_j} (1 \leq j \leq n!)$ – the time cost of b_j;

- $r_{j,k}(1 \leq j \leq n!, 1 \leq k \leq n)$ – the blending step in the k-th repetition of the pipeline for b_j;
- $t_{j,k}(1 \leq j \leq n!, 1 \leq k \leq n)$ – the earliest start time of $r_{j,k}$;
- $t^{'}_{j,k}(1 \leq j \leq n!, 1 \leq k \leq n)$ – the time cost to finish $r_{j,k}$;
- t_b – the time cost to perform a two-operand blending;
- $\overbrace{0\cdots0}^{i}\overbrace{2\cdots2}^{n+1-2i}\overbrace{3\cdots3}^{i-1}$ – a special notation for b_j's that satisfy the condition where there is 0 blendable operand in each of the first i repetitions, 2 blendable operands in each of the immediately following $n+1-2i$ repetitions, and 3 blendable operands in each of the last $i-1$ repetitions.

Based on these notations, we define a series of lemmas that lead to the final time cost estimation.

Lemma 2.3.3. *Within the blending group B_i, $(1 \leq i \leq \lceil \frac{n}{2} \rceil)$, the blending order b_{max} with the maximum blending time has the form of $\overbrace{0\cdots0}^{i}\overbrace{2\cdots2}^{n+1-2i}\overbrace{3\cdots3}^{i-1}$.*

Proof. We prove this lemma in two steps: i) find the maximum blending time in the given blending group, and ii) prove that this maximum blending time is incurred by the specified b_{max} blending order.

It is obvious that

$$t_{j,k} = \max\{t_{j,k-1} + t^{'}_{j,k-1}, k \cdot t_a\} \ (t^{'}_{j,k-1} \in \{0, t_b, 2 \cdot t_b\}) \tag{2.23}$$

and

$$t_{j,k-1} + t_a \geq (k-1) \cdot t_a + t_a = k \cdot t_a (1 \leq k \leq n). \tag{2.24}$$

Plugging (2.24) into (2.23), for $1 \leq k \leq n$, we have

$$\begin{aligned} t_{j,k} &\leq \max\{t_{j,k-1} + t^{'}_{j,k-1}, t_{j,k-1} + t_a\} \\ &\leq t_{j,k-1} + \max\{t^{'}_{j,k-1}, t_a\}. \end{aligned} \tag{2.25}$$

By iteratively expanding (2.25) from $k = n$ to 1, we get

$$\begin{aligned} t_{j,n} &\leq t_{j,n-1} + \max\{t'_{j,n-1}, t_a\} \\ &\leq t_{j,n-2} + \max\{t'_{j,n-2}, t_a\} + \max\{t'_{j,n-1}, t_a\} \\ &\leq \cdots \\ &\leq \max\{t'_{j,1}, t_a\} + \max\{t'_{j,2}, t_a\} + \cdots + \max\{t'_{j,n-1}, t_a\}. \end{aligned} \tag{2.26}$$

The $n - 1$ "max" operators on the right side of (2.26) are solvable. For $t'_{j,k}$ $(k = 1, 2, \cdots, n-1)$, we know that i of them are 0, $n + 1 - 2i$ are t_b, and the rest $i - 1$ are $2 \cdot t_b$, according to (2.16) and (2.17). Thus, to solve these $n - 1$ operators, we need to know the relation between t_a and t_b in terms of their quantities. Considering that t_a and t_b are independent of each other, we consider all of their possible relations for completeness:

- If $t_a \leq t_b$, i of the n max operators yield t_a, $n + 1 - 2i$ yield t_b, and $i - 1$ yield $2 \cdot t_b$. Thus, we have

$$t_{j,n} \leq i \cdot t_a + (n + 1 - 2 \cdot i) \cdot t_b + (i - 1) \cdot 2 \cdot t_b; \tag{2.27}$$

- If $t_b \leq t_a \leq 2 \cdot t_b$, similarly, we have

$$t_{j,n} \leq i \cdot t_a + (n + 1 - 2 \cdot i) \cdot t_a + (i - 1) \cdot 2 \cdot t_b; \tag{2.28}$$

- If $2 \cdot t_b \leq t_a$, we have

$$t_{j,n} \leq i \cdot t_a + (n + 1 - 2 \cdot i) \cdot t_a + (i - 1) \cdot t_a. \tag{2.29}$$

Furthermore, we would like to show that in any of the above three cases, the "=" symbol is established for $b_{j'}$, which has the form of $\overbrace{0 \cdots 0}^{i}\overbrace{2 \cdots 2}^{n+1-2i}\overbrace{3 \cdots 3}^{i-1}$.

If $t_a \leq t_b$, for repetition $r_{j',k}$ $(k \in [1, i])$, their corresponding $t'_{j',k} = 0$. Plugging it into (2.23), we have

$$t_{j',k} = \max\{t_{j',k-1}, k \cdot t_a\} \ (k \in [1, i]).$$

Since $t_{j',0} = 0$, we get

$$t_{j',k} = k \cdot t_a (k \in [1, i]).$$

For repetition $r_{j',k}, \ k \in [i+1, n-i+1]$, according to (2.23) and the relation $t_a \leq t_b$, we get

$$\begin{aligned} t_{j',k-1} + t_b &\geq (k-1) \cdot t_a + t_a \\ &\geq k \cdot t_a. \end{aligned} \tag{2.30}$$

Plugging (2.30) into (2.23), we get

$$\begin{aligned} t_{j',k} &= \max\{t_{j',k-1} + t_b, \ k \cdot t_a\} \\ &= t_{j',k-1} + t_b. \end{aligned}$$

Since $t_{j',i} = i \cdot t_a$, we have

$$t_{j',k} = i \cdot t_a + (k-i) \cdot t_b \ (k \in [i+1, n-i+1]).$$

For $k \in [n-i+2, n]$, we similarly get

$$t_{j',k} = i \cdot t_a + (n - 2 \cdot i + 1) \cdot t_b + 2 \cdot (k - n + i - 1) \cdot t_b;$$

Especially for $t_{j',n}$, we have

$$\begin{aligned} t_{j',n} &= i \cdot t_a + (n - 2 \cdot i + 1) \cdot t_b + 2 \cdot (n - n + i - 1) \cdot t_b \\ &= i \cdot t_a + (n - 2 \cdot i + 1) \cdot t_b + 2 \cdot (i - 1) \cdot t_b. \end{aligned}$$

In the cases where $t_b \leq t_a \leq 2 \cdot t_b$ and $2 \cdot t_b \leq t_a$, we can prove that $b_{j'}$ would incur the maximum time cost similarly as we do for case $t_b \geq t_a$. Hence, $b_{max} = b_{j'}$. Proof ends. □

For convenience, we denote $T_{b_{max}}(B_i)$ by $max(T_{B_i})$.

Lemma 2.3.4. *For any $i \in [1, n-1]$ and its corresponding $max(T_{B_i})$, $max(T_{B_i}) \leq max(T_{B_{i+1}})$.*

Proof. Similarly, we prove Lemma 2.3.4 case by case.

If $t_a \leq t_b$,

$$max(T_{B_i}) = i \cdot t_a + (n + 1 - 2 \cdot i) \cdot t_b + (i - 1) \cdot 2 \cdot t_b,$$

$$max(T_{B_{i+1}}) = (i + 1) \cdot t_a + (n - 2 \cdot i - 1) \cdot t_b + i \cdot 2 \cdot t_b,$$

$$max(T_{B_{i+1}}) - max(T_{B_i}) = t_a \geq 0;$$

If $t_b \leq t_a \leq 2 \cdot t_b$,

$$max(T_{B_i}) = i \cdot t_a + (n + 1 - 2 \cdot i) \cdot t_a + (i - 1) \cdot 2 \cdot t_b,$$

$$max(T_{B_{i+1}}) = (i + 1) \cdot t_a + (n - 2 \cdot i - 1) \cdot t_a + i \cdot 2 \cdot t_b,$$

$$max(T_{B_{i+1}}) - max(T_{B_i}) = t_a - 2 \cdot t_a + 2 \cdot t_b = 2 \cdot t_b - t_a \geq 0;$$

If $2 \cdot t_b \leq t_a$,

$$max(T_{B_i}) = i \cdot t_a + (n + 1 - 2 \cdot i) \cdot t_a + (i - 1) \cdot t_a,$$

$$max(T_{B_{i+1}}) = (i + 1) \cdot t_a + (n - 2 \cdot i - 1) \cdot t_a + i \cdot t_a,$$

$$max(T_{B_{i+1}}) - max(T_{B_i}) = t_a - 2 \cdot t_a + t_a = 0.$$

Proof ends. □

Theorem 2.3.5. *For any $b_j \in B_i(1 \leq i \leq \lceil \frac{n}{2} \rceil)$ and its corresponding T_{b_j}, $T_{b_j} \leq max(T_{B_{\lceil \frac{n}{2} \rceil}})$ and*

$$max(T_{B_{\lceil \frac{n}{2} \rceil}}) = \begin{cases} \lceil \frac{n}{2} \rceil \cdot t_a + n \cdot t_b \\ \textit{if } t_a \leq t_b, \\ (n + 1 - \lceil \frac{n}{2} \rceil) \cdot t_a + (2 \cdot \lceil \frac{n}{2} \rceil - 1) \cdot t_b \\ \textit{if } t_b \leq t_a \leq 2 \cdot t_b, \\ n \cdot t_a + t_b \\ \textit{if } 2 \cdot t_b \leq t_a. \end{cases} \tag{2.31}$$

Proof.

$$\begin{aligned} max(T_{B_{\frac{n+1}{2}}}) =& k \cdot t_a + (n - 2k + 1) \cdot \max\{t_a, t_b\} \\ &+ (k - 1) \cdot \max\{t_a, 2 \cdot t_b\} + t_b. \end{aligned}$$

Proof ends. □

For convenience, we further denote $max(T_{B_{\lceil \frac{n}{2} \rceil}})$ as $max(T_B)$, which represents the maximum possible time cost of the original *over* operator.

Theorem 2.3.6. *Given a set of blending groups B_i, $(1 \leq i \leq \lceil \frac{n}{2} \rceil)$, the minimum blending time $T_{b_{min}}$ among all the groups is incurred by any blending order in B_1, i.e., $T_{b_{min}} = T(B_1) = (n-1) \cdot \max\{t_a, t_b\} + t_a$.*

Proof. If $t_b \leq t_a$, for any blending order b_j and its corresponding $t_{j,n}$, according to (2.24), we know that

$$t_{j,n} \geq n \cdot t_a. \tag{2.32}$$

If $t_b \geq t_a$, according to (2.23), we get

$$\begin{aligned} t_{j,n} &\geq t_{j,n-1} + t^{'}_{j,n} \\ &\geq t_{j,n-2} + t^{'}_{j,n-1} + t^{'}_{j,n} \\ &\geq \cdots \\ &\geq t^{'}_{j,1} + t^{'}_{j,2} + \cdots + t^{'}_{j,n} = \sum_{m=1}^{n} t^{'}_{j,m}. \end{aligned} \tag{2.33}$$

According to the definition of $t^{'}_{j,m}$, we know that $\sum_{m=1}^{n} t^{'}_{j,m}$ represents the total time for blending only, which remains the same for any arriving order and is equal to $(n-1) \cdot t_b$. In addition to the first waiting time t_a, we can rewrite T_{b_j} as

$$T_{b_j} \geq (n-1) \cdot t_b + t_a. \tag{2.34}$$

Combining (2.32) and (2.34), we have

$$T_{b_j} \geq (n-1) \cdot \max\{t_a, t_b\} + t_a. \tag{2.35}$$

Therefore, the minimum blending time $T_{b_{min}} = (n-1) \cdot \max\{t_a, t_b\} + t_a$.

For any blending order $b \in B_1$, it has the form of $\overbrace{0}^{1}\overbrace{2 \cdots 2}^{n-1}$, which is a regular 2-step pipeline, and hence its corresponding blending time is $T_{b, b \in B_1} = (n-1) \cdot \max\{t_a, t_b\} + t_a$. Proof ends. □

Fully Generalized *over* Operator For the fully generalized *over* operator, we view its blending procedure as a 2-step pipeline, i.e., waiting and multi-operand blending, plus one final summation step.

We introduce several more notations to facilitate the time cost analysis of the proposed *over* operator:

- T_{G_1} – the end time of the two-step pipeline;
- t_c – the time cost of the multi-operand blending step;
- t_d – the time cost of the final summation step;
- T_G – the total time cost of the blending procedure using the fully generalized *over* operator.

The blending pipeline of the fully generalized *over* operator works in the same way as the original one in the first step, but improves significantly in the second step. According to Alg. 23, for any newly available operand $P_{j'}$ and its order k among the operands, the fully generalized operator launches $1 + 3 \cdot (n - k + 1)$ threads, one for the α-channel and $n - k + 1$ for each of three color component channels. The thread for the α-channel updates $\alpha^{'}$ as shown in line 8 of Alg. 23, which essentially contains two basic multiplications. Each of the $n - k + 1$ threads for each color component channel updates one single term in the series in line 4 or 6 of Alg. 7. Updating either of them uses only two basic operations, so the workload is balanced for each of the $n-k+1$ threads. Ideally, the time cost of the multi-operand blending step is incurred by two basic operations, and can be calculated as

$$T_{G_1} = (n - 1) \cdot \max\{t_a, t_c\} + t_c. \tag{2.36}$$

The final summation step is to add up all n serial terms in each color component channel. Using the optimal binary-tree structure, the n-operand summation can be done with $\lceil \log n \rceil$ basic operations. The best performance is achieved if the summation takes place simultaneously on each color component channel. Thus, the final step involves $\lceil \log n \rceil$ basic operations.

Combining the time cost of the pipeline and that of the final stage, we obtain the total time cost of the blending procedure using the proposed *over* operator:

$$\begin{aligned} T_G &= T_{G_1} + t_d \\ &= (n-1) \cdot \max\{t_a, t_c\} + t_c + t_d. \end{aligned} \tag{2.37}$$

Different from the variable time cost of the original *over* operator, the fully generalized *over* operator guarantees a stable and consistent performance. Since t_b in (2.35) involves 18 basic operations, t_c involves 2 basic operations and t_d involves $\lceil \log n \rceil$ basic operations, the fully generalized operator incurs less computing time in the blending step. Especially when $t_a \leq t_c < t_b$, the proposed *over* operator performs much better than the original one.

2.3.3 Implementation and Experimental Results

To achieve a high level of parallelism, we implement the new *over* operator in a heterogeneous computing environment comprised of both CPUs and GPUs. We use two computer languages to implement Algorithm 23, i.e., *C* for CPU programming and CUDA for GPU programming. To be more specific, the lines 9 and 14 in Algorithm 23 are parallel functions, which are executed with a large number of threads and are hence implemented in CUDA. The rest of the code is sequential and is hence implemented in *C*. Since there is no global memory addressing between CPUs and GPUs in existing heterogeneous computing environments, explicit data transfer is needed to support communications between them, which inevitably incurs an overhead. In practice, we overlap such data transfer overhead with other activities to minimize the cost.

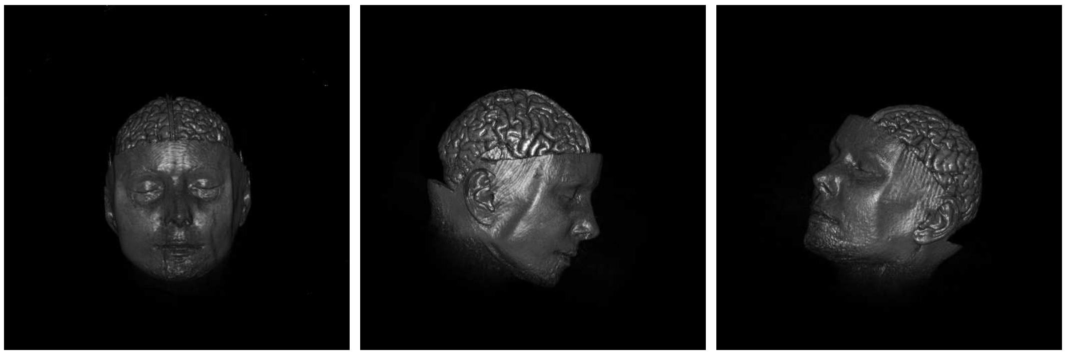


Figure 2.3 Composited images of Human Brain.

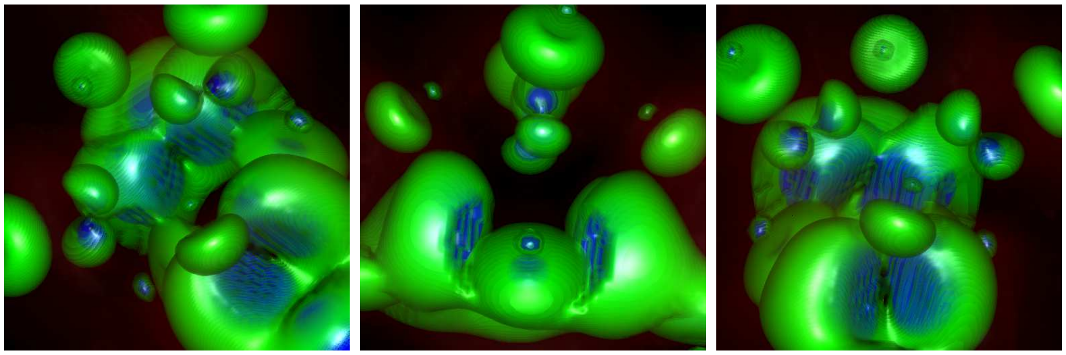


Figure 2.4 Composited images of HIPIP Surfaces.

To test and evaluate the proposed *over* operator, we apply it to the image composition of a 3D human brain volume dataset using parallel visualization. We plot the final images composited by the proposed *over* operator in Figure 2.3, which justifies the validity of the proposed *over* operator and the correctness of our implementation.

For a practical performance evaluation of these two operators, we use the image composition task in Figure 2.3, 2.4, 2.5, as a benchmark with a sequence of partial images of the same size without any blank pixels. Each input image is assigned a certain sequential blending index. For a comprehensive comparison, in the image composition experiments, we consider 2 image sizes: 2048^2 and 3072^2; 3 different numbers of input images: 8, 16, and 32; 6 image arriving intervals: 0s, 0.1s, 0.2s, 0.3s, 0.4s, and 0.5s; and 7 different image arriving (i.e., availability) orders, in each sequence of input images. We define the k-distance arriving order as an arriving

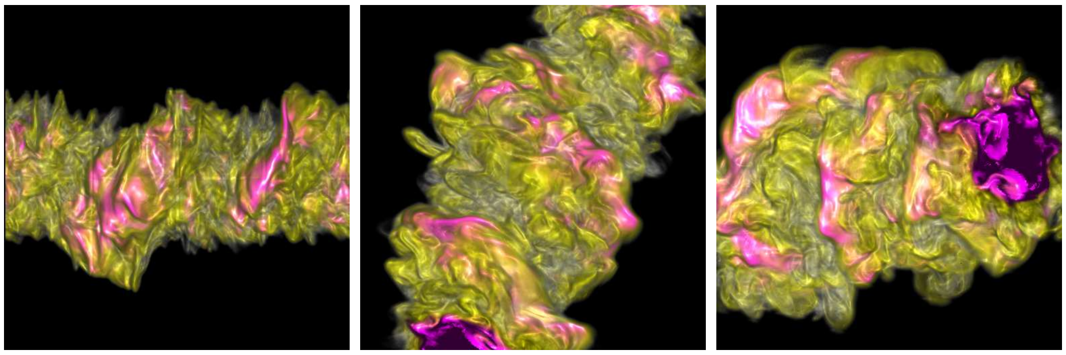


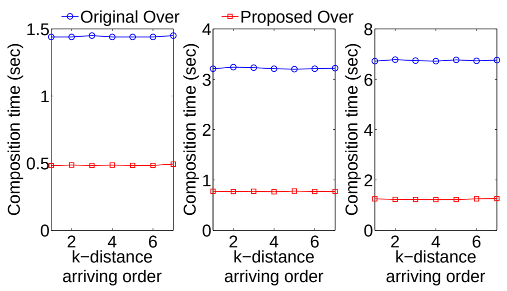
Figure 2.5 Composited images of Jet Ejections.

order where the difference between the indices (which refer to the blending order) of any two subsequently arriving input images is k. We generate seven different arriving orders as follows. Given n operands (i.e., partial images), at time t, when $k \neq 2$, the index of the newly arriving operand is computed as

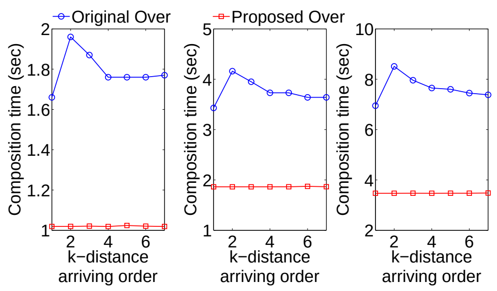
$$F(k,t,n) = \begin{cases} (t \bmod \lceil n/k \rceil) \cdot k + \lfloor t/\lceil n/k \rceil \rfloor \\ \text{if } 0 \leq t < (n \bmod k) \cdot \lceil n/k \rceil, \\ (t \bmod \lfloor n/k \rfloor) \cdot k + n - 1 \\ -\lfloor (n-1-t)/\lfloor n/k \rfloor \rfloor \\ \text{if } (n \bmod k) \cdot \lceil n/k \rceil \leq t < n, \end{cases} \tag{2.38}$$

where $k \in [1,3,4,5,6,7]$, $n \in [8,16,32]$, and $t \in [0,n)$. When $k = 2$, the index of the newly arriving operand is computed as

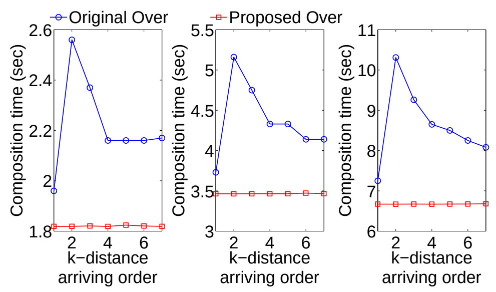
$$F(k,t,n) = \begin{cases} (t \bmod \lceil n/k \rceil) \cdot k + 1 \\ \text{if } 0 \leq t < (n \bmod k) \cdot \lceil n/k \rceil, \\ (t \bmod \lfloor n/k \rfloor) \cdot k \\ \text{if } (n \bmod k) \cdot \lceil n/k \rceil \leq t < n. \end{cases} \tag{2.39}$$



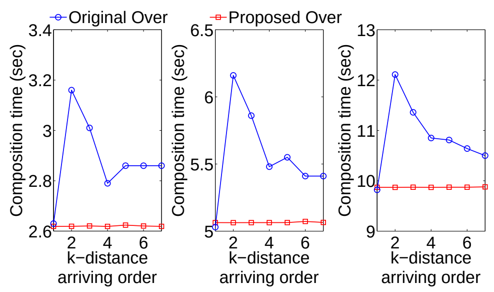
(a) Arriving interval of 0s.



(b) Arriving interval of 0.1s.



(c) Arriving interval of 0.2s.

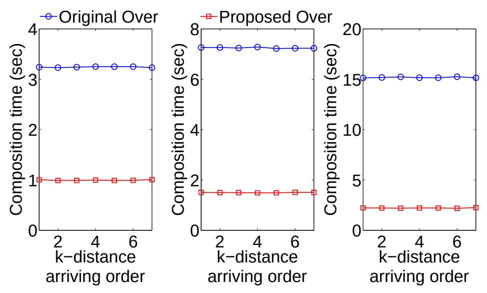


(d) Arriving interval of 0.3s.

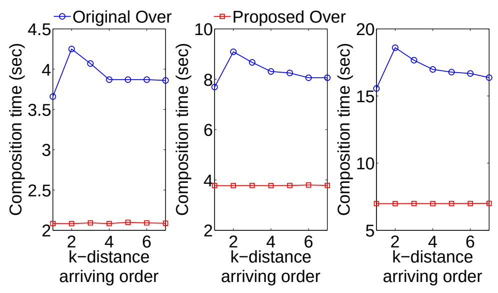
Figure 2.6 Comparison of two *over* operators using images of size 2048^2: composition time versus arriving order. The three sub-figures in Figures 2.6a, 2.6b, 2.6c, and 2.6d correspond to the cases of 8, 16, 32 input images, respectively, from left to right.

We plot the experimental results in Figures 2.6 and 2.7, which show that given the same arriving interval and the same number of input images of the same size, the fully generalized *over* operator achieves a robust blending performance against different arriving orders, while the original one suffers from such variations in the arriving order. We observe that the 2-distance arriving order, whose corresponding blending procedure is depicted as $\overbrace{0\cdots0}^{\lceil\frac{n}{2}\rceil}\ \overbrace{2\cdots2}^{n+1-2\cdot\lceil\frac{n}{2}\rceil}\ \overbrace{3\cdots3}^{\lceil\frac{n}{2}\rceil-1}$, incurs the most time cost, while the 1-distance arriving order, whose corresponding blending procedure is depicted as $\overbrace{0\cdots0}^{1}\overbrace{2\cdots2}^{n-1}$, incurs the least time cost. These observations are consistent with our theoretical analysis results.

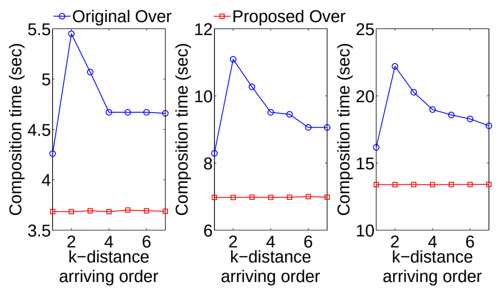
The arriving interval also affects the performance of the blending algorithms. From Figures 2.6 and 2.7, we observe that the fully generalized *over* operator significantly outperforms the best case of the original one when the arriving interval



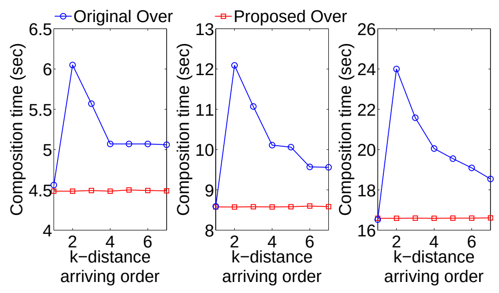
(a) Arriving interval of 0s.



(b) Arriving interval of 0.2s.



(c) Arriving interval of 0.4s.



(d) Arriving interval of 0.5s.

Figure 2.7 Comparison of two operators using images of size 3072^2: composition time versus arriving order. The three sub-figures in Figure 2.7a, 2.7b, 2.7c, and 2.7d correspond to the cases of 8, 16, and 32 input images, respectively, from left to right.

is small, as shown in Figures 2.6a, 2.6b, 2.7a, and 2.7b. As the arriving interval increases, the performance difference between these two operators decreases until converging, as shown in Figure 2.6c, 2.6d, 2.7c, and 2.7d. In addition, the arriving intervals leading to the performance convergence are almost identical for the input images of the same size, regardless of the number of input images.

With a larger image size, both of the operators achieve about the same performance with a larger arriving interval. Based on our observations, such an arriving interval leading to performance convergence usually approximates the time cost of blending two images of the same size using the original *over* operator. With a smaller arriving interval, the input images arrive faster and the blending operation becomes the bottleneck, so optimizing the blending operation would improve the overall performance. The fully generalized *over* operator achieves more performance gains with smaller arriving intervals. As the arriving interval increases, the performance gain becomes marginal. When the arriving interval is larger than the blending time, the arriving interval becomes the bottleneck and there is not much performance gain through blending optimization. These results provide guidelines for optimizing asynchronous, order-dependent image composition.

2.4 Asynchronous, Order-unknown Image Composition

The original *over* operator is a binary order-dependent operator, working on two operands at a time in a strict blending order. Given a sequence of input operands that arrive in an arbitrary order, the original operator may suffer from serious performance disadvantages, because it must wait until all the operands have arrived.

The performance limitation due to order dependency still stands out in practice. In many existing blending frameworks, the operands must be pre-sorted before the actual composition can take place [54][28], hence causing a significant performance issue, especially in dynamic environments where there is a long delay in producing all the operands. Unfortunately, very limited efforts have been devoted to this issue. Given the proposed *over* operator, our work removes the order dependency of the existing *over* operator and hence makes an advancement in the field.

In an image composition problem where the operands arrive in an arbitrary order, the i-th term $\prod_{1\leq j<i}(1-\alpha_j)\cdot\alpha_i\cdot C_i\cdot 1$ of (2.1) calculates the color component value of the i-th pixel $P_{i'}$ weighted by each of its $i-1$ upstream pixels in the form of $(1-\alpha_j)$, and is free of dependency with any downstream pixels. Also, according to the associative law of multiplication, the order of the operand's weight has no effect on the product. For a newly arriving pixel, it follows that the weight factor in the corresponding term is only determined by its relative depth relationship with all the other pixels that have arrived: 1) the current pixel will be affected by all of its upstream pixels that have arrived, and 2) it will make a contribution to each of its downstream pixels that have arrived.

```
struct  P_Pack
{
     P_Pack( ): W(1.0){ }
     float   C[3];
     float   D;
     float   W;
     float   α;
};
```

Figure 2.8 The P_Pack{} data structure for sequential image composition based on the proposed operator.

It is worth pointing out that (2.5) for computing the α value does not have any order restriction since every operand in (2.5) makes the same contribution. Hence, when an operand (pixel) arrives, we can immediately multiply its α value with the accumulated one.

2.4.1 Algorithm Design and Analysis

To apply the proposed *over* operator to a blending scenario where the operands arrive in an arbitrary order, we design two versions of the algorithm for two different platforms: Sequential Generalized *over* Operator based Order-unknown Composition (SGOUC) for sequential execution, and Parallel Generalized *over* Operator based Order-unknown Composition (PGOUC) for parallel execution.

2.4.2 SGOUC

In SGOUC, as shown in Figure 2.8, we use a data structure P_Pack to store the intermediate results for a pixel P: a float array C for P's three color components in the order of R, G, B, a float variable α for P's α channel, a float variable "D" for P's actual depth value, a float variable "W" for P's accumulated weight factor from upstream pixels, which is initialized to be 1.0. The pseudocode of SGOUC is provided in Algorithm 5, which consists of three parts.

- Part 1 (Lines 1 to 3): initialize the variables.

- Part 2 (Lines 4 to 15): The j-th arriving operand $P_j^{'}$ is assigned the j-th instance of P_Pack from S_P. The "for" loop updates the W components of all the existing instances: for instances whose W components are larger than that of the new one, multiply their "W" components by $(1 - S\_P[j].\alpha)$; for instances whose W components are smaller than that of the new one, multiply the j-th pixel's "W" component by $(1 - S\_P[i].\alpha)$ from all those smaller instances one at a time.
- Part 3 (Lines 16 to 18): subtract C_α from 1 to obtain the α channel of the composited pixel, and add up the color channels of all the existing instances to obtain the color channels of the composited pixel.

In Part 2, for the j-th iteration of the "for" loop, its embedded "for" loop runs in $O(j)$. Hence, the time complexity of Algorithm 5 is of $O(n^2)$.

Function 5 SGOUC($P_1^{'}, P_2^{'}, \cdots, P_n^{'}$)
Input: n RGBA-formatted operands $P_1^{'}, P_2^{'}, \cdots, P_n^{'}$ arriving in an arbitrary order
Output: a single RGBA-formatted blended operand $P^{'}$

```
 1: P_Pack Array $S\_P[n]$;
 2: Array $P^{'}[4] = [0, 0, 0, 0]$;
 3: float $C\_\alpha = 1$;
 4: for $(j = 0;\ j \leq n - 1;\ j++)$ do
 5:     Receive an operand $P_j^{'}$;
 6:     $S\_P[j].D = P_j^{'}(D)$;
 7:     $S\_P[j].C = P_j^{'}(C)$;
 8:     $S\_P[j].\alpha = P_j^{'}(\alpha)$;
 9:     $C\_\alpha = C\_\alpha \cdot (1 - P_j^{'}(\alpha))$;
10:     for $(i = 1;\ i \leq j - 1;\ i++)$ do
11:         if $(S\_P[i].D > S\_P[j].D)$ then
12:             $S\_P[i].W = S\_P[i].W \cdot (1 - S\_P[j].\alpha)$;
13:         else
14:             if $(S\_P[i].D < S\_P[j].D)$ then
15:                 $S\_P[j].W = (1 - S\_P[i].\alpha) \cdot S\_P[j].W$;
16: $P^{'}[3] = 1 - C\_\alpha$;
17: for $(m = 0;\ m \leq n - 1;\ m++)$ do
18:     $P^{'}[0:2] = P^{'}[0:2] + S\_P[m].C \cdot S\_P[m].W \cdot S\_P[m].\alpha$;
19: return $P^{'}$;
```

```
struct  ParaP_Pack
{
   ParaP_Pack( ): W(1.0), pNext(NULL){ }
   float          C[3];
   float          D;
   float          W;
   float          α;
   ParaP_Pack  *pNext;
};
```

Figure 2.9 The ParaP_Pack{} data structure for parallelized image composition based on the proposed operator.

2.4.3 PGOUC

We parallelize SGOUC for further performance improvement. There exist several parallelizable operations in SGOUC, for example, the variable updates as a new operand arrives and the final summation operations. Such parallelization improves time efficiencies of SGOUC but does not change SGOUC's time complexity. In Algorithm 1, each "for" loop is to decide the relative order between the new operand and all existing ones to calculate their "W" components. There are two cases: Case 1: the operands with larger "D" component values than that of the new one are modified by the new one. Case 2: the new operand is modified by those with smaller "D" component values than itself. In Case 1, the same weight factor are applied to all existing operands, which could be parallelized. In Case 2, different weight factors are applied to the same operand, which makes direct parallelization impossible because of the possible conflict in concurrently modifying the same variable. For parallelization, as shown in Figure 4.3, we use a new ParaP_Pack structure with a pointer component "$pNext$", which represents the index of a ParaP_Pack instance whose "D" component is the smallest among those with larger "D" components than the current operand. By default, "$pNext$" is set to be $NULL$, indicating that there is no downstream operand found yet.

With the "*pNext*" component, we are able to parallelize the modifications to the same new operand, by directly setting the arriving operand's "W" component to be $(1 - P\_P[i].\alpha) \cdot P\_P[i].W$, where $P\_P[i]$ satisfies that $P\_P[i].D \leq P_j^{'}(D)$ and $P\_P[P\_P[i].pNext].D \geq P_j^{'}(D)$. Here, $P\_P[i]$ represents the existing immediate upstream (preceding) operand of $P_j^{'}$, $P\_P[i].W$ stores the weight factors from all the existing upstream operands of $P\_P[i]$, and $(1 - P\_P[i].\alpha) \cdot P\_P[i].W$ is the weight factors from all the existing upstream operands of $P_j^{'}$.

The pseudocode of PGOUC is provided in Algorithm 23, which uses 3 MIMD-featured functions, i.e., MIMDMULTI, MIMDWEIGHT and MIMDSUM, given in Functions 7, 8 and 22, respectively. MIMDMULTI updates the W components of all existing instances in parallel in the embedded "for" loop and reduces the time complexity of each embedded "for" loop to the order of $O(1)$. MIMDWEIGHT and MIMDSUM together parallelize the previous n-step summations at the end of SGOUC by organizing them in a binary tree and reduces the time cost to the order of $\lceil \log n \rceil$. Parallelizing these two time-consuming parts in PGOUC results in a reduced overall time complexity of $O(n)$. We illustrate difference of the SGOUC and PGOUC in Figure 2.10 for better understanding

2.4.4 Theoretical Performance Analysis

We consider n operands arriving in an arbitrary order with arriving intervals of $t_1, \cdots, t_i, \cdots, t_{n-1}$[1]. We use T_R and T_G to represent the time cost of the composition procedures based on the traditional and proposed *over* operators, respectively.

With the traditional *over* operator, the composition cannot start until all operands become available and are sorted globally. After sorting, the operands are composited one at a time in each channel in the sorted order. Hence, the total time cost using a traditional *over* operator contains three parts for waiting, sorting, and

[1]The time interval between the $(i-1)$-th and i-th arriving operands is t_i, $i = 1, 2, \ldots, n-1$.

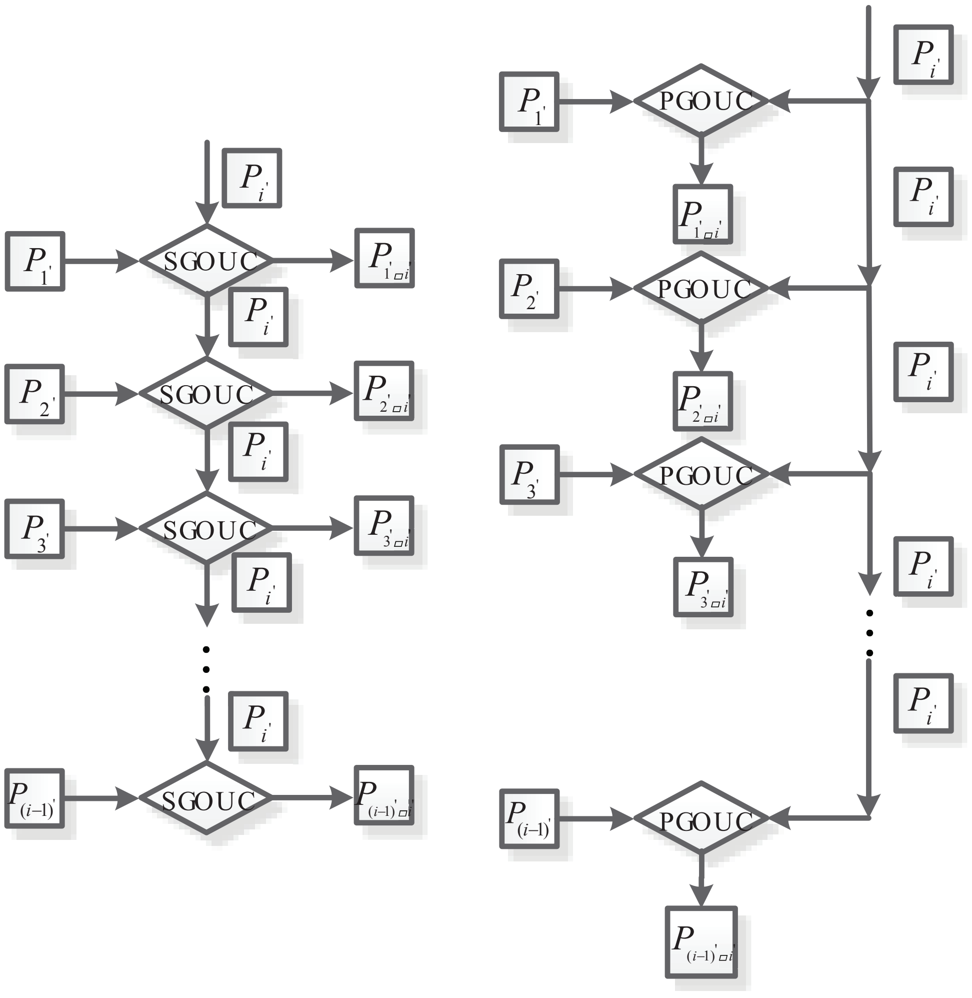


Figure 2.10 Comparisons between SGOUC and PGOUC.

Function 6 PGOUC($P'_1, P'_2, \cdots, P'_n$)
Input: n RGBA-formatted operands $P'_1, P'_2, \cdots, P'_n$ arriving in an arbitrary order
Output: a single RGBA-formatted blended operand P'

```
 1: ParaP_Pack $P\_P[n]$;
 2: Array $P'[4] = [0, 0, 0, 0]$;
 3: int $j = 0$;
 4: float $C\_\alpha = 1$;
 5: for ($j = 0$; $j \leq n - 1$; $j++$) do
 6:     Receive an operand $P_{j'}$;
 7:     $P\_P[j].D$= $P'_j(D)$;
 8:     $P\_P[j].\alpha$= $P'_j(\alpha)$;
 9:     $P\_P[j].C$=$P'_j(C)$;
10:     $C\_\alpha = C\_\alpha \cdot (1 - P'_j(\alpha))$;
11:     if ($j > 0$) then
12:         MIMDMULTI $\ll j \gg (P\_P, P_{j'}, j)$;
13: $P'[3] = 1 - C\_\alpha$;
14: MIMDWEIGHT$\ll n \gg (P\_P)$;
15: for ($r = 1$; $r \leq \lceil \log n \rceil$; $r++$) do
16:     MIMDSUM $\ll \lceil \frac{n}{2^r} \rceil \gg (P\_P, r, n)$;
17: $P'[0:2] = P\_P[0].C$;
18: return $P'$;
```

Function 7 MIMDMULTI($P\_P[], P_{j'}, j$)
Input: the data structures $P\_P[]$ of all exiting operands, the newly available operand $P_{j'}$, the number of available operands j
Output: $P\_P[]$ storing the operand's updated weight factor related to $P_{j'}$

```
1: int $i = blockIdx.x$; //obtain the thread ID in CUDA
2: if ($P\_P[i].D > P'_j(D)$) then
3:     $P\_P[i].W = P\_P[i].W \cdot (1 - P'_j(\alpha))$;
4: else
5:     if ($P\_P[i].D < P'_j(D)$ and $P\_P[P\_P[i].pNext].D > P'_j(D)$) then
6:         $P\_P[j].pNext = P_i(pNext)$;
7:         $P\_P[j].W = (1 - P\_P[i].\alpha) \cdot P\_P[i].W$;
8:         $P\_P[i].pNext = j$;
```

Function 8 MIMDWEIGHT($P\_P[\,]$)
Input: the data structures $P\_P[\,]$ of all existing operands
Output: $P\_P[\,]$ storing the operand's final weight factor

```
1: int $m = blockIdx.x$; //obtain the thread ID in CUDA
2: $P\_P[m].C = P\_P[m].C \cdot P\_P[m].W \cdot P\_P[m].\alpha$;
```

Function 9 MIMDSUM($P\_P[\,], r, n$)
Input: the data structures $P\_P[\,]$ of all existing operands, the current iteration r for summation, the total number n of operands
Output: $P\_P[\,]$ storing the operand's intermediate summation result

```
1: int $m = blockIdx.x$; //obtain the thread ID in CUDA
2: int $k = \lfloor \log_2 n \rfloor$;
3: int $interval = 2^{k-r}$;
4: if ($n \geq m + interval$) then
5:     $P\_P[m].C = P\_P[m].C + P\_P[m + interval].C$;
```

compositing, respectively, i.e.,

$$T_R = \sum_{i=1}^{n-1} t_i + t_{srt} + 4(n-1) \cdot t_c, \tag{2.40}$$

where t_{srt} denotes the time for sorting, t_c denotes the time for compositing a single channel. Since there are 4 channels in each operand, the time for compositing all n operands is $4(n-1) \cdot t_c$.

With the proposed *over* operator, there is no clear distinction between the waiting and composition stages, which often overlap with each other. To facilitate the analysis of such mixture, we construct a pipeline model that exactly maps the waiting stage and the composition stage to two continuous steps of the pipeline. The time cost for the waiting step varies, depending on the operands' arriving intervals, i.e., $t_1, \cdots, t_i, \cdots, t_{n-1}$. In the composition step of each incoming operand, we create a separate thread for every existing operand. The time cost for the second step is determined by the thread, which composites with the existing immediate upstream (preceding) operand of the current operand, and hence takes the longest to complete among all.

The threads in PGOUC are comprised of the same set of operations executed in possibly different orders. We thus introduce t_{max} to uniformly represent the time cost of the second step in all the running instances. The time cost for the entire composition process using the proposed *over* operator is

$$T_R = \sum_{i=1}^{n-1} \max(t_i, t_{max}) + t_{sum}, \tag{2.41}$$

where $\sum_{i=1}^{n-1} \max(t_i, t_{max})$ is the time cost of the pipeline, and t_{sum} denotes the time for the final summation.

Comparing (2.40) with (2.41), since in general $t_i > t_{max}$, it is straightforward to see the performance advantage of the proposed operator over the traditional one.

2.4.5 Implementation and Experimental Results

For a high level of parallelism, we implement the new *over* operator in a heterogeneous computing environment comprised of both CPUs and GPUs. We use two languages to implement Algorithm 23, i.e., *C* for CPU programming and CUDA for GPU programming. To be more specific, lines 12, 14 and 16 in Algorithm 23 are parallel functions, which are executed with a large number of threads and are hence implemented in CUDA. The rest of the code is sequential and is hence implemented in *C*. Since there is no global memory addressing between CPUs and GPUs in the existing heterogeneous computing environment, explicit data transfer is needed to support communications between them, which inevitably incurs an overhead. In practice, we overlap such data transfer with other activities to minimize the cost.

To test and evaluate the proposed *over* operator, we apply it to the image composition in parallel visualization of volume datasets of 3D Human Brain, High-Potential Iron Protein (HIPIP), and Jet Ejections using ray casting. We plot the final images composited by the new *over* operator in Figures 2.11, 2.12 and 2.13, respectively, which illustrate the validity of the proposed operator and the correctness of our implementation.

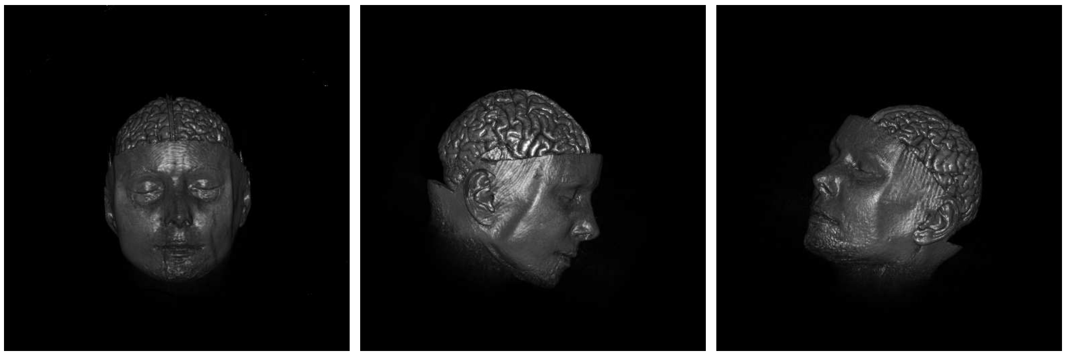


Figure 2.11 Composited images of Human Brain.

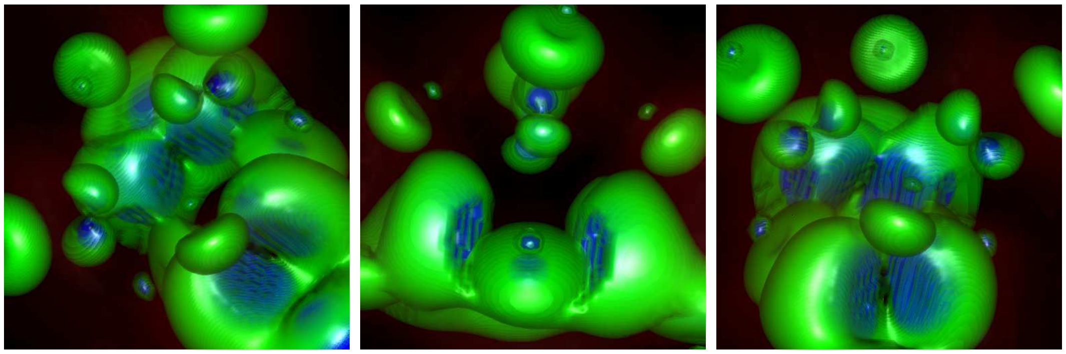


Figure 2.12 Composited images of HIPIP Surfaces.

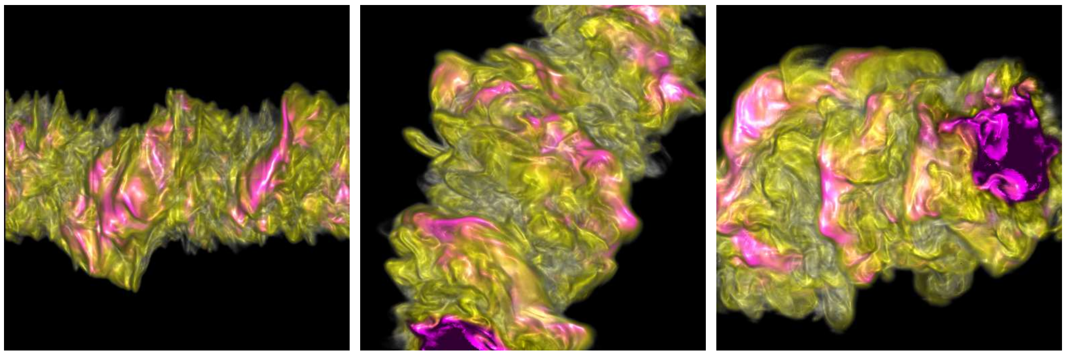


Figure 2.13 Composited images of Jet Ejections.

For a practical performance evaluation of these two operators, we use the image composition task in Figure 2.11 as a benchmark with a sequence of partial images of the same size without any blank pixels. For a comprehensive comparison, in the image composition experiments, we consider 2 image sizes: 2048^2 and 3072^2; 3 different numbers of input images: 8, 16, and 32; 6 image arriving intervals: 0s, 0.1s, 0.2s, 0.3s, 0.4s, and 0.5s; and 7 different image arriving (i.e., availability) orders, in each sequence of input images.

We define the k-distance arriving order as an arriving order where the difference between the blending orders of any two subsequently arriving input images is k. In the experiments, we generate 7 different arriving orders as follows. Given n operands (i.e., partial images), at time t, when $k \neq 2$, their arriving order is generated as

$$F(k,t,n) = \begin{cases} (t \bmod \lceil \frac{n}{k} \rceil) \cdot k + \lfloor t / \lceil \frac{n}{k} \rceil \rfloor \\ \text{if } 0 \leq t < (n \bmod k) \cdot \lceil \frac{n}{k} \rceil; \\ (t \bmod \lfloor \frac{n}{k} \rfloor) \cdot k + n - 1 - \lfloor (n-1-t) / \lfloor \frac{n}{k} \rfloor \rfloor \\ \text{if } (n \bmod k) \cdot \lceil \frac{n}{k} \rceil \leq t < n; \end{cases} \tag{2.42}$$

where $k \in [1,3,4,5,6,7]$, $n \in [8,16,32]$, and $t \in [0,n)$. When $k = 2$, the arriving order of the operands is generated as

$$F(k,t,n) = \begin{cases} (t \bmod \lceil \frac{n}{k} \rceil) \cdot k + 1 \\ \text{if } 0 \leq t < (n \bmod k) \cdot \lceil \frac{n}{k} \rceil, \\ (t \bmod \lfloor \frac{n}{k} \rfloor) \cdot k \\ \text{if } (n \bmod k) \cdot \lceil \frac{n}{k} \rceil \leq t < n. \end{cases} \tag{2.43}$$

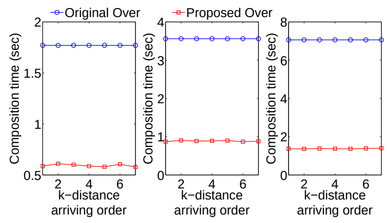
We plot the experimental results in Figure 2.14, which show that given the same arriving interval and the same number of input images of the same size, but different arriving orders, both algorithms exhibit relatively stable performance curves, which imply that both of the operators are immune to the arriving order. For the traditional operator, since it must wait for all the operands to become available, the arriving order does not affect the time cost; while for the proposed operator, whichever operand arrives, the composition is always performed in constant time in parallel, hence resulting in the identical total composition time.

However, the proposed operator takes consistently much less time than the original one in all the cases, which confirms our theoretical analysis. We further observe that the performance differences become more obvious as the image size and the number of input images increase, as shown in each subfigure (from a to h) of Figure 2.14.

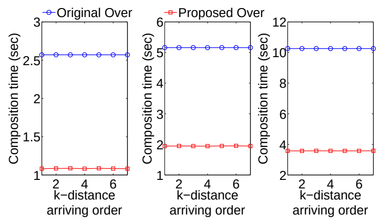
Furthermore, in Figure 2.14i and Figure 2.14j, as the arrival interval increases, the proposed operator's relative performance gain over the original one remains quite stable, i.e., the gap between two performance curves does not vary much, which is justified by (2.40) and(2.41). For the proposed operator, the time to perform all the computations for a newly arriving operand is constant. As the arriving interval increases, such computing time becomes negligible, (2.41) is dominated by the other two time cost components, i.e.,

$$T_R = \sum_{i=1}^{n-1} t_i + t_{sum}. \tag{2.44}$$

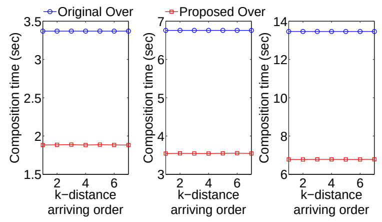
Comparing (2.44) with (2.40), the difference only lies in t_{sum} and $t_{srt} + 4(n-1) \cdot t_c$, which are all fixed given the same number of input images and the same image size. Thus, the performance difference between two these operators as the arriving interval increases is essentially the difference of these two terms.



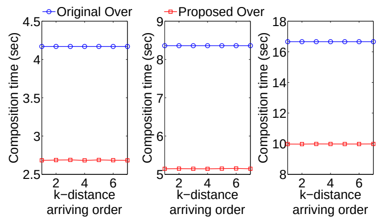
(a) arriving interval 0s for image size of 2048^2.



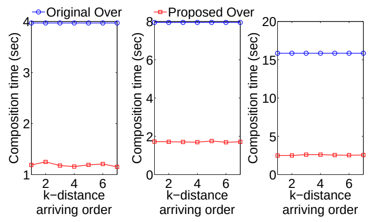
(b) arriving interval 0.1s for image size of 2048^2.



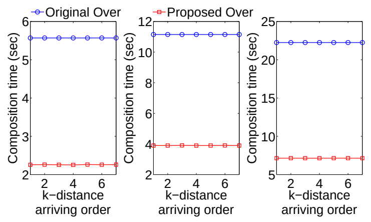
(c) arriving interval 0.2s for image size of 2048^2.



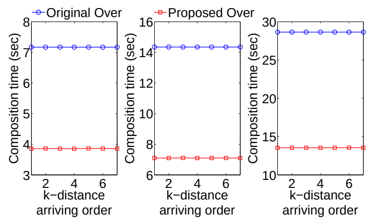
(d) arriving interval 0.3s for image size of 2048^2.



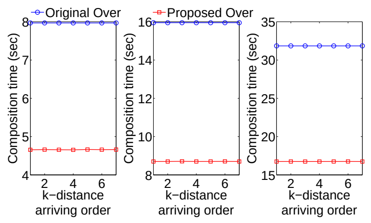
(e) arriving interval 0s for image size of 3072^2.



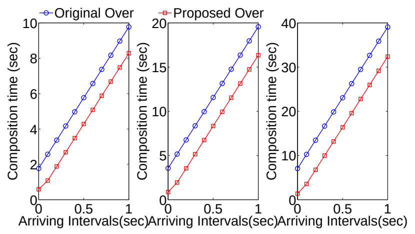
(f) arriving interval 0.2s for image size of 3072^2.



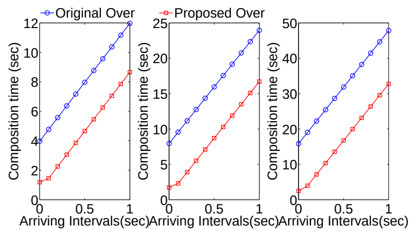
(g) arriving interval 0.4s for image size of 3072^2.



(h) arriving interval 0.5s for image size of 3072^2.



(i) Image size of 2048^2.



(j) Image size of 3072^2.

Figure 2.14 Comparisons of two operators on the image size of 2048^2 and 3072^2 with varying arriving intervals and numbers of input images using the 3D brain dataset. The three subfigures in each group correspond to the cases of 8, 16, and 32 input images, respectively, from left to right.

2.5 The In-practice Real-time *over* Operator

The generalized *over* (GOO) operator simply considers composition at a single pixel position and introduces multiple additional elements to support the desired extent of parallelization. When it comes to the composition of multiple partial images, where the number of involved pixel positions increases dramatically, treating each position independently and then applying GOO directly is a straight-forward but efficiency-disadvantageous method. To best suit the new composition situation, we propose the following optimizations oriented to GOO and its relevant structure ParaP_Pack{} : 1. correlating all fragments in the same partial image to a common depth value,"PID", which is minimal of all "D" values related to the partial image, instead of storing depth component "D" for each fragment in the image. Validity of the common "D" value is "D" justifiable according to assumptions, its specific value is determined during rendering. 2. breaking ParaP_Pack{} down into 3 finer-grained data structures FragCompC, FragCompAlpha, FragCompW, which correspond to "C[3]", "D","W" components in ParaP_Pack respectively. Such division provides more flexibility while exchanging data between CPU and GPU, enables the prioritization of needed data components, and facilitates a more efficient and complete usage of the limited GPU memory resources. To distinguish the *over* operator with the above optimizations, we denote it as IROO (in-practice real-time *over* operator).

2.5.1 IROO Based Image Composition and the General Optimization Strategies

The image composition problem considered here contains n $h \times k$-sized asynchronously-arriving partial images, m homogeneously-configured composition devices c_j, $j = 0, 1, \cdots, m-1$. Each partial image is divided into m identically-sized tiles t_j, $j = 0, 1, \cdots, m-1$. The fragment (x, y), $0 \leq x < w$, $0 \leq y < h$, belongs to tile t_j, if $j = \lfloor y/n \rfloor$. The composition unit c_j, $0 \leq j < n$, only composites fragments belonging to tile t_j.

To be more specific, each considered composition unit here provides a heterogeneous computing enviroment that contains both GPU and CPU. GPU performs most of the parallelization works and CPU coordinates the communications.[2] We study several performance-gaining strategies within such enviroment, which include 1) reducing data movement between CPU and GPU, 2) increasing data access rates, 3) modeling the amounts of requisite resources on GPU (mainly the global memory and threads, which are critical to the operator's parallelization), 4) exploring the parallelization opportunities with the given amounts of resources. We first address each of them individually, then apply and integrate them in proposed algorithms.

1) Data movement between host memory and device memory occurs when data on one side is requested from the other. Before desired data arrive, the operations depending on the requested data are halted. Such movement halts proceeding of all dependent operations and introduces non-negligible performance degrading. Our main optimization principle is thus to best locate the requested data along with the requesting side, i.e. locating requested data for GPU/CPU in device/host memory. In the current form of the proposed operators, operations on GPU require the W component of each participating fragment. Considering the W components are initialized uniformly for all fragments, we initialize all W components on device directly instead of doing that on host and later moving from host to device.

2) We consider increasing data access rates for GPU related operations through exploring the usage of share memory on GPU. Considering the scarcity of share memory and its preference over certain access patterns to deliver highest access rates, we accordingly propose the following acceleration strategies: 1. reserving the share memory for newly-arriving image's alpha values, which are highly concurrently demanded while updating all existing images' W components; 2. swapping data in/out of share memory at a size equivalent to the share memory's capacity so as to

[2]the GPU device we select are CUDA compatible, the terms follow CUDA conventions naturally.

minimize the relevant data movement overheads, i.e., all required alpha components are splited into n parts, where n is the number of locations the share memory exist, one alpha component-part corresponding to one share memory-location; 3. coordinating the each thread's share-memory access scheme so as to minimize bank confliction.

3) In our real-time composition problem, as new image arrives, the cost of memory to keep the W components and alpha values of all available operands on board also increases. It is possible that the required amounts in such a manner exceed the available at a particular point. We figure out the feasibility of avoiding/delaying such a point based on the observation that the requisite alpha values as new image arrives come from two images at most: one is the newly arriving one, the other is immediately in front of the arriving one among the currently available ones, leaving all the rest eligible for being replaced without affecting the performance. In addition, the immediately in-front image has less priority than the new one, the memory requirement of new image should be always considered first. When the memory shortage occurs, we further propose to substitute the alpha values of existing partial images with the W component and alpha values of the newly arriving one. Methods to determine the shortage and the corresponding treatment are given in the modeling and algorithm design, respectively.

In compliance with the problem specification and proposed optimization, we propose the following modeling for the proposed operator's resource requirement, covering both memory and thread. For memory requirement, there exist the following metrics: 1) a constant size C to hold the permanent variables which live through the whole composition procedure; 2) a varying size $D(j)$ to support the update of all j existing images' W components in parallel, which requires the simultaneous availability of these images' all related W components and at least two images' all related alpha values, thus determines that $D(j) = (j + 2) \times h \times k$, where $h \times k$ is the size of each considered image, $j \times h \times k$ is allocated to W components of all considered images, $2 \times h \times k$ is for alpha values of the newly arriving image and its immediately-before one; 3) $D(j)$ is the size of memories actually accessed in the parallel execution, reserving memory space merely of size $D(j)$ doesn't guarantee optimal performance, since the new image's immediately-before image is determined only after the new image is ready, it requires another round of data transferring if the needed immediately-before image is not in device memory yet, instead, if all existing images' alpha components are already in memory when the new image arrives, there is no such follow-up transferring need, the required amounts of memory in such situation is $(j + 2) \times h \times k$ then. Given available amounts of memories m_a, we categorize the according composition of the j images as: 1) sufficient memory(SM) if $m_a > 2j \times h \times k$; 2) acceptable memory(AM) if $(j+2) \times h \times k < m_a < 2j \times h \times k$; 3) deficient memory(DM) if $m_a < (j+2) \times h \times k$.

For thread requirement, we follow the similar logics. The requisite number of threads for completely parallel composition of the j images is $j \times h \times k$, which tells the thread sufficiency in the considered situation as 1)insufficient threads(IT) 2) sufficient threads(ST).

The above analysis tells the extent of possible parallelism in the given situation, points out the issues to consider while exploring parallelization, and provides a concise framework to organise the according algorithm design. We thus consider our algorithm design based on the above categorization of memory availability and cover the divided cases of 1. SM 2. AM 3. DM individually.

2.5.2 IROO Based Image Composition Algorithms

SM With the ideal availability of memory resources, we consider improving the performance from the perspectives of increasing memory access-rates and best utilizing the given thread pool individually.

To best utilize the limited high-speed memories, and thus increase overall access rates, in each step of the composition, we devote such memories to the most commonly, concurrently requested data, i.e., the alpha values of the newly available image. Considering the shortage of such resource as well as its split into multiple same-sized units on the device, we also divide all alpha values of the new image into the same number of blocks, assign one block to one high-speed unit, and swap data in/out of the unit to address the deficiencies. The specific division and assignment logics are given in Figure 2.17 for better understanding. Another restriction related to access of such high-speed memory is that only threads in the same thread-block can access the same high-speed memory unit. We bear such restriction in mind while specifying each thread's job assignment. A proper algorithm integrating the above considerations is given in Algorithm 10.

Given partial image tiles, $PIT_0, \cdots, PIT_i, \cdots, PIT_{n-1}$ as input, the functionality body of Algorithm 10 consists of the following parts: 1) variable declaration, from lines 1 to 6, where requisite variables are introduced and briefly illustrated with comments; 2) weight updating as each image arrives, from lines 7 to 12, which involves sending the received image to GPU and launching the kernal there for parallelization of related operations; 3) weight finalizing after all images are available at line 12, i.e., executing the kernel "FinalAlpha" on GPU; 4) weight summarizing for desired results, from lines 13 to 14, i.e., launching rounds of kernel "IROOSUM" before the desired image is ready to be sent back to CPU. Among the multiple kernels introduced in the algorithm, "FinalAlpha" and "IROOSUM" in part 3) and 4) are straight-forward extensions from the single-operator version, thus skipped for brevity. We focus on the proposed two kernels for the weight updating in part 2), "CacheUPD_SM", utilizing the high-speed cache for increasing access rates, and "ThreadUPD_SM", fully utilizing available thread resources for higher degree of parallelization.

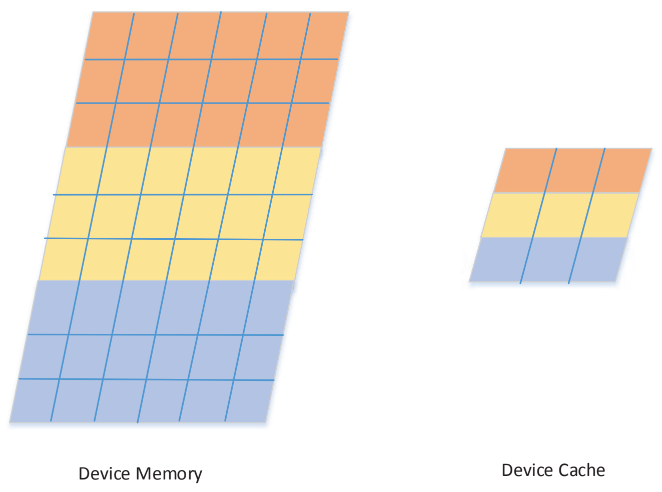


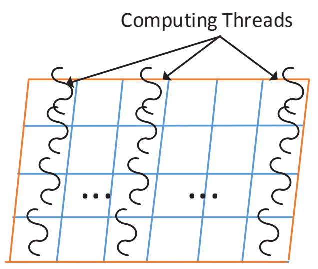
Figure 2.15 Illustration for the correlation between the global memory and high-speed cache.

For kernel "CacheUPD_SM", its functionality body consists of the following 3 parts, 1) variable declaration, lines 1 to 6; 2) determination of the immediately-before and immediately-behind images for the new one, lines 7 to 11; 3) updating the W components of the new image and all others that are behind the new one, lines 12 to 25. Logics and validity for part 1) and 2) are straight forward, we focus on the illustration to part 3). The update procedure in part 3) is generally in the form of an l-iteration loop, where l is the ratio between the size of each input image, i.e., $h \cdot w$, and that of the high-speed memory, $BufferSize$. At the beginning of i-th iteration, all threads wait till the high speed cache is fulfilled with the new image's i-th alpha component block, each of which is of the same size, $BufferSize$, then diverge on whether the images they are assigned to fall before and behind the new image: threads assigned to the behind images update all its W components in parallel; threads assigned to immediately-before images update the W component of the new image in parallel; other threads idle .

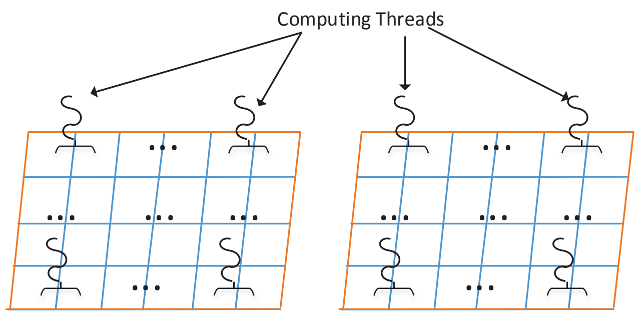
For kernel "ThreadUPD_SM", it is similar to the "CacheUPD_SM" on the aspects of first two parts, we thus focus only on their differences on specification for the update loop. The proposed update loop here consists of l iterations, where l is the number of positions each thread needs to deal with. Given iteration i of the l ones, each thread tells whether to update the considered position based on the same depth info that is utilized in “CacheUPD_SM”. The kernel finishes when the loop is done. An illustrating figure for each thread’s assignment of pixel positions is given in Figure 2.16.

2.5.3 AM

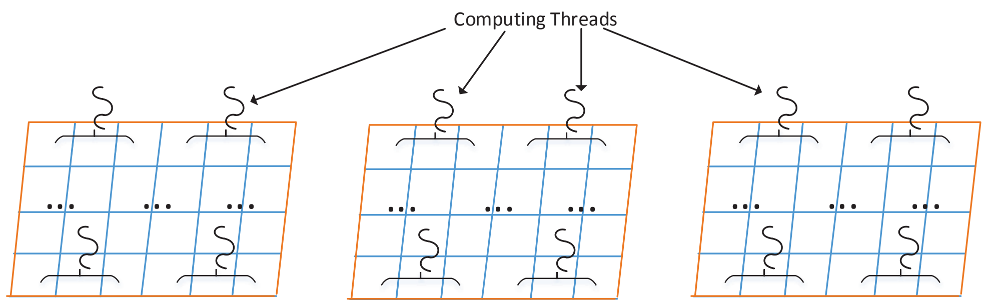
With barely enough memory resources, proper memory management is also required to secure the procedure’s overall performance. We incorporate the proposed optimizations in Section 2.5.1 to speed up the composition in this new situation. According solution integrating the above is given in Algorithm 15. Functionality body of Algorithm 15 is divided into the same number of parts as Algorithm 10, 1) variable declaration, lines 1 to 8; 2) weight updating as each image arrives, from lines 9 to 13; 3)weight finalizing after all images available, lines 14 to 18; 4) weight summarizing for desired image, lines 19 to 20. Each divided part shares a common goal with its counterpart in Algorithm 10 and adopts new strategies to cope with the difference. We thus focus only on new strategies introduced in each part. Launched kernels, “CacheUPD_AM” and “ThreadUPD_AM”, in Part 2) consider the possible unavailability of the immediately before-image’s alpha components while updating the new image’s “W” components and address the problem by checking the variable $AlphaTag$, which indicates the index of the image whose *alpha* values are currently kept in device memory: if the value of $AlphaTag$ points to the desired immediately in-front image, the W components of the new image are updated directly, otherwise, updating of such components are delayed until *alpha* components of the desired image are transferred to device memory. Weight Finalizing in Part 3) operates in two parallel streams to fully utilize the allocated space that can only accommodate alpha



(a) Available number of threads is equal to number of positions.



(b) Available number of threads is half of number of positions.



(c) Available number of threads is equal to one third of positions.

Figure 2.16 Illustrating figure for each thread's assignment of pixel positions.

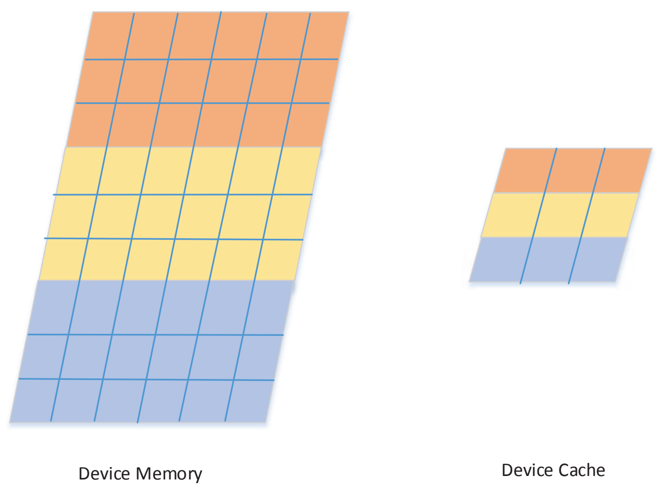


Figure 2.17 Illustration for the correlation between the global memory and high-speed cache.

components of two images, to be more specific, all odd-numbered images are assigned to one stream, all even-numbered images are assigned to the other stream. Images in each stream go through the process of transferring *alpha* components to GPU and finalizing them by “FinalAlpha”. Weight summarizing in Part 4) has no difference with its counterparts in “CacheUPD_SM” or “ThreadUPD_SM”, is thus skipped.

2.5.4 DM

As the memory resource falls out of the self-sufficiency range, we consider the strategies of frequently swapping data between the host and device memory and other underlying possibilities to minimize the related overheads. The accordingly proposed algorithm is given in Algorithm 18. Its functionality body is divided into the same parts : 1) variable declaration; 2) weight updating as each image arrives; 3) weight finalizing after all images are available; 4) weight summarizing for desired image. We also focus only on new strategies oriented to the new problem setting.

Weight updating in Part 1) operates in d parallel streams, where d is the maximal number of images the selected GPU can support for keeping their relevant *alpha* values within device memory. The image PIT_j is attributed to the stream

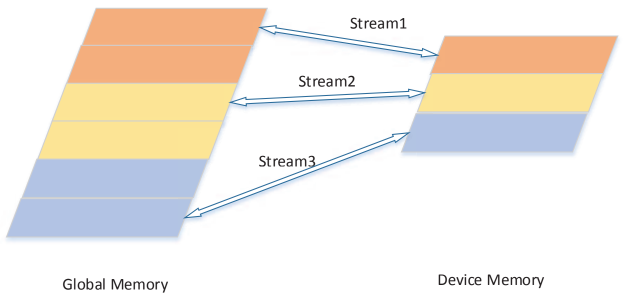


Figure 2.18 Illustration of the assignment of images to streams.

$StrmNum(0 \leq StrmNum < d)$ if and only if $StrmNum == (j/\lfloor n/d \rfloor)$. As PIT_i arrives, its *alpha* values are transferred to GPU first, then images in each stream go through the process of weight updating by "ThreadUPD", transferring back updated weight back to host one after another. Illustration for the assignment of images to streams is given in Figure 2.18.

Weight finalizing in Part 2) operates in maximal $\lfloor \frac{d}{2} \rfloor$ parallel streams, considering that the required space for each image is $h \cdot w \cdot 2$ at this part, one half for alpha components, the other for W components. The image PIT_j is assigned to the stream $StrmNum(0 \leq StrmNum < \lfloor \frac{d}{2} \rfloor)$ if and only if $StrmNum == (j/\lfloor n/\lfloor \frac{d}{2} \rfloor \rfloor$. Images in each stream go through the process of transferring *alpha* components, W components to GPU, weight finalizing by "FinalAlpha" and transferring back finalized weight to host one image after another.

Weight summarization in Part 3) consists of two subparts. Subpart 1, lines 22 to 30, operates in maximal $\lfloor \frac{d}{2} \rfloor$ parallel streams, since the required space in each stream is also $h \cdot w \cdot 2$, one half for *alpha* components of the fixed image, the other for *alpha* components of a varying image. The image PIT_j is assigned to the stream $StrmNum(0 \leq StrmNum < \lfloor \frac{d}{2} \rfloor)$ if and only if $StrmNum == (j/\lfloor n/\lfloor \frac{d}{2} \rfloor \rfloor)$. The least-numbered image in each stream is selected as the fixed image. At the beginning of each stream, finalized weights of the fixed image are transferred to GPU first, then rest images in the same stream follow the steps of sending their W components to GPU, summing up with the fixed image one after another. Subpart 2, lines 31 and 32, finishes summing up the last $\lfloor \frac{d}{2} \rfloor$ images similarly as IROO-SM, IROO-AM did.

Function 10 IROO-SM($PIT_0, \cdots, PIT_i, \cdots, PIT_{n-1}$)
Input: the data structures $PIT_i (0 \leq i < n)$ which contains the rendered tile from c_i and the corresponding PID_i value
Output: Composited Tile

```
 1: __constant__ $DInfo[n]$; // Depth value for each input image
 2: __constant__ $NextInfo[n]$;//Index of immediately behind-peer for each input image
 3: int $i = 0$;
 4: $FinalImg$=Malloc($h \cdot w$);//CPU memory for composited final images
 5: $FragW$=Cudamalloc($h \cdot w \cdot n$);//dedicated GPU memory for W compoment of all input
    images
 6: $FragAlpha$=Cudamalloc($h \cdot w \cdot n$);//dedicated GPU memory for Alpha compoment of
    all input images
 7: while (Receive $PIT_i$) do
 8:    $DInfo[i]$=$PIT_i.PID_i$;
 9:    $FragAlpha[h \cdot w \cdot i]$=Host2DeviceCpy($PIT_i.Tile$);
10:    ThreadUPD_SM$\ll 1, \{BuffSize, i+1\} \gg$ ( $FragW$, $FragAlpha$, $DInfo$, $NextInfo$, $h$, $w$, $AlphaTag$);
11:    $i++$;
12: FinalAlpha$\ll 1, \{h, w, min(n, \frac{ThDNum}{h \cdot w})\} \gg (FragW, FragAlpha)$;
13: for ($r = 1$; $r \leq \lceil \log n \rceil$; $r++$) do
14:    IROOSUM$\ll \lceil \frac{n}{2^r} \rceil, \{h, w\} \gg (FragW, 0)$;
15: $FinalImg$=Device2HostCpy($FragW$);
16: return $FinalImg$
```

Function 11 CacheUPD_SM($FragW, FragAlpha, DInfo, NextInfo, h, w$)
Input: $FragW$, array of W component; $FragAlpha$, array of $alpha$ components; $DInfo$, array of depth info; $NextInfo$, array of indexes for immediately-next image; h, image height; w, image width
Output: Updated $FragW$

```
int $tx = threadIdx.x$;
int $BuffSize = blockDim.x$;
int $k = blockDim.y - 1$;
int $l = h \cdot w / BuffSize$;
int $ty = threadIdx.y$;
int $PreImg = 0$;
**if** $tx == 0$ **then**
    **if** ($DInfo[ty] < DInfo[k]$ and $DInfo[NextInfo[ty]] > DInfo[k]$ ) **then**
        $NextInfo[k] = NextInfo[ty]$;
        $NextInfo[ty] = k$;
        $PreImg = ty$;
**for** $i = 0$ to $l - 1$ **do**
    __shared__ FragCompAlpha $ShareFrag[BuffSize]$;
    **if** $ty == 0$ **then**
        $ShareFrag[tx] = FragAlpha[h \cdot w \cdot k + i \cdot BuffSize + tx]$;
    __syncthreads();
    **if** ($DInfo[ty] > DInfo[k]$) **then**
        $Windex = h \cdot w \cdot ty + i \cdot BuffSize + tx$;
        $FragW[Windex] = FragW[Windex] \cdot (1 - ShareFrag[tx])$;
    **else**
        **if** ($PreImg == ty$ ) **then**
            $Windex = h \cdot w \cdot k + i \cdot BuffSize + tx$;
            $PreWindex = h \cdot w \cdot ty + i \cdot BuffSize + tx$;
            $Aindex = PreWindex$;
            $FragW[Windex] \cdot = FragW[PreWindex] \cdot (1 - FragAlpha[Aindex])$;
```

Function 12 ThreadUPD_SM($FragW, FragAlpha, DInfo, NextInfo, w$)
Input:$FragW$, array of W component; $FragAlpha$, array of $alpha$ components; $DInfo$, array of depth info; $NextInfo$, array of indexes for immediately-next image; w, image width
Output Updated $FragW$

```
1: int $tx = threadIdx.x$;
2: int $ty = threadIdx.y$;
3: int $tz = threadIdx.z$;
4: int $k = blockDim.x$;
5: int $h = blockDim.y$;
6: int $l = w/blockDim.z$;
7: int $PreImg = 0$;
8: if $tx == 0$ then
9:    if ($DInfo[ty] < DInfo[k]$ and $DInfo[NextInfo[ty]] > DInfo[k]$ ) then
10:      $NextInfo[k] = NextInfo[ty]$;
11:      $NextInfo[ty] = k$;
12:      $PreImg = ty$;
13: for (int $i = 0; i < l; i++$) do
14:    if ($DInfo[tx] > DInfo[k]$) then
15:      $Windex = h \cdot w \cdot tx + w \cdot ty + tz \cdot l + i$;
16:      $Aindex = h \cdot w \cdot k + w \cdot ty + tz \cdot l + i$;
17:      $FragW[Windex] = FragW[Windex] \cdot (1 - FragAlpha[Aindex])$;
18:    else
19:      if ($ty == PreImg$ ) then
20:         $Windex = h \cdot w \cdot k + w \cdot ty + tz \cdot l + i$;
21:         $PreWindex = h \cdot w \cdot tx + w \cdot ty + tz \cdot l + i$;
22:         $Aindex = PreWindex$;
23:         $FragW[Windex] \cdot = FragW[PreWindex] \cdot (1 - FragAlpha[Aindex])$;
```

Function 13 FinalAlpha($FragW, FragAlpha$)
Input: $FragW$, array of W component; $FragAlpha$, array of $alpha$ components
Output Finalized $FragW$

```
1: int $tx = threadIdx.x$;
2: int $ty = threadIdx.y$;
3: int $tz = threadIdx.z$;
4: int $h = blockDim.x$;
5: int $w = blockDim.y$;
6: $WPos = tz \cdot h \cdot w + ty \cdot w + tx$;
7: $Frag[WPos] = Frag[WPos] \cdot FragAlpha[WPos]$;
```

Function 14 IROOSUM($FragW, SumNum$)
Input: $FragW$, array of W component; $SumNum$, finished rounds of summation
Output: $FragW$ storing the operand's intermediate summation result

```
1: int $tx = threadIdx.x$;
2: int $ty = threadIdx.y$;
3: int $tz = threadIdx.z$;
4: int $interval = SumNum/blockDim.z$;
5: int $h = blockDim.x$;
6: int $w = blockDim.y$;
7: $WPos = tz \cdot h \cdot w \cdot interval + ty \cdot w + tx$;
8: $PairWPos = (2 \cdot tz + 1) \cdot (interval/2) \cdot h \cdot w + ty \cdot w + tx$;
9: $FragW[WPos] = FragW[WPos] + FragW[PairWPos]$;
```

Function 15 IROO-AM($PIT_0, \cdots, PIT_j, \cdots, PIT_{n-1}$)
Input: the data structures $PIT_j (0 \leq j < n)$ which contains the rendered tile from c_i and the corresponding PID_i value
Output: Composited Tile

```
1: __constant__ $DInfo[n]$;
2: __constant__ $NextInfo[n]$;
3: int $k = 0$;
4: int $AlphaTag = -1$;
5: cudaStream_t $streams[2]$;
6: $FragW$=Cudamalloc($h \cdot w \cdot n$);
7: $FragAlpha$=Cudamalloc($h \cdot w \cdot 2$);
8: $HostAlpha$=Hostmalloc($h \cdot w \cdot n$);
9: while (Receive $PIT_k$) do
10:     $DInfo[k] = PIT_k.PID_k$;
11:     $FragAlpha[0]$=Host2DeviceCpy($PIT_k.Tile$);
12:     ThreadUPD$\ll 1, \{k+1, h, 1\} \gg$($FragW, FragAlpha, DInfo, NextInfo, w,$);
13:     $k++$;
14: while $k > 0$ do
15:     $StrmNum = k \mod 2$;
16:     $FragAlpha[h{\cdot}w{\cdot}StrmNum]$=AyncHost2DeviceCpy($PIT_k.Tile, streams[StrmNum]$);
17:     FinalAlpha$\ll 1, \{h, w, 1\}, streams[StrmNum] \gg (FragW[h \cdot w \cdot k], FragAlpha[h \cdot w \cdot StrmNum])$;
18:     $k--$;
19: for ($r = 1$; $r \leq \lceil \log n \rceil$; $r++$) do
20:     $IROOSUM \ll \lceil \frac{n}{2^r} \rceil, \{h, w\} \gg (FragW, 0)$;
21: $FinalImg$=Device2HostCpy($FragW$);
22: return $FinalImg$
```

Function 16 CacheUPD_AM($FragW, FragAlpha, DInfo, NextInfo, h, w$)
Input: $FragW, FragAlpha$, array of W and $alpha$ components; $DInfo$, array of depth info; $NextInfo$, array of indexes for immediately-next image; h, w, image height and width;
Output: Updated $FragW$

```
1: int $tx = threadIdx.x, ty = threadIdx.y;$ ;
2: int $BuffSize = blockDim.x, PreImg = 0;$
3: int $k = blockDim.y - 1, l = h \cdot w / BuffSize;$
4: **if** $tx == 0$ and $DInfo[ty] < DInfo[k]$ and $DInfo[NextInfo[ty]] > DInfo[k]$ ) **then**
5:    $NextInfo[k] = NextInfo[ty];$
6:    $NextInfo[ty] = k, PreImg = ty;$
7: **for** $i = 0$ to $l - 1$ **do**
8:    __shared__ FragCompAlpha $ShareFrag[BuffSize];$
9:    **if** $ty == 0$ **then**
10:       $ShareFrag[tx] = FragAlpha[h \cdot w \cdot k + i \cdot BuffSize + tx];$
11:    __syncthreads();
12:    **if** $(DInfo[ty] > DInfo[k])$ **then**
13:       $Windex = h \cdot w \cdot ty + i \cdot BuffSize + tx;$
14:       $FragW[Windex] = FragW[Windex] \cdot (1 - ShareFrag[tx]);$
15:    **else**
16:       **if** $(PreImg == ty$ ) **then**
17:          $Windex = h \cdot w \cdot k + i \cdot BuffSize + tx;$
18:          $PreWindex = h \cdot w \cdot ty + i \cdot BuffSize + tx;$
19:          **if** $AlphaTag! = ty$ **then**
20:             **if** $i == 0$ and $tx == 0$ **then**
21:                $FragAlpha[h \cdot w]$=Host2DeviceCpy($PIT_{ty}.Tile$);
22:                $AlphaTag = ty;$
23:             __syncthreads();
24:          **else**
25:             $Aindex = h \cdot w + i \cdot BuffSize + tx;$
26:             $FragW[Windex] \cdot = FragW[PreWindex] \cdot (1 - FragAlpha[Aindex]);$
```

Function 17 ThreadUPD_AM($FragW, FragAlpha, DInfo, NextInfo, w$)
Input:$FragW$, $FragAlpha$, array of W and $alpha$ components; $DInfo$, array of depth info; $NextInfo$, array of indexes for immediately-next image; w, image width;
Output: Updated $FragW$

```
int $tx = threadIdx.x, ty = threadIdx.y$;
int $tz = threadIdx.z, PreImg = 0$;
int $k = blockDim.x, h = blockDim.y$;
int $l = w/blockDim.z$;
int $AlphaTag = -1$;
if $tx == 0$ and $DInfo[ty] < DInfo[k]$ and $DInfo[NextInfo[ty]] > DInfo[k]$ ) then
    $NextInfo[k] = NextInfo[ty]$;
    $NextInfo[ty] = k$;
    $PreImg = ty$;
for (int $i = 0; i < l; i++$) do
    if ($DInfo[tx] > DInfo[k]$) then
        $Windex = h \cdot w \cdot tx + w \cdot ty + tz \cdot l + i$;
        $Aindex = h \cdot w \cdot k + w \cdot ty + tz \cdot l + i$;
        $FragW[Windex] = FragW[Windex] \cdot (1 - FragAlpha[Aindex])$;
    else
        if ($ty == PreImg$ ) then
            $Windex = h \cdot w \cdot k + w \cdot ty + tz \cdot l + i$;
            $PreWindex = h \cdot w \cdot tx + w \cdot ty + tz \cdot l + i$;
            if $AlphaTag != ty$ then
                if $i == 0$ and $tx == 0$ then
                    $FragAlpha[h \cdot w]$=Host2DeviceCpy($PIT_{ty}.Tile$);
                    $AlphaTag = ty$;
                __syncthreads();
            else
                $Aindex = h \cdot w + i \cdot BuffSize + tx$;
                $FragW[Windex] \cdot = FragW[PreWindex] \cdot (1 - FragAlpha[Aindex])$;
```

Function 18 IROO-DM($PIT_0, \cdots, PIT_j, \cdots, PIT_{n-1}$)
Input: the data structures $PIT_i (0 \leq i < n)$ which contains the rendered tile from c_i and the corresponding PID_i value
Output: Composited Tile

```
 1: __constant__ $DInfo[n], NextInfo[n]$;
 2: int $i = 0$;
 3: $FragW$=Cudamalloc($h \cdot w \cdot d$);
 4: $HostW$=Malloc($h \cdot w \cdot n$);
 5: cudaStream_t $streams[d]$;
 6: $FragAlpha$=Cudamalloc($h \cdot w \cdot 2$);
 7: while (Receive $PIT_i$) do
 8:     $DInfo[i] = PIT_i.PID_i$;
 9:     int $StrmNum = i \mod d$;
10:     $FragAlpha[h \cdot w \cdot StrmNum]$=AyncHost2DeviceCpy($PIT_i.Tile, streams[StrmNum]$);
11:     ThreadUPD$\ll 1, \{1, h, w\}, streams[StrmNum] \gg$ ($FragW, FragAlpha, DInfo, NextInfo$);
12:     $HostW[h \cdot w \cdot i]$=AyncDevice2HostCpy($FragAlpha[h \cdot w \cdot StrmNum], streams[StrmNum]$);
13:     $i++$;
14: while $i > 0$ do
15:     $StrmNum = i \mod \lfloor \frac{d}{2} \rfloor$;
16:     $FragW[h \cdot w \cdot StrmNum \cdot 2]$=AyncHost2DeviceCpy($HostW[h \cdot w \cdot i], streams[StrmNum]$);
17:     $FragW[h \cdot w \cdot (StrmNum \cdot 2+1)]$=AyncHost2DeviceCpy($PIT_i.Tile, streams[StrmNum]$);
18:     $FinalAlpha \ll 1, \{h, w, 1\}, streams[StrmNum] \gg (FragW[h \cdot w \cdot StrmNum \cdot 2], FragW[h \cdot w \cdot (StrmNum \cdot 2 + 1)])$;
19:     $HostW[h \cdot w \cdot i]$=AyncDevice2HostCpy($FragW[h \cdot w \cdot StrmNum \cdot 2], streams[StrmNum]$);
20:     $i--$;
21: while $i < n$ do
22:     $StrmNum = (i \mod \lfloor \frac{d}{2} \rfloor)$;
23:     $TileNum = (i \mod \lfloor \frac{d}{2} \rfloor) \cdot (n/\frac{d}{2} \rfloor) + i/\lfloor \frac{d}{2} \rfloor$;
24:     if $i/\lfloor \frac{d}{2} \rfloor == 0$ then
25:         $FragW[h \cdot w \cdot StrmNum \cdot 2]$=AyncHost2DeviceCpy($HostW[h \cdot w \cdot TileNum]$, $streams[StrmNum]$);
26:     else
27:         $FragW[h \cdot w \cdot (StrmNum \cdot 2+1)]$=AyncHost2DeviceCpy($HostW[h \cdot w \cdot TileNum]$, $streams[StrmNum]$);
28:     $IROOSUM \ll 1, \{h, w\} \gg (FragW, StrmNum \cdot 2)$;
29:     $i--$;
30: for ($r = 1$; $r \leq log\lfloor \frac{d}{2} \rfloor$; $r++$) do
31:     $IROOSUM \ll \lceil \lfloor \frac{d}{2} \rfloor / 2^r \rceil, \{h, w\} \gg (FragW, 0)$;
32: $FinalImg$=Device2HostCpy($FragW$);
33: return $FinalImg$
```

CHAPTER 3

IMAGE COMPOSITION WORKFLOW ORGANIZATION

3.1 Image Composition Workflow Organization

There exist a wide range of research efforts in the aspect of image composition workflow scheduling. Among the most traditional ones, the Binary Swap method proposed by Ma *et al.* divides the entire procedure into several sub-stages according to the number of available processors [42]. At each sub-stage, each processor is paired with its counterpart according to its own processor label and the current sub-stage number. This method evenly distributes the composition workload among all the participating processors, and minimizes the number of sub-stages. The main issue associated with this method is its constraint on the number of processors being a power of 2.

Lee *et al.* proposed the Parallel Pipeline method, where all the processors are arranged in a 2-D grid and the entire composition procedure is divided into two sub-stages accordingly [37]. At each sub-stage, each processor joins a group of processors sharing one common coordinator in the established 2-D grid coordinate system, and communicates with others in the current group to blend its own pixels. Different from Binary Swap, this method removes the power-of-2 constraint on the number of processors. However, it introduces more communication cost than Binary Swap when the number of available processors happens to be a power of 2. Eilemann *et al.* proposed the classical Direct Send method, which can be viewed as a simplified version of the Parallel Pipeline method in 1-D [18].

Lin *et al.* proposed the Rotate Tiling (RT) method, which integrates the Binary Swap method with the Parallel Pipeline method [41]. By carefully dividing the rendered image of each processor into a certain number of blocks, Rotate Tiling reduces the communication overhead incurred in the entire procedure at the cost of unbalanced workload, which may negatively affect the overall visualization performance.

Yu *et al.* proposed the 2-3 swap composition method with the intention to generalize the original Binary Swap method [71]. In this generalized method, each processor is initially assigned a different number of pixels for composition. As the process continues, the difference in the number of pixels on each processor may become smaller, and eventually, it reaches a finish point when the number of pixels each processor needs to handle is equal and the pixels on all the processors constitute a complete image. This method removes the constraint on the number of processors, but each participating processor may incur more communications with others and some processors may be left idle in the process. In addition, they also proposed and analyzed several intuitive methods such as Reduced Binary Swap to extend the application scope of Binary Swap.

More recently, Peterka *et al.* proposed the Radix-k method, in which the number of processors can be factorized arbitrarily [55]. Once a certain factorization is selected, the composition procedure is scheduled according to the number of factors and the value of each factor involved in the factorization. This method does not impose any constraint on the number of processors, and has potential to achieve a good performance in terms of communication cost with a carefully selected factorization. However, since this method is intended for order-dependent problems, it raises an issue on how a proper factorization should be selected in the case of a dynamic image arriving order. In general, an exhaustive search of all possible factorizations might be needed for the best performance.

We proposed Grouping More and Pairing Less (GMPL) method, which provides a generalization framework that encompasses several existing algorithms. GMPL takes a prime factorization-based approach to strategically divide the processors into a set of groups at the finest possible grain and form a well-structured grid to minimize the communication overhead with evenly distributed workload. When the number of processors is a prime or a power of 2, GMPL reduces to an improved version of the Direct Send algorithm applied to all the processors or in each established group.

3.2 GMPL Algorithm for Image Composition Workflow Scheduling

We consider a general image composition problem in sort-last parallel rendering that involves N homogeneous processors, denoted as $W_0, W_1, \ldots, W_{N-1}$, which are interconnected through a local switch. Each processor has a locally rendered image of an identical size P without any blank pixels, and needs to blend its own pixels with those corresponding ones of the same position on all other processors using the Z-depth test, i.e., Z-buffer method. In this image composition problem, we consider a multi-port communication model such that each processor is able to send and receive messages simultaneously with an equal amount of bandwidth.

3.2.1 Algorithm Design

Based on a thorough investigation into the existing methods for image composition, we provide a summary of the observations and rationales in support of our algorithm design:

- Given N processors, i) if they are arranged in the way of Direct Send, the latency would proportionally relate to the number of processors [71]; ii) if they are arranged in a 2-D grid of $f_0^i \times f_1^i$, where $N = f_0^i \times f_1^i$ and $f_0^i, f_1^i \geq 1$, the latency can be reduced compared with Direct Send [37]. The difference between these two arrangements lies in the decision on whether or not to factorize the number of nodes. Peterka *et al.* further suggested that the N processors be arranged in a multidimensional grid based on the factors of N [55]. As factorization is conducive to the latency performance and can be conducted recursively, one may be able to achieve the highest performance gain by factorizing the number of nodes into prime numbers, i.e., to the lowest degree.

- If the number N of processors happens to be prime, a grid-based grouping approach through factorization does not work, and therefore these N nodes may have to be arranged as in Direct Send. As Direct Send generally incurs a high latency, more efficient methods are needed to improve its latency performance.

By integrating the above two aspects into algorithm design, we propose a Grouping More and Pairing Less (GMPL) method for order-independent image composition, whose pseudocode is provided in Algorithm 19, which calls PrimeFactor-Encoding() in Algorithm 20 and ImprovedDS() in Algorithm 21. The PrimeFactor-Encoding() function is used to calculate the code series of a given processor ID, which comprises of a sequence of coordinates in the hyperspace of the prime factors derived from the number of processors.

Note that the decision version of the prime or integer factorization problem is generally considered within the class of UP (Unambiguous Non-deterministic Polynomial-time) and outside the class of P [13]. However, there exist many special-purpose algorithms including trial division [23] and elliptic curves [32], and general-purpose algorithms including Dixon's factorization method [16] and continued fraction factorization [38], which are highly efficient in factorizing very big numbers of dozens of digits. Considering the scale of today's PC clusters is still quite limited with merely hundreds or thousands of processors on average, prime factorization can be performed online to support real-time operations. For larger numbers, prime factorization can be always done off-line before the visualization begins. The algorithm ImprovedDS is designed to coordinate the processors' sending/receiving and blending workflows to improve the latency performance of the original Direct Send method.

In Algorithm 19, each processor first obtains its code series from the function, PrimeFactorEncoding(), in line 2. Then, the composition procedure is divided into k sub-stages, where k is the number of prime factors of the given number N of processors. At each sub-stage, we label N' groups in line 6, where $N' = N/f_t$ and f_t is the t-th factor of N. Here, each "group" represents a collection of processors, which only communicate with others in the same group to blend their image tiles. In line 7, "simultaneously" means that all the groups and their member processors are expected to start performing their tasks in parallel at the same time. In line 8, each denoted group ID also obtains its corresponding code series using the PrimeFactorEncoding() function with respect to N'. The N processors are then assigned to the N' groups as follows: a processor W_i is added to G_j if their corresponding code series (i.e., coordinate sequences) $(d_0^i, d_1^i, \cdots, d_{k-1}^i)$ and $(d_0^j, d_1^j, \cdots, d_{k-2}^j)$ satisfy (3.1), as shown in line 10:

$$\begin{cases} d_r^i = d_r^j, & \text{if } 0 \leq r \leq t-2; \\ d_r^i = d_{r+1}^j, & \text{if } t \leq r \leq k-1. \end{cases} \tag{3.1}$$

When the group assignment is completed for all the processors, we call the ImprovedDS algorithm for each group in line 14 to coordinate the processors' communication and blending workflows within the same group. After executing ImprovedDS, we remove all the group associations established in the current sub-stage in line 16 and move on to the next sub-stage for a regrouping of the processors. After completing these k sub-stages, each processor holds a composited image tile, which is then collected by processor W_0 using a Binary-Tree scheme in line 18.

For ImprovedDS in Algorithm 21, given a group G_j containing h processors and the sub-stage, i.e., t, where the algorithm is applied, all the h processors are first arranged in an increasing order according to their original processor IDs in line 1. Similarly, "simultaneously" in line 2 means that all the processors are expected to start performing their tasks in parallel at the same time. Each processor also needs to divide the sub-image it currently holds into h equal-sized tiles and label them by

Function 19 GMPL($W_0, W_1, W_2 \cdots, W_{N-1}$)
Input: N processors with partial images
Output: A composited final image on W_0

```
 1: for $i = 0$ to $N - 1$ do
 2:     ($d^i_0, d^i_1, d^i_2 \cdots, d^i_{k-1}$)=PrimeFactorEncoding($i$,$N$);
 3: for $t = 0$ to $k - 1$ do
 4:     Set $N' = N/f_t$;
 5:     Use $G_0, G_1, \cdots, G_{N'-1}$ to denote $N'$ groups;
 6:     for all $j \in [0, N' - 1]$ simultaneously do
 7:         ($d^j_0, d^j_1, d^j_2 \cdots, d^j_{k-2}$) = PrimeFactorEncoding($j$,$N'$);
 8:         for all $i \in [0, N - 1]$ do
 9:             if the coordinate sequences ($d^i_0, d^i_1, \cdots, d^i_{k-1}$) of $W_i$ and ($d^j_0, d^j_1, \cdots, d^j_{k-2}$) of
                $G_j$ satisfy (3.1) then
10:                 Add $W_i$ to Group $G_j$;
11:         ImprovedDS(Group $G_j$, sub-stage number $t$);
12:     Remove all group associations;
13: Processor $W_0$ collects the composited tiles held by all other processors $W_i$ ($1 \leq$
    $i \leq N - 1$) using a Binary-Tree scheme to compose the final image $I$;
14: return the final image $I$ on processor $W_0$;
```

Function 20 PrimeFactorEncoding(i,N)
Input: a processor ID $i \in [0, N - 1]$, the number N of processors
Output: A sequence of coordinates of the processor ID i in $N's$ prime-factors' hyperspace

```
1: Prime factorize $N$ such that $N = f_0 \cdot f_1 \cdot f_2 \cdot \cdots \cdot f_{k-1}$, where the factors are
   arranged in a descending order;
2: $pid = i$;
3: for $r = k - 1$ to 0 do
4:     $d^i_r = pid$ mod $f_r$;
5:     $pid$=$pid$ div $f_r$;
6: return ($d^i_0, d^i_1, d^i_2, \cdots, d^i_{k-1}$);
```

$I_{t,n}(0 \leq n \leq h-1)$ in line 4 to facilitate the sub-image composition in the following $h-1$ steps, which are described from lines 5 to 9. Note that each processor executes lines 6, 7, and 8 in parallel. At each step s $(0 \leq s \leq h-2)$ of sub-stage t, processor $W_{i'_n}$ sends out the image-tile $I_{t,j(s)}$ to processor $W_{i'_{l(s)}}$, where

$$j(s) = \begin{cases} (n \cdot \lfloor \frac{h}{2} \rfloor + s) \bmod h, & \text{if } 0 \leq s \leq \lfloor \frac{h}{2} \rfloor; \\ (n \cdot \lfloor \frac{h}{2} \rfloor + s + 1) \bmod h, & \text{if } \lfloor \frac{h}{2} \rfloor < s \leq h-2; \end{cases} \tag{3.2}$$

and

$$l(s) = \begin{cases} n + 1 \bmod h, & \text{if } 0 \leq s \leq \lfloor \frac{h}{2} \rfloor - 1; \\ (2h - 2 \cdot j(s) + 1) \bmod h, & \text{if } \lfloor \frac{h}{2} \rfloor \leq s \leq h-2. \end{cases} \tag{3.3}$$

In the mean time, processor $W_{i'_n}$ is also ready to receive an image-tile labeled as $I_{t,b(s)}$ from processor $W_{i'_{m(s)}}$, where

$$b(s) = \begin{cases} (\lfloor \frac{h}{2} \rfloor \cdot T(n) + s) \bmod h, & \text{if } 0 \leq s \leq \lfloor \frac{h}{2} \rfloor; \\ ((n+1) \cdot \lfloor \frac{h}{2} \rfloor + 1) \bmod h, & \text{if } \lfloor \frac{h}{2} \rfloor < s \leq h-2; \end{cases} \tag{3.4}$$

$T(n) = (n - 1 + h) \bmod h$, and

$$m(s) = \begin{cases} n - 1 + h \bmod h, & \text{if } 0 \leq s < \lfloor \frac{h}{2} \rfloor; \\ (n - 1 + 2 \cdot s) \bmod h, & \text{if } s = \lfloor \frac{h}{2} \rfloor; \\ (n + 1 + 2 \cdot s) \bmod h, & \text{if } \lfloor \frac{h}{2} \rfloor < s \leq h-2. \end{cases} \tag{3.5}$$

Function 21 ImprovedDS(Group G_j of h member processors, sub-stage number t)
Input: group G_j of h member processors and their sub-images, number of the sub-stage where the method is applied
Output: each member processor has its partial image composited with other member processors

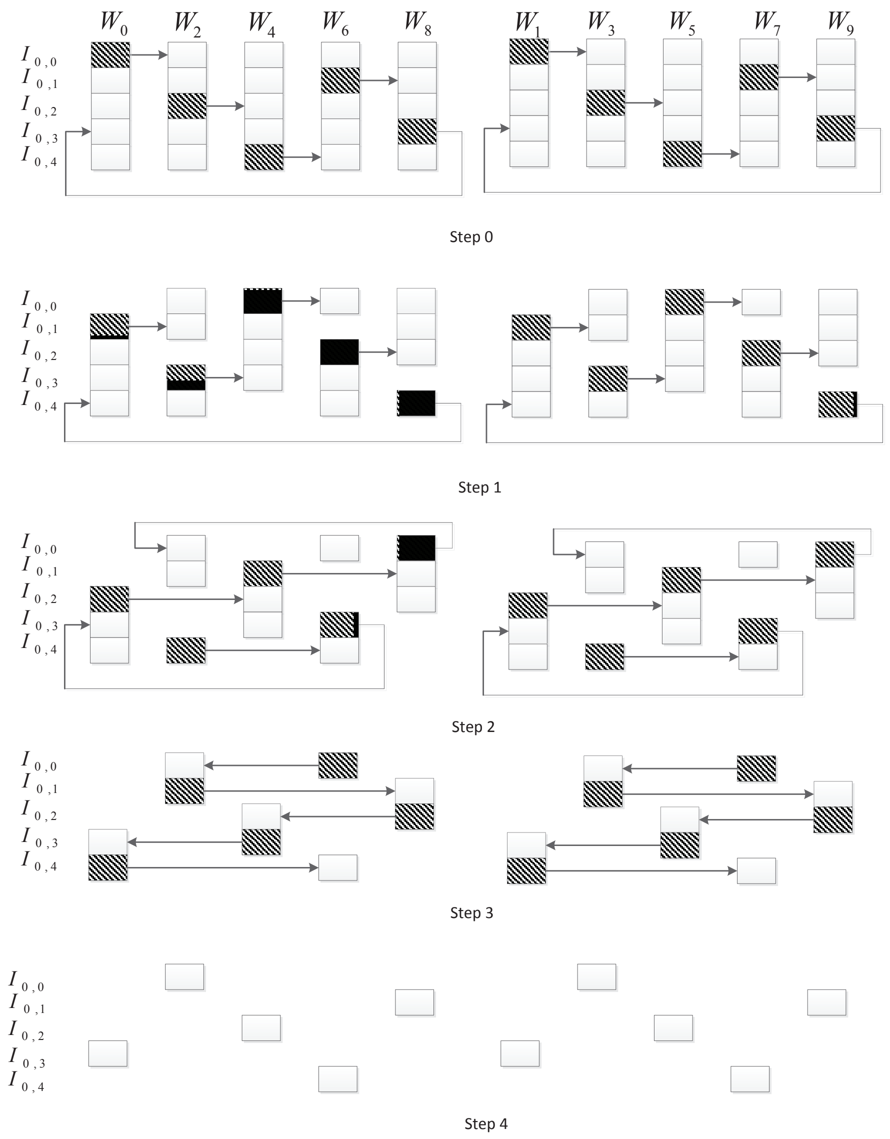
```
1: Denote the $h$ processors in Group $G_j$ as $W_{i'_0}, \cdots, W_{i'_n}, \cdots, W_{i'_{h-1}}$, where $i'_0 \leq \cdots \leq i'_n \cdots \leq i'_{h-1}$;
2: for all $h$ processors simultaneously do
3:   Denote the partial image $W_{i'_n}$ currently holds as $I_t$;
4:   Divide $I_t$ into $h$ equal tiles and label them as $I_{t,0}, \cdots I_{t,n}, \cdots I_{t,h-1}$;
5:   for step $s = 0$ to $h - 2$ simultaneously do
6:     $W_{i'_n}$ sends its tile labeled as $I_{t,j(s)}$ to $W_{i'_{l(s)}}$, where $j(s)$ and $l(s)$ are specified by (3.2) and (3.3), respectively;
7:     $W_{i'_n}$ receives a tile labeled as $I_{t,b(s)}$ from $W_{i'_{m(s)}}$, where $b(s)$ and $m(s)$ are specified by (3.4) and (3.5), respectively;
8:     $W_{i'_n}$ blends its own tile labeled as $I_{t,b(s-1)}$ with the one received at the previous, i.e., $s - 1$ step, when $s > 0$;
```

To better explain the proposed GMPL method, we present a step-by-step example of 10 processors as shown in Figure 3.1. In this example, the number of processors, i.e., 10, is prime factorized into 5×2, and the composition procedure is thereby divided into 2 sub-stages with 2 and 5 groups, respectively. In the first sub-stage, i.e., sub-stage 0, processors W_0, W_2, W_4, W_6, and W_8 constitute one group and act as $W_{i'_0}, W_{i'_1}, W_{i'_2}, W_{i'_3}$, and $W_{i'_4}$, respectively, in the application instance of the ImprovedDS method to their group; processors W_1, W_3, W_5, W_7, and W_9 constitute another group and act as $W_{i'_0}, W_{i'_1}, W_{i'_2}, W_{i'_3}$, and $W_{i'_4}$, respectively. Similarly, we can derive the grouping and each processor's activities in sub-stage 1 using the proposed algorithms.

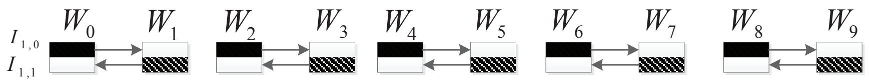
3.2.2 Algorithm Analysis

We provide a thorough analysis of our proposed GMPL method in this subsection in reference to a composition procedure comprised of k sub-stages.

Analysis of ImprovedDS Considering a processor group G_j of h processors, at sub-stage t of ImprovedDS, we have the following properties:



(a) Sub-stage 0



(b) Sub-stage 1

Figure 3.1 Illustration of the proposed GMPL method in the case of 10 processors.

- Property 1: At each step s $(0 \le s \le h-2)$ of group G_j's composition procedure, each processor $W_{i'_n}$ $(0 \le n \le h-1)$ has an exclusive sending destination $W_{i'_{l(s)}}$ and receiving source $W_{i'_{m(s)}}$ within the group.

- Property 2: When the composition within group G_j is finished, processor $W_{i'_n}$ ($0 \leq n \leq h-1$) holds an exclusive tile labeled as $I_{t,r(n)}$, into which all the h tiles with the same label in the group have been blended, where

$$r(n) = ((n+1) \cdot \lfloor \frac{h}{2} \rfloor + 1) \bmod h. \tag{3.6}$$

Property 1 indicates that at any step of ImprovedDS, each processor's sending or receiving partner has no dependency with any other one performing the same job in the same group, which, together with the multi-port communication model, leads to a high level of parallelism in the communication process of ImprovedDS. Property 2 is used to establish the correctness of its composition procedure.

Proof of Property 1: Without loss of generality, we focus on the sending process, as the proof for the receiving process is similar.

Considering group G_j with $h-1$ steps, we divide these steps into two disjoint sets: $[0, \lfloor \frac{h}{2} \rfloor]$, and $(\lfloor \frac{h}{2} \rfloor, h-2]$.

In the first set $[0, \lfloor \frac{h}{2} \rfloor]$, according to the upper part of (3.3), which determines the destination of the tile sent from each processor at a specific step s, each processor's sending destination corresponds one-to-one to the processor's unique ID in its current group. Therefore, there is no conflict of data sending between any two processors in the same group.

In the second set $(\lfloor \frac{h}{2} \rfloor, h-2]$, from the lower part of (3.3), we know that given a processor $W_{i'_n}$ and its current step s, $W_{i'_n}$'s sending destination at step s, i.e., $W_{i'_{l(s)}}$, is related to the following three parameters: n (i.e., the processor's unique ID in the current group, h (i.e., the number of processors in the group), and s (i.e., the step number in the entire procedure). Considering that h and s are the common parameters for the entire group at a given step, each processor's sending destination is only determined by its unique group ID. For clarity, we represent $W_{i'_n}$'s sending destination at a given step s, i.e., $W_{i'_{l(s)}}$, as $W(n)$. The conflict-free sending problem at a given step s can be converted into the following problem: given $i, j \in [0, h-1]$, if $i \neq j$, then $W(i) \neq W(j)$.

We use a proof-by-contradiction strategy to prove the above property. Given $i, j \in [0, h-1]$, let us assume that $i < j$ and $W(i) = W(j)$. Based on the properties of the $\mod$ operation, which dominates the operations in function $W(n)$, we can infer that $W(j-i) = 0$, i.e., $j - i$ is a cyclic period of $W(n)$. On the other hand, from $W(n)$'s definition, we know that $W(n)$ naturally has a cyclic period h. For $j - i$ and h, since $0 \leq i, j < h$, $j > i$, it follows that $0 < j - i < h$. According to a basic theorem in Algebra Theories [69], the following equality involving $j - i$ and h holds:

$$c = q \cdot (j - i) + p \cdot h, \tag{3.7}$$

where q and p are both integers, and c is the greatest common divisor between $j - i$ and h.

From (3.7), we know that c is also a cyclic period of $W(n)$. Since h is a prime, c must be 1. It means that $W(0) = W(1) = \cdots = W(h-2) = W(h-1)$, which obviously is incorrect. For example, $W(0) = s - 1$, $W(1) = (\lfloor \frac{h}{2} \rfloor + s - 1) \bmod h$, and $W(0) \neq W(1)$. Therefore, it conflicts with our assumption that $W(i) = W(j)$. Proof ends. □

Proof of Property 2: The proof of Property 2 is divided into two parts: i) processor $W_{i'_n}$ holds a unique tile when composition is completed, and ii) the obtained tile $I_{t,r(n)}$ has blended all the h tiles with the same label.

Combining Equations (3.2) and(3.4), we know that for processor $W_{i'_n}$, after the $h-1$-step image-tile sending and receiving, all the tiles have been sent out except the ones labeled as $I_{t,b(h-2)}$. After blending them, we obtain the final tile held by $W_{i'_n}$, i.e., $I_{t,r(n)}$, where $r(n) = ((n+1) \cdot \lfloor \frac{h}{2} \rfloor + 1) \bmod h$.

For the problem of complete composition, we consider the tile labeled as $I_{t,r}$,$(0 \leq r \leq h-1)$, and the proof for the rest can be done in a similar way. According to the upper part of (3.4), we know that in steps $[0, \lfloor \frac{h}{2} \rfloor]$, there are $\lfloor \frac{h}{2} \rfloor$ processors receiving tiles labeled as $I_{t,r}$, one tile for each receiving processor. These receiving processors would blend immediately once receiving the $I_{t,r}$-labeled tile. Thus, the number of $I_{t,r}$-labeled tiles in group G_j is reduced from h to $h - \lfloor \frac{h}{2} \rfloor - 1$. In the following steps $(\lfloor \frac{h}{2} \rfloor, h-2]$, $I_{t,r}$-labeled tiles are sent to $W_{i'_n}$, one tile at each step. When the latter steps are finished, the number of $I_{t,r}$-labeled tiles in group G_j is further reduced from $h - \lfloor \frac{h}{2} \rfloor - 1$ to $h - \lfloor \frac{h}{2} \rfloor - 1 - (h - 2 - \lfloor \frac{h}{2} \rfloor) = 1$, which is the final composited tile. Proof ends. $\square$

We would like to point out that the blending operations in ImprovedDS can also be performed in parallel with data communications with necessary synchronization. For example, upon the receival of a tile $I_{t,b(s)}$, the processor starts blending, and in the meanwhile, it continues to receive another tile. Hence, the sending/receiving parts of ImprovedDS overlap with the blending part. With some buffer in place for the purpose of data caching, such parallelization can be further exploited, especially on a homogeneous system.

In addition, compared with the original Direct Send algorithm, ImprovedDS results in a great improvement on latency. For a group containing h processors, since each processor sends/receives with a fixed partner in the first $\lfloor \frac{h}{2} \rfloor$ steps, the connections need to be established only once, instead of $\lfloor \frac{h}{2} \rfloor$ times as in the Direct Send algorithm, hence leading to a significant performance gain, especially when h is large.

Analysis of GMPL In the GMPL method, the activities of different groups are also performed in parallel. At each sub-stage $t (0 \leq t \leq k-1)$, based on the unique code series of each processor and group, the way of group association in GMPL ensures that each processor is associated with only one group. Therefore, all the groups are independent of each other and work in parallel. Ideally, on a homogeneous system, each processor is expected to start and finish stage t at the same time.

The above analysis, together with that in Section 3.2.2, shows that the proposed GMPL method features completely balanced workload, a high level of resource utilization, and a high degree of parallelism, which, collectively, contribute to an improved latency performance. Furthermore, GMPL removes the power-of-2 constraint and works on an arbitrary number of processors. Particularly, when the number N of processors is a power of 2, GMPL reduces to the traditional Binary Swap method. When N is prime, the GMPL procedure has only one sub-stage where all the processors are in the same group executing the ImprovedDS algorithm directly.

3.2.3 Theoretical Analysis and Comparison of Latency Performance

We conduct a theoretical analysis and comparison of latency performance between the proposed GMPL method and several existing methods including Binary Swap, Direct Send, and Parallel Pipeline. We consider the cost models for both data communication and image blending as follows:

- Data communication: the time cost of sending/receiving a tile is calculated as: $\alpha + n\beta$, where α is the link delay, β is transmission time for each pixel, and n is the number of pixels in the tile.
- Image blending: the time cost of each image tile blending is calculated as $n\gamma$, where γ is the blending time of each pixel, and n is the number of pixels in the tile.

Based on the above cost models, the entire composition time mainly consists of 3 components: connection establishing time T_l due to the link delay, image transfer time T_c, and image blending time T_b. Our performance analysis and comparison are focused on these 3 aspects as well as the total composition time T_a.

Binary Swap, Direct Send, and Parallel Pipeline Binary Swap (BS) is one of the most traditional methods for image composition and has been thoroughly analyzed in the literature with the following three time cost components [42, 71]:

$$T_l(BS) = \alpha \cdot \log_2 N,$$

$$T_c(BS) = \beta \cdot P \cdot (1 - \frac{1}{N}),$$

$$T_b(BS) = \gamma \cdot 2 \cdot P \cdot (1 - \frac{1}{N}),$$

where N is the number of processors and P is the number of pixels in the image to be composited. The total time $T_a(BS)$ BS takes is:

$$T_a(BS) = T_l(BS) + T_c(BS) + T_b(BS).$$

In Direct Send (DS), the three time cost components are [18, 71]:

$$T_l(DS) = \alpha \cdot (N - 1),$$

$$T_c(DS) = \beta \cdot P \cdot (1 - \frac{1}{N}),$$

$$T_b(DS) = \gamma \cdot P,$$

respectively, and the total time cost is:

$$T_a(DS) = \max\{T_l + T_c, T_b\}. \tag{3.8}$$

Parallel Pipeline (PP) arranges the given N processors in a $f_0^{i'} \times f_1^{i'}$ mesh architecture, where $f_0^{i'}$ and $f_1^{i'}$ can be any two over-1 integers that follow $N = f_0^{i'} \cdot f_1^{i'}$ [37]. Given a specific pair of $f_0^{i'}$ and $f_1^{i'}$, the three time cost components of PP are as follows:

$$T_l(PP) = \alpha \cdot \sum_{s=0}^{1} (f_s^{i'} - 1),$$

$$T_c(PP) = \beta \cdot P \cdot (1 - \frac{1}{N}),$$

$$T_b(PP) = \gamma \cdot P \cdot (1 - \frac{1}{N}),$$

respectively, and its total time cost is dependent on the factorization result.

Grouping More and Pairing Less(GMPL) In Grouping More and Pairing Less (GMPL), given N processors, if $N = f_0 \cdot f_1 \cdot f_2 \cdots f_{k-1}$, the composition process contains k sub-stages. From Algorithm 19 and 21, at each sub-stage t $(0 \leq t \leq k-1)$, the number $P(t)$ of pixels each processor holds when sub-stage t starts is

$$P(t) = \begin{cases} P, & \text{if } t = 0; \\ \frac{P}{f_0 \cdot f_1 \cdot f_2 \cdots f_{t-1}}, & \text{if } 1 \leq t \leq k-1; \\ \frac{P}{N}, & \text{if } t = k, \end{cases} \tag{3.9}$$

the number g_t of groups in stage t is $\frac{N}{f_t}$, and the number w_t of working processors contained in each group is equal to f_t.

At sub-stage t, although the processors are assigned to different groups, each processor still works in a similar way. Each processor needs to simultaneously send to and receive from other processors for $w_t - 1$ times. For the first $\lfloor \frac{w_t}{2} \rfloor$ transfers, each processor sends/receives its tiles to/from the same processor, so the connection needs to be established only once when the first tile is transferred, and the rest $\lfloor \frac{w_t}{2} \rfloor - 1$ transfers do not incur any extra connection overhead. Therefore, the connection establishment latency for the first $\lfloor \frac{w_t}{2} \rfloor$ transfers is α.

For the last $w_t - 1 - \lfloor \frac{w_t}{2} \rfloor$ transfers, each processor changes its sending/receiving partners at each step, and hence needs to establish a new connection each time. The connection establishment latency for the last $w_t - 1 - \lfloor \frac{w_t}{2} \rfloor$ transfers is $(w_t - 1 - \lfloor \frac{w_t}{2} \rfloor)\alpha$. By summing up the connection establishment cost for all w_t transfers, we calculate the sending/receiving latency T_l^t in sub-stage t as:

$$\begin{aligned} T_l^t &= \alpha \cdot (1 + w_t - 1 - \lfloor \frac{w_t}{2} \rfloor) \\ &= \alpha * (w_t - \lfloor \frac{w_t}{2} \rfloor) \\ &= \alpha \cdot (f_t - \lfloor \frac{f_t}{2} \rfloor). \end{aligned} \tag{3.10}$$

Furthermore, the total latency T_l of each processor for all k sub-stages is calculated as:

$$\begin{aligned} T_l &= \sum_{t=0}^{k-1} T_l^t \\ &= \alpha \cdot \sum_{t=0}^{k-1} (f_t - \lfloor \frac{f_t}{2} \rfloor) \\ &\approx \sum_{t=0}^{k-1} (\frac{f_t}{2}) \cdot \alpha. \end{aligned} \tag{3.11}$$

The image transfer time T_c^t of each processor in sub-stage t is straightforward, i.e.,

$$\begin{aligned} T_c^t &= \beta \cdot \frac{P(t)}{w_t} \cdot \lfloor \frac{w_t}{2} \rfloor + \beta \cdot (w_t - 1 - \lfloor \frac{w_t}{2} \rfloor) \cdot \frac{P(t)}{w_t} \\ &= \beta * \frac{P(t)}{w_t} * (w_t - 1 - \lfloor \frac{w_t}{2} \rfloor + \lfloor \frac{w_t}{2} \rfloor) \\ &= \beta \cdot \frac{P(t)}{w_t} \cdot (w_t - 1) \\ &= \beta * (P(t) - \frac{P(t)}{w_t}) \\ &= \beta * (P(t) - \frac{P(t)}{f_t}) \\ &= \beta \cdot (P(t) - P(t+1)). \end{aligned} \tag{3.12}$$

Similarly, the total image transfer time T_c of each processor for all k sub-stages is:

$$\begin{aligned} T_c &= \sum_{t=0}^{k-1} T_c^t = \sum_{t=0}^{k-1} \beta \cdot (P(t) - P(t+1)) \\ &= \beta \cdot (P(0) - P(k) = \beta \cdot P \cdot (1 - \frac{1}{N}). \end{aligned} \tag{3.13}$$

The image blending time T_b^t of a processor at sub-stage t can be derived in a similar way to (3.12):

$$T_b^t = \gamma \cdot (P(t) - P(t+1)). \tag{3.14}$$

It follows, for the total image composition time T_b of a processor during all the k sub-stages, that:

$$T_b = \gamma \cdot P \cdot (1 - \frac{1}{N}), \tag{3.15}$$

which is also derived in a similar way to (3.13).

Similar to Direct Send, since the processors in the same group are arranged in a pipeline to overlap the data sending/receiving and image blending tasks, the total time T_a^t a processor spends at sub-stage t is:

$$T_a^t = \max\{T_l^t + T_t^t, T_b^t\}. \tag{3.16}$$

Considering the homogeneity in the computing power, the amount of workload, and the composition workflow, the processors in the same group enter and leave the sub-stage t at the same time. As the same is true in all the groups, T_a^t is actually the time that each processor spends in sub-stage t. Similarly, the total time cost T_a for the entire composition procedure is:

$$T_a = \sum_{t=0}^{k-1} T_a^t. \tag{3.17}$$

As a summary, we provide the above theoretical latency performance analysis results of the image composition methods in comparison in Table 3.1. The comparison of the image transfer and blending time is straightforward, but not the comparison

Table 3.1 Latency Performance Analysis of Four Methods in Comparison, where T_l is the Connection Establishing Time, T_c is the Image Transfer Time,T_b is the Image Blending Time

Methods	T_l	T_c	T_b
BS	$\alpha \cdot \log_2 N$	$\beta \cdot P \cdot (1 - \frac{1}{N})$	$2 \cdot \gamma \cdot P \cdot (1 - \frac{1}{N})$
DS	$\alpha \cdot (N - 1)$	$\beta \cdot P \cdot (1 - \frac{1}{N})$	$\gamma \cdot P \cdot (1 - \frac{1}{N})$
PP	$\alpha \cdot \sum_{t=0}^{1} (f_t^{i'} - 1)$	$\beta \cdot P \cdot (1 - \frac{1}{N})$	$\gamma \cdot P \cdot (1 - \frac{1}{N})$
GMPL	$\alpha \cdot \sum_{t=0}^{k-1} (f_t - \lfloor \frac{f_t}{2} \rfloor)$	$\beta \cdot P \cdot (1 - \frac{1}{N})$	$\gamma \cdot P \cdot (1 - \frac{1}{N})$

of the connection establishing time, which is in some form of summation over the factors of the processor number N. For an accurate comparison, we introduce a new variable F^i. Given one arbitrary factorization of an arbitrary integer N, i.e., $N = f_0^i \cdot f_1^i \cdot f_2^i \cdots f_{k_i-1}^i$, we use F^i to denote the following factors summation:

$$F^i = \sum_{t=0}^{k_i-1} (f_t^i - 1).$$

The connection establishing time of PP can be represented by F^i under the constraint that k_i can be at most 2. The connection establishing time of DS can be represented by F^i under the constraint that k_i must be 1, which also means that f_0^i is N itself. The connection establishing time of GMPL can be represented by F^i under the constraint that all the factors $f_j^i (0 \leq j \leq k_i - 1)$ must be prime.

Based on F^i, we are able to compare the connection establishing time of these four methods if we can determine how F^i varies with different factorizations. We provide below one observation of such pattern:

Given an arbitrary integer $N^{'}$, if there exists one possible 2-factor factorization $N^{'} = f_0^{i^{'}} \cdot f_1^{i^{'}}$, where $f_0^{i^{'}} \geq 1, f_1^{i^{'}} \geq 1$, then $N^{'}$ and $F^{i^{'}}$ have the following relationship:

$$\begin{aligned} N^{'} - 1 - F^{i^{'}} &= N^{'} - 1 - \sum_{t=0}^{1}(f_t^{i^{'}} - 1) \\ &= f_0^{i^{'}} \cdot f_1^{i^{'}} - 1 - (f_0^{i^{'}} + f_1^{i^{'}} - 2) \\ &= (f_0^{i^{'}} - 1) \cdot (f_1^{i^{'}} - 1) \geq 0. \end{aligned} \tag{3.18}$$

The equality in (3.18) is satisfied only if N is prime.

(3.18) can be further generalized as follows: for an arbitrary N and its factorization f^i, where $N = f_0^i \cdot f_1^i \cdots f_{k_i-1}^i$, if there exists f_t^i $(0 \leq t \leq k_i - 1)$, which is non-prime and greater than 1, then f_t^i can be further factorized as $f_t^i = f_0^{i^{'}} \cdot f_1^{i^{'}}$. The newly derived factors $f_0^{i^{'}}$ and $f_1^{i^{'}}$, as well as the rest $k_i - 1$ factors in f^i, constitute a new factorization f^j of N and we denote its corresponding factor-summation as F^j. Then based on (3.18), it follows that $F^i \leq F^j$. If there still exists a non-prime factor in f^j, the replacement of the non-prime factor can be recursively performed until all the factors are prime, at which point, the minimum factor-summation F^l is achieved.

The notation F^i is also applicable to BS, where the processor number N is required to be a power of 2, i.e.,

$$N = \prod_{t=0}^{\log_2 N - 1} 2, \tag{3.19}$$

and its corresponding connection establishing time is calculated as:

$$\alpha \cdot \sum_{t=0}^{\log_2 N - 1} (2 - \lfloor \frac{2}{2} \rfloor) = \alpha \cdot \log_2 N. \tag{3.20}$$

Obviously, (3.19) is the prime factorization required by GMPL. The performance superiority of BS over PP and DS confirms what we derive from the notation F^i. Based on (3.18), we conclude that the connection establishing time of GMPL is less than, or at most equal to, that of PP, DS, and BS.

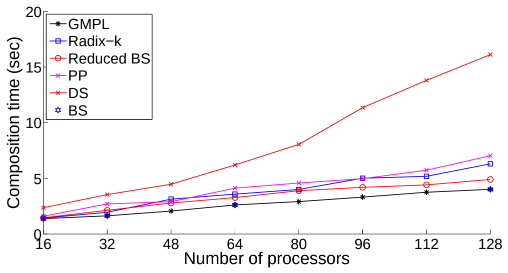
3.2.4 Performance Evaluation

To illustrate the actual composition result and evaluate the performance of GMPL, we implement and test it on a high-performance visualization cluster together with the other five algorithms in comparison including DS, PP, Reduced BS, and BS to support our theoretical analysis, as well as Radix-k to evaluate the effects of the order-dependence restriction. This cluster consists of 1 head node and 16 compute nodes, each of which is equipped with dual 6-core processors at the speed of 2.3 GHz for each core and 64GB RAM, and is connected through a 1GigE switch. The cluster is able to launch $17 \times 12 = 204$ MPI jobs in parallel.

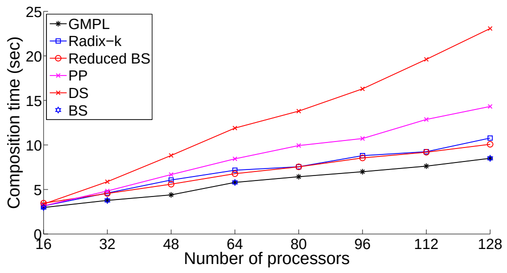
We run GMPL and the other algorithms on different numbers of cores ranging from 16 to 128 at an interval of 16 with different image sizes from $2048 \times 2048, 3072 \times 3072, 4096 \times 4096$, to 5120×5120 pixels. To facilitate comparison, the arbitrary factorization of the number N of cores in PP and Radix-k is specified as a factorization into $2 \times \frac{N}{2}$ and all its prime factors, respectively. Each input image is processed directly by all the composition methods without applying any optimization techniques such as blank pixel elimination and image re-coding.

For each problem instance, we repeat the experiment for five times, and plot in Figure 3.2 the average of the total latency performance measurements of all the methods in comparison on four different image sizes. From these performance curves, we have the following observations.

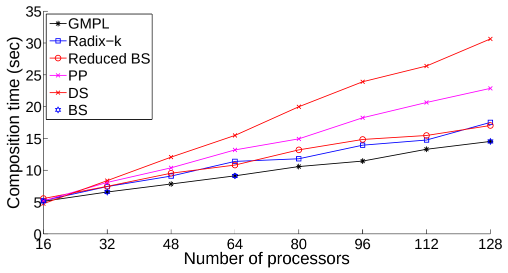
1) For each image size, DS does not factorize the number N of processors, PP factorizes N into two factors, Radix-k, GMPL, Reduced BS, and BS factorize



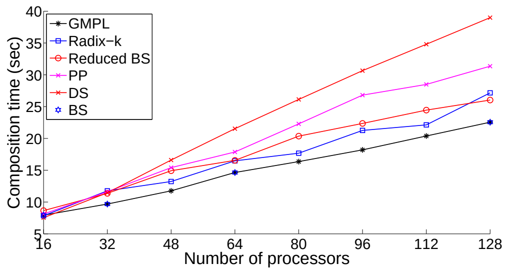
(a) Image size of 2048^2.



(b) Image size of 3072^2.



(c) Image size of 4096^2.

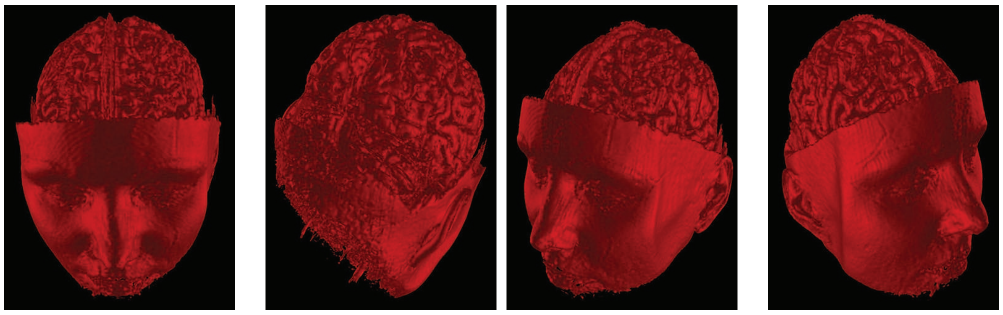


(d) Image size of 5120^2.

Figure 3.2 Latency performance of different methods on different image sizes and numbers of processors.

N into prime factors (the prime factors are $2's$ for Reduced BS and BS). The methods with more thorough factorization seem to outperform those with less factorization in terms of the total composition time, which is consistent with our derivation in (3.18).

2) When the image size is fixed, as the number of processors increases, the performances of PP and DS degrade quickly as their data transfer time cost is linearly related to the number of processors. The performances of BS and GMPL are relatively more stable with the increase of processors as their workload is evenly distributed among the working processors and the data transfer cost logarithmically depends on the number of processors. The performance of Radix-k lies between PP and GMPL, since it takes relatively thorough factorization but retains the restriction of order-dependence. The performance of Reduced BS is unstable as it treats different numbers of processors in different



(a) Front View **(b)** Back View **(c)** Left View **(d)** Right View

Figure 3.3 Composited brain images of size 2048^2 using the GMPL method.

ways. In most cases, Reduced BS takes longer than GMPL to complete the composition, even up to twice of GMPL on 112 processors with an image size of 5120×5120. This observation may be related to the fact that the more pixels one image contains, the longer it takes those processors beyond 2^n ($n = log_2\lfloor N \rfloor$, N is the number of processors) and their partners to transfer and composite images when the number of processors is not a power of 2.

3) When the number of processors is fixed, as the image size increases, the processing time of each method increases correspondingly as expected.

Figure 3.3 shows a set of composited images of size 2048^2 rendered from a brain CT dataset using the proposed GMPL method from different view angles. The composited images of other sizes are qualitatively similar.

CHAPTER 4

V4BD

Many extreme-scale scientific applications generate colossal amounts of data that require a large number of processors for parallel visualization. Among the three well-known visualization schemes, i.e., sort-first/middle/last, sort-last, which is comprised of two stages, i.e., image rendering and composition, is often preferred due to its adaptability to load balance. We propose a very-high-speed pipeline-based architecture for parallel sort-last visualization of big data by developing and integrating three component techniques: i) a fully parallelized per-ray integration method that significantly reduces the number of iterations required for image rendering; ii) a real-time *over* operator that not only eliminates the restriction of pre-sorting and order-dependency, but also facilitates a high degree of parallelization for image composition; and iii) a novel sort-last visualization pipeline that overlaps rendering and composition to completely avoid waiting time between these two stages. The performance superiority of the proposed parallel visualization architecture is evaluated through rigorous theoretical analyses and further verified by extensive experimental results from the visualization of various real-life scientific datasets on a high-performance visualization cluster.

4.1 Background

Next-generation simulation-based e-sciences are producing colossal amounts of data, now frequently termed as "Big Data", on the order of terabyte at present and petabyte or even exabyte in the predictable future. Such data must be visualized and analyzed in a timely manner for knowledge discovery and scientific innovation. In fact, on-line simulation monitoring and interactive computational steering through visual feedback constitutes a critical part of these research processes.

Among the above three well-known parallel schemes, sort-last is often preferred in many applications due to its adaptability to load balance. In general, after data partitioning and distribution, a sort-last parallel visualization scheme takes two sequential steps, i.e., image rendering and image composition, which consume most of the time in the entire visualization process. There exist separate research efforts in these two individual problems, for example, early ray termination for image rendering and binary-swap for image composition, and GPU-based parallelization for both. However, the overall performance of parallel volume visualization still suffers mainly from the following three aspects: i) the traditional raycasting approach follows an almost serial procedure and hence may incur a long delay when sampling and blending a large number of data blocks; ii) the traditional *over* operator performs order-dependent composition and hence may incur a long idle time when fragments arrive out of order; iii) the treatment of rendering and composition as two strictly sequential steps may incur a long waiting time between them due to unbalanced workload. These issues are becoming even more prominent as the data volume continues to increase and the computing platform continues to expand at an unprecedented pace, which makes it extremely challenging to achieve load balance among different nodes in a parallel computing architecture.

In this chapter, we propose an architecture for Very-high-speed Value-added Volume Visualization of Big Data, referred to as V4BD, by developing and integrating three component techniques: i) a fully parallelized per-ray integration method that significantly reduces the number of iterations required for image rendering; ii) a real-time *over* operator that not only eliminates the restriction of pre-sorting and order-dependency, but also facilitates a high degree of parallelization for image composition; and iii) a novel sort-last visualization pipeline that overlaps rendering and composition to completely avoid waiting time between these two stages. The performance superiority of the proposed architecture is evaluated through rigorous theoretical analyses and further verified by extensive experimental results from the visualization of various real-life scientific datasets on a high-performance visualization cluster.

4.2 Related Work

We provide a survey of existing work related to each of the proposed component techniques.

4.2.1 Raycasting and its Acceleration

Research on raycasting is mainly focused on two aspects: 1) required adjustments as applied to non-uniform volume data, e.g. Bunyk [9] considered irregular grid data, Zhu [73] considered segmented regular volume data, and Marmitt [44] considered tetrahedral and hexahedral meshes; 2) performance improvements, e.g. Grimm [27] achieved performance gains through optimized memory utilization, Bernardon [5] achieved the same goal by decomposing the screen into tiles, Kruger [34] adopted early ray termination (Z-test feature), and Lee [36] parallelized the integration procedure for different rays and ported it to GPU platforms. However, the work on parallelizing per-ray integration in big data visualization still remains largely unexplored.

4.2.2 *over* Operator for Image Composition

There exist several research efforts in addressing the performance issues associated with the traditional *over* operator. Meshkin [45] approximated the *over*-composition result for n input pixels by ignoring the order-sensitive parts in their extended *over*-composition formula. Bavoil and Myers [4] approximated n-pixel *over*-composition by calculating the average of all input pixels and substituting it for each pixel, which is a special case of the extended n-pixel *over*-composition formula with all the input pixels being identical. Patney *et al.* [54] proposed a generalized formula for the color components without considering $\alpha-$channel of transparency.

The performance limitation on image composition caused by order dependency still stands out in practice. In many existing composition frameworks, fragments must be pre-sorted before the actual composition can take place [54, 28], hence significantly limiting the composition performance, especially in dynamic environments where fragments may arrive out of order. Unfortunately, very limited efforts have been devoted to this issue. Our work removes the order dependency of the traditional *over* operator and hence makes an important advancement in this field.

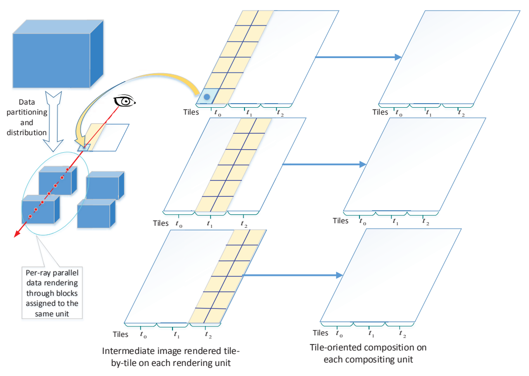
4.2.3 Sort-last Visualization Architecture

Several research efforts have been made to develop pipeline-based visualization architectures. Cavin [11] proposed to overlap the generation processes for consecutive images in the expected sequence. Such a pipeline outperforms the traditional sequential process when visualizing multiple images, but does not exhibit a strong performance advantage when visualizing a single image. Fang [20] employed similar strategies to pipeline image composition also for multiple images. In this paper, we develop a parallel computing architecture that pipelines the entire visualization process for big data including the rendering and composition of single and multiple images.

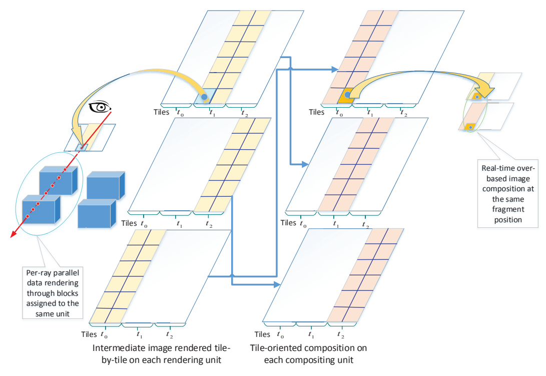
4.3 V4BD: A Very High-speed Value-added Volume Visualization Architecture for Big Data

We consider the visualization of big data B using a parallel computing architecture with $2n$ homogeneous nodes, each of which supports at most N_t concurrent threads. These nodes are connected via a high-speed switch and are divided into two equal-sized groups, i.e., $\{r_0, r_1, \cdots, r_{n-1}\}$, $\{c_0, c_1, \cdots, c_{n-1}\}$, for dedicated image rendering and composition, respectively.

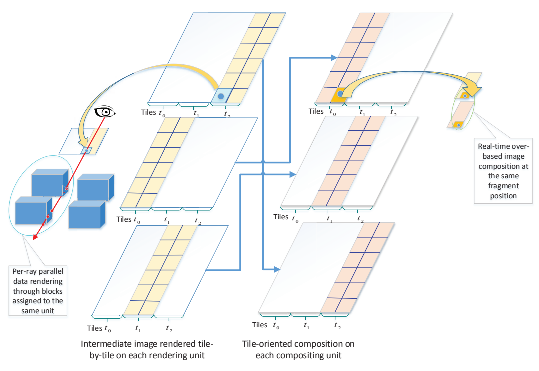
In volume visualization, a user typically specifies a view port VP and a size $x \times y$ of the final image for display. To support parallel processing, the data B is first divided


Data partitioning and distribution
Per-ray parallel data rendering through blocks assigned to the same unit
Tiles t_0 t_1 t_2
Intermediate image rendered tile-by-tile on each rendering unit
Tile-oriented composition on each compositing unit

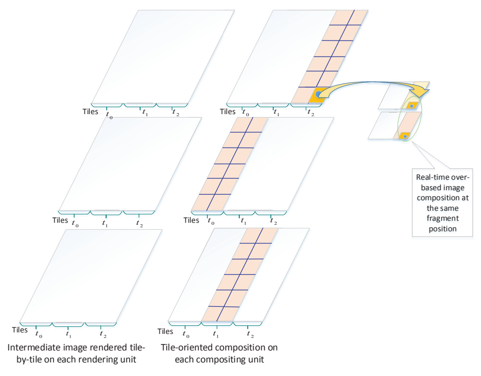
(a) Step 1


Per-ray parallel data rendering through blocks assigned to the same unit
Tiles t_0 t_1 t_2
Real-time over-based image composition at the same fragment position
Intermediate image rendered tile-by-tile on each rendering unit
Tile-oriented composition on each compositing unit

(b) Step 2


Per-ray parallel data rendering through blocks assigned to the same unit
Tiles t_0 t_1 t_2
Real-time over-based image composition at the same fragment position
Intermediate image rendered tile-by-tile on each rendering unit
Tile-oriented composition on each compositing unit

(c) Step 3


Tiles t_0 t_1 t_2
Real-time over-based image composition at the same fragment position
Intermediate image rendered tile-by-tile on each rendering unit
Tile-oriented composition on each compositing unit

(d) Step 4

Figure 4.1 The execution process of the proposed V4BD architecture in a simple case of three rendering/composition units.

into k equal-sized blocks $B_0, B_1, \cdots, B_{k-1}$, $k >> n$, which are then distributed in their adjacency to different raycasting-based rendering units r_i, $i = 0, 1, \cdots, n-1$, with an attempt to achieve load balance. For the convenience of referencing the fragments in an intermediate image and the pixels in the final image, we employ a commonly used coordinate system, where the top-left corner of the display region is set as the origin, and the x-axis/y-axis is aligned with the horizontal/vertical dimension of the image.

We propose architecture for Very-high-speed Value-added Volume Visualization of Big Data, referred to as V4BD, as shown in Figure 4.1, which illustrates the execution process of a simple case with three units for rendering and composition, respectively. The proposed V4BD architecture integrates three component techniques: a fully parallelized per-ray integration method, a real-time *over* operator, and a sort-last visualization pipeline.

In the proposed V4BD architecture, the entire image area is divided into n equal-sized tiles (n is the number of rendering/composition units and $n = 3$ in this example as shown in Figure 4.1). Each rendering unit renders a full-size intermediate image on a tile-by-tile basis horizontally, and each composition unit is responsible for compositing all the intermediate tiles at the same position vertically. The entire visualization process takes $n + 1$ steps as follows. In Step 1, each rendering unit performs data integration independently for a different image tile, and sends the rendered (intermediate) image tile to its corresponding composition unit once finished. In Steps 2 to n, all processing units perform rendering or composition simultaneously on different image tiles, and the intermediate image tiles are sent from a rendering unit to a composition unit as the pipeline progresses. In Step $n+1$, each composition unit performs the last image composition to produce the final image tile at one particular position. Note that during the above process, the rendering units employ the proposed rendering method for parallel data integration and the composition units employ the proposed *over* operator for parallel image composition. By pipelining image rendering and composition using the proposed techniques, we are able to overlap the operations of rendering and composition and hence completely avoid waiting time between these two stages.

4.3.1 Parallelized Per-Ray Integration (PPRI)

Derivation of Data Integration Process In volume visualization, image rendering typically takes longer than image composition and often becomes the bottleneck of the entire process. The traditional raycasting approach is a serial procedure, which does not scale well with the number of data blocks each ray has to traverse. This problem becomes worse as the data volume continues to grow rapidly. We propose a parallelized per-ray integration method to address this problem.

Given a particular ray and its propagation direction, we denote its n RGBA-formatted and sequentially sampled points as $S_1, S_2, \cdots$, and S_n. To facilitate our explanation, we introduce another notation $S_{i,j}$, $1 \leq i \leq j \leq n$, which represents the integration result of $S_i \oplus S_{i+1} \oplus \cdots \oplus S_j$ and also serves as a uniform representation for any possible (raw, intermediate, or final) integrating results in the entire integration process. For example, when $i = j$, it refers to the raw sampling point S_i or S_j; when $i = 1$ and $j = n$, it refers to the final integrated result from all n raw sampling points.

Given $S_i = [c_{R_i}, c_{G_i}, c_{B_i}, \alpha_i]^T$, $i \in \{1, 2\}$, we have

$$\begin{aligned} S_{1,2} = S_1 \oplus S_2 &= [c_{R_{1,2}}, c_{G_{1.2}}, c_{B_{1,2}}, \alpha_{1,2}]^T \\ &= S_1 + (1 - \alpha_1) \cdot S_2, \end{aligned} \tag{4.1}$$

and the following law of association holds [42]:

$$S_1 \oplus (S_2 \oplus S_3) = (S_1 \oplus S_2) \oplus S_3.$$

Given n sampling points $S_1, S_2, \cdots, S_i, \cdots, S_n$, we have

$$\begin{aligned}
&S_1 \oplus S_2 \oplus \cdots \oplus S_i \oplus \cdots \oplus S_n \\
=&S_{1,2} \oplus S_{3,4} \oplus \cdots \oplus S_{i,i+1} \oplus \cdots \oplus S_{n-1+n\%2,n} \\
=&S_{1,4} \oplus \cdots \oplus S_{i,i+3} \oplus \cdots \oplus S_{n-3+(n\%4),n} \\
=&\cdots \\
=&S_{1,\lfloor \frac{n}{2} \rfloor} \oplus S_{\lfloor \frac{n}{2} \rfloor+1,n},
\end{aligned} \tag{4.2}$$

which inspires a parallel integration of n sampling points in a binary tree structure if all the sampling points along a given ray are simultaneously available.

Algorithm Design For each ray, we first create n threads to calculate n sampling points simultaneously in the assigned data blocks and then integrate them in a parallelized procedure with $\ln(n)$ steps. Following (4.2), at step t, $0 \le t \le \ln(n)$, there exist $\lceil n/2^t \rceil$ sampling points in the form of $S_{1,2^t}, S_{1+2^t,2^t+2^t}, \cdots, S_{1+k\cdot 2^t,2^t+k\cdot 2^t}, \cdots S_{n-2^t+(n\%2^t),n}$. We launch $\lceil \lceil n/2^t \rceil /2 \rceil$ threads st_h, $h = 1, 2, \cdots, \lceil \lceil n/2^t \rceil /2 \rceil$, each of which is used to integrate the sampling points $S_{1+2h\cdot 2^t,(2h+1)\cdot 2^t}$ and $S_{1+(2h+1)\cdot 2^t,(2h+2)\cdot 2^t}$. For illustration, we show the parallel per-ray data integration process in Figure 4.2 using the proposed PPRI method for 8 sampling points. The pseudocode of this method is provided in Algorithm 22.

4.3.2 Real-Time *over* Operator for Online Image Composition

To best accommodate the considered scenario here, we make necessary adaption the proposed algorithm in section. We design a Parallel Real-Time *over* Operator (PRTO) for parallel real-time composition and define a $FragCompInfo\{\}$ structure, as shown in Figure 4.3, for storing the composition information of each fragment throughout the parallel image composition process. Given an operand (a fragment) P, its $FragCompInfo\{\}$ instance consists of the following items: a float array C

$$\begin{tikzcd}
S_1 & \oplus \arrow[d] & S_2 & S_3 & \oplus \arrow[d] & S_4 & S_5 & \oplus \arrow[d] & S_6 & S_7 & \oplus \arrow[d] & S_8 \\
 & S_{1,2} & \oplus \arrow[d] & & S_{3,4} & & & S_{5,6} & \oplus \arrow[d] & & S_{7,8} & \\
 & & S_{1,4} \arrow[rrr] & & & \oplus \arrow[d] & & & S_{5,8} \arrow[lll] & & & \\
 & & & & & S_{1,8} & & & & & &
\end{tikzcd}$$

Figure 4.2 Illustration of the proposed parallel $\oplus$ operation for per-ray integration with 8 sampling points.

Function 22 PPRI($RStart, REnd, B_k, s$)
Input: the ray's first intersecting point $RStart$ with the volume, the ray's exiting point $REnd$ from the volume, a given data block B_k, and the number s of sampling points
Output: the integration result S_0 of all sampling points

```
1: for all $g \in [0, s)$ simultaneously do
2:     $C_g = RStart + g \cdot (REnd - RStart)/s$;
3:     $S_g = Sampling(C_g, B_k)$;
4: for $t \in [0, \ln s)$ do
5:     if ($Ztest(S_0) == true$) then
6:         for all $g \in [0, 2^{(\ln s)-t})$ simultaneously do
7:             $S_g = S_g \oplus S_{g+2^t}$;
8:     else
9:         exit;
10: return $S_0$;
```

```
struct FragCompInfo
{
    FragCompInfo( ): W(1.0), pNext(NULL){ }
    float   C[3];
    float   D;
    float   W;
    float   α;
    FragCompInfo *pNext;
};
```

Figure 4.3 The $FragCompInfo\{\}$ data structure for storing the composition information of each fragment.

that stores P's 3 color components in the order of R, G, B, a float variable α that stores P's α channel, a float variable D for P's actual depth value, a float variable W that stores P's accumulated weight factor from its nearer operands and is initialized to be 1.0, and a pointer $pNext$ that points to the $FragComInfo\{\}$ instance of its next farther fragment that has arrived. By default, $pNext$ is initialized to be $NULL$, indicating that no farther fragment exists among the fragments that have already arrived.

Given an array $FCI[n]$ of $FragCompInfo\{\}$, whose element $FCI[j]$ corresponds to the j-th arriving operand P_j, we divide the operations triggered by the arrival of a new operand into two groups: 1) operations applied to the $FragCompInfo\{\}$ instances of the existing operands, and 2) operations applied to the $FragCompInfo\{\}$ instance of the newly arriving operand. We apply a different parallelization strategy to each group. Parallelizing the operations in the first group is straightforward as they are independent of each other. However, parallelizing the operations in the second group is more complicated. For the j-th arriving operand P_j, we set $FCI[j].W$ to be $(1 - FCI[i].\alpha) \cdot FCI[i].W$, where $FCI[i]$ is the immediate nearer (preceding) operand of P_j that has arrived, i.e., $FCI[i].D \leq FCI[j].D$ and $^*(FCI[i].pNext).D \geq FCI[j].D$. Also, $FCI[i].W$ stores the weight factors from all the existing nearer operands of $FCI[i]$, and $(1-FCI[i].\alpha)\cdot FCI[i].W$ stores the weight factors from all the existing nearer operands of P_j. Note that in this composition process, we only need to perform one comparison in each of the threads created for all existing operands to decide the position of a new operand. The pseudocode of the PRTO operator is provided in Algorithm 23.

Function 23 PRTO($FCI[\,]$, i)
Input: an array $FCI[\,]$ of $FragCompInfo\{\}$ instances of the existing fragments at the same position as the newly arriving fragment in the image space, the index i of the newly arriving fragment in $FCI[]$
Output: updated $FCI[\,]$

```
1: for all threads $ct_j, j \in [0, i-1]$ simultaneously do
2:   if ($FCI[j].D > FCI[i].D$) then
3:     $FCI[j].W = FCI[j].W \cdot (1 - FCI[i].\alpha)$;
4:   else
5:     if ($*(FCI[j].pNext).D > FCI[i].D$ and $FCI[j].D < FCI[i].D$) then
6:       $FCI[i].pNext = FCI[j].pNext$;
7:       $FCI[i].W = (1 - FCI[j].\alpha) \cdot FCI[j].W$;
8:       $FCI[j].pNext = \&FCI[i]$;
```

4.3.3 A Tile-based Visualization Pipeline

Pipelining Rendering and Composition Traditionally, image rendering and image composition are treated as two sequentially executed stages. In order to eliminate the waiting time between them, we propose to overlap these two stages by dividing the image area into n equal-sized tiles t_i, $i = 0, 1, \cdots, n-1$, as shown in Figure 4.1. A fragment (x, y), $0 \leq x < w$, $0 \leq y < h$, belongs to tile t_i, $i = \lfloor y/n \rfloor$. Rendering and composition are performed on a tile-by-tile basis. We provide in Figure 4.4 an example to illustrate the differences between the pipeline in [11] and the proposed one.

To best utilize computing resources, we create multiple composition-rendering pipelines: 1) the composition unit c_j, $0 \leq j < n$, only composites fragments belonging to tile t_j; 2) to align composition with rendering, rendering unit r_i renders and sends tiles $t_{(i+k)\%n}$, $k = 0, 1, \cdots, n-1$, to the corresponding composition unit.

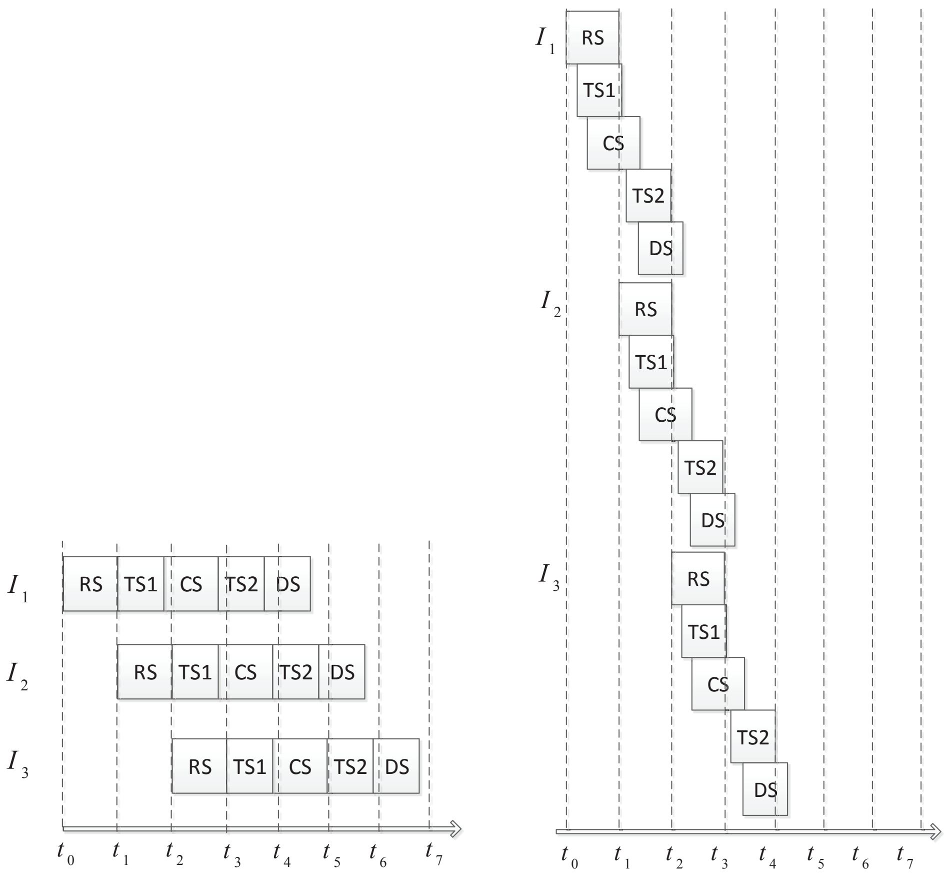


Figure 4.4 An illustration of the pipeline in [11] (left) and the proposed one (right), where "RS" represents a rendering step, "TS1" represents a data transfer from a rendering unit to a composition unit, "CS" represents a composition step, "TS2" represents a data transfer from a composition unit to a display unit, and "DS" represents a display step.

The benefits of the proposed tile-by-tile pipelining scheme are multifold as follows: 1) it overlaps rendering and composition, hence eliminating the waiting time between them; 2) it only needs to store one tile in each intermediate image on the composition unit, hence minimizing the memory use; 3) similarly, it minimizes the required buffer space on the rendering unit.

Algorithm Design We present the proposed visualization pipeline in Algorithm 24 and 25 for rendering and composition, respectively. In Algorithm 24, the data type *Pix* in Line 2 denotes the data structure for storing a RGBA-formatted fragment, which is the same as the structure of the sampling point S in Section 4.3.1. The *Pix* array "*F_Tiles*" is used to store rendered tiles on each rendering unit. The entire rendering process on each unit takes n steps, each of which is further divided into two tasks of tile rendering and sending. At each step, each rendering unit first calculates the scope of the "to-be-rendered" tile as shown in Lines 4 and 5, and then performs a per-ray data integration procedure for each fragment in the tile. The raycasting procedure calls VolIntersect() to calculate the ray's first intersecting point *RStart* with the data block and the exiting point *REnd* from the data block, and then employs the proposed PPRI method (Algorithm 22) to integrate the sampling points along the ray within the data block. The tile sending task follows immediately, as shown in Line 10, once the rendering task is completed.

In Algorithm 25, we define a 2D array $F\_Tile[w, h/n]$ of *Pix* in Line 4 to store the final composited tile, a 1D array $Frags[w \times h]$ to store the $FragCompInfo$ instances for the fragments within the tile composited by c_j, an array $FCI[\ ]$ to temporarily store the $FragCompInfo$ instances for all the fragments with the same coordinates in the image space, and an array *Wt_Frag* to temporarily store the color components of all the fragments with the same coordinates in the image space. The entire composition procedure consists of three stages: 1) Tile receiving and updating in Lines 6-14. Upon the arrival of a new tile, we initialize a corresponding region in $Frags$, and then employ the PRTO method to update the corresponding regions in $Frags$ for all the existing and the newly arriving tiles on a fragment-by-fragment basis. 2) Tile finalizing in Lines 15-16. After all the tiles have arrived and been updated, we compute the final color component values of all the fragments within $Frags$. 3) Pixel generation in Lines 17-22. We generate each pixel in the final image by adding up the final color component values of all the fragments with the same coordinates in parallel using a binary-tree structure, as shown in Lines 20 and 21 where Algorithm 5 is called.

Function 24 Rendering($B_0, B_1, \cdots, B_{n-1}, w, h, VP$)
Input: data blocks $B_0, B_1, \cdots, B_{n-1}$, width w and height h of the image, view point VP
Output: a sequence of rendered tiles in the intermediate images

```
1: for all $r_i \in [0, n-1]$ simultaneously do
2:    $Pix\ F\_Tiles[w][h]$;
3:    for $k = 0$ to $n-1$ do
4:       $ystart = (i+k)\%n \cdot (h/n)$;
5:       $yend = (i+k+1)\%n \cdot (h/n)$;
6:       for $cx = 0$ to $w$ do
7:          for $cy = ystart$ to $yend$ do
8:             $RStart, REnd$ = VolIntersect($VP, cx, cy, B_k$);
9:             $F\_Tiles[cx, cy]$ = PPRI($RStart, REnd, B_k, S$);
10:      send the tile indexed by $ystart$ and $yend$ in $F\_Tiles$ to $c_{(i+k)\%n}$;
```

Performance Analysis We conduct theoretical performance analysis of the existing and proposed visualization pipelines. For the traditional pipeline, we follow the main modeling framework in [11] and have the following three assumptions: 1) There is no bandwidth sharing between different rendering machines. 2) The time cost of rendering is strictly linear with respect to the size of the bounding box where the partitioned data block is projected. 3) All bounding boxes are of the same size as the display image. The time cost T_{old} of the traditional visualization pipeline in [11]

Function 25 Composition(w, h, n)
Input: the width w and height h of the image, the number n of composition units
Output: the composited tile of the final image on each composition unit

```
 1: for all $c_j, j \in [0, n-1]$ simultaneously do
 2:     $Pix\ F\_Tile[w][h/n]$;
 3:     $FragCompInfo\ Frags[w \times h]$;
 4:     $FragCompInfo\ FCI[n]$;
 5:     $Pix\ Wt\_Frag[n]$;
 6:     for $i = 0$ to $n-1$ do
 7:         receive one tile from $r_{(j+n-g)\%n}$ and store it in $F\_Tile$;
 8:         $ystart = (i+j)\%n \cdot (h/n)$;
 9:         $yend = (i+j+1)\%n \cdot (h/n)$;
10:         for $k = ystart \cdot w$ to $yend \cdot w$ do
11:             Initialize($Frags[k]$);
12:         for $cx = 0$ to $w$ do
13:             for $cy = ystart$ to $yend$ do
14:                 $Frag\_Ind = cy \cdot w + cx$;
15:                 for $k = 0$ to $i$ do
16:                     $FCI[k] = Frags[cx + i \cdot \frac{w \cdot h}{n}]$;
17:                 PRTO($FCI, i$);
18:     for $k = 0$ to $w \cdot h$ do
19:         $Frags[k].C = Frags[k].C \cdot Frags[k].W \cdot Frags[k].\alpha$;
20:     for $tx = 0$ to $w$ do
21:         for $ty = 0$ to $h/n$ do
22:             for $k = 0$ to $n$ do
23:                 $Wt\_Frag[k] = Frags[cx + k \cdot \frac{w \cdot h}{n}].C$;
24:             for $t = 0$ to $\ln n$ do
25:                 ParaSum($Wt\_Frag, t, n$);
26:             $F\_Tile[tx, ty]$=$Wt\_Frag[0]$;
27: return  the final image tile $F\_Tile$;
```

Function 26 ParaSum(Wt_Frag, t, n)
Input: a weighted fragment array Wt_Frag, an integer t, the number n of tiles to composite;
Output: updated Wt_Frag from the addition of a pair of elements;

```
1: for all $ct_{g}, g \in [0, 2^{(\ln acnt)-t}]$ simultaneously do
2:     $Wt\_Frag[g].C = Wt\_Frag[g].C \oplus Wt\_Frag[g + 2^t].C$;
```

is as follows:

$$T_{old} = T_r + T_{t_1} + T_c + T_{t_2} + T_d, \tag{4.3}$$

where T_r is the time cost of rendering the assigned data block using the traditional raycasting method, T_{t_1} is the time cost of sending an intermediate image from a rendering unit to a composition unit, T_c is the time cost of compositing all the tiles on each composition unit, T_{t_2} is the time cost of sending one composited tile from a composition unit to a display unit.

For the proposed pipeline, we calculate the time cost as

$$T_{new} = T_{r'} + T_{t_1'} + T_{c'} + (n-1) \cdot max\{T_{r'}, T_{t_1}, T_{c'}\} + T_{t_2} + T_d, \tag{4.4}$$

where $T_{r'}$ is the time cost of rendering one tile using the proposed raycasting method, $T_{t_1'}$ is the time cost of sending one intermediate image tile from a rendering unit to a composition unit, $T_{r'}$ is the time cost of compositing one tile using the proposed composition operator, $T_{t_2'}$ is the time cost of sending one composited tile from a compositing unit to a display unit, and n is the number of divided tiles. Comparing (4.3) and(4.4), since $n \cdot T_{r'} < T_r$, $n \cdot T_{t_1'} < T_{t_1}$, and $n \cdot T_{c'} < T_c$, it is obvious that $T_{new} < T_{old}$.

4.3.4 Implementation and Experimental Results

To evaluate the performance of the proposed V4BD architecture, we implement the component techniques on a high-performance visualization cluster and compare them with existing methods. The visualization cluster consists of 1 head node and 16 compute nodes, each of which is equipped with one GTX580 GPU, dual 6-core processors at the speed of 2.3 GHz for each core and 64GB RAM, and is connected

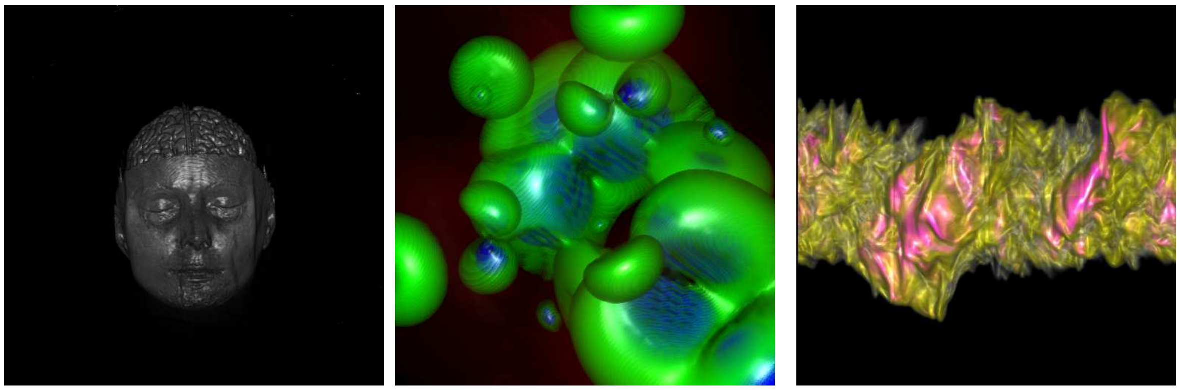


Figure 4.5 Composited images of Human Brain, HIPIP and Jet Ejection.

through a 1GigE switch. We assign half of them as rendering nodes and the other half as composition nodes.

We use V4BD to visualize three real-life scientific datasets from different domains: 3D Human Brain, High-Potential Iron Protein (HIPIP), and Jet Ejection. We plot a representative final image for each of them in Figure 4.5, which confirms the validity of the proposed methods and the correctness of our implementations.

We use the 3D Human Brain data as a benchmark dataset for performance comparison. For the proposed parallel rendering method, we assign the entire raw data to a single node and blend different numbers of sampling points within the assigned data block. The performance comparison with the existing rendering method is plotted in Figure 4.6, where we observe that the performance superiority of the proposed method over the existing one becomes more obvious as the number of sampling points increases. With a small number of sampling points, the existing method may outperform the proposed one because of the overheads for thread spawning and synchronization, which could be reduced by launching threads in groups for different rays and optimizing synchronization timings.

For the real-time *over* operator for image composition, we consider: 1) different numbers of input images: 8, 16, and 32; 2) different image arriving intervals: 0s and 0.1s; and 3) 7 different image arriving (i.e., availability) orders, in each sequence of

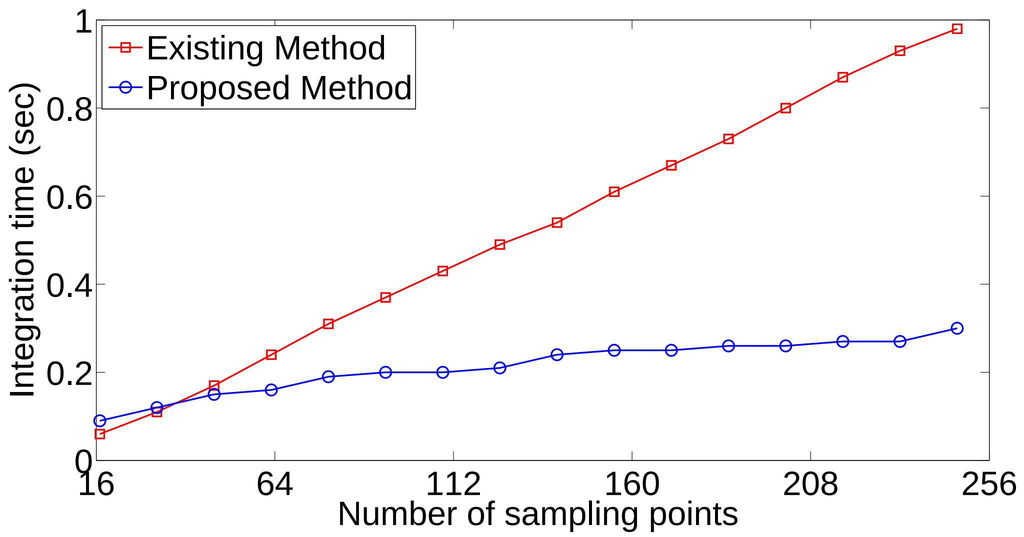


Figure 4.6 Performance comparison between the proposed and existing raycasting methods.

input images. We define the k-distance arriving order as an arriving order where the difference between the blending orders of any two subsequently arriving input images is k. In the experiments, the seven different arriving orders are generated as follows. Given n operands (i.e., intermediate fragments), at time t, when $k \neq 2$, their arriving order is generated as

$$F(k,t,n) = \begin{cases} (t \bmod \lceil \frac{n}{k} \rceil) \cdot k + \lfloor t / \lceil \frac{n}{k} \rceil \rfloor \\ \text{if } 0 \leq t < (n \bmod k) \cdot \lceil \frac{n}{k} \rceil; \\ (t \bmod \lfloor \frac{n}{k} \rfloor) \cdot k + n - 1 - \lfloor (n-1-t) / \lfloor \frac{n}{k} \rfloor \rfloor \\ \text{if } (n \bmod k) \cdot \lceil \frac{n}{k} \rceil \leq t < n; \end{cases} \tag{4.5}$$

where $k \in [1, 3, 4, 5, 6, 7]$, $n \in [8, 16, 32]$, and $t \in [0, n)$. With $k = 2$, the arriving order of the operands is generated as

$$F(k,t,n) = \begin{cases} (t \bmod \lceil \frac{n}{k} \rceil) \cdot k + 1 \\ \text{if } 0 \leq t < (n \bmod k) \cdot \lceil \frac{n}{k} \rceil, \\ (t \bmod \lfloor \frac{n}{k} \rfloor) \cdot k \\ \text{if } (n \bmod k) \cdot \lceil \frac{n}{k} \rceil \leq t < n. \end{cases} \tag{4.6}$$

We plot the performance measurements in Figure 4.7, which shows that given the same arriving interval and the same number of input images of the same size, but different arriving orders, both algorithms exhibit relatively stable performance, implying that both of the operators are immune to the arriving order. For the traditional operator, since it must wait for all the operands to become available, the arriving order does not affect the time cost; while for the proposed operator, whichever operand arrives, the composition is always performed in constant time in parallel, hence resulting in the identical total composition time. The proposed operator takes consistently much less time than the original one in all the cases, which confirms our theoretical analysis. We further observe that the performance differences become more obvious as the image size and the number of input images increase, as shown in each subfigure (from a to h) of Figure 4.7.

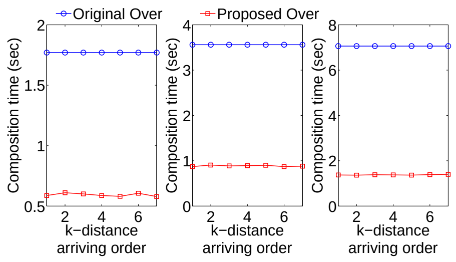
Furthermore, in Figure 4.7a and Figure 4.7b, as the arrival interval increases from 0s to 0.1s, the proposed operator's relative performance gain over the original one remains quite stable, i.e., the gap between two performance curves does not vary much, which is justified by Equations (2.40) and (2.41). For the proposed operator, the time to perform all the computations for a newly arriving operand is constant. As the arriving interval increases, such computing time becomes negligible, and (2.41) is dominated by the other two time cost components, i.e.,

$$T_S = \sum_{i=1}^{n-1} t_i + t_{sum}. \tag{4.7}$$

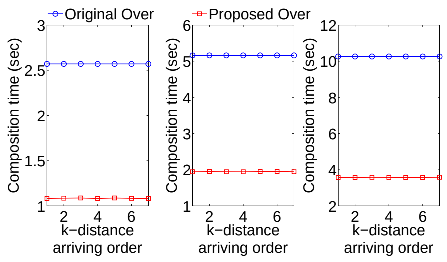
Comparing (4.7) with (2.40), the difference only lies in t_{sum} and $t_{srt} + 4(n-1) \cdot t_c$, which are all fixed given the same number of input images and the same image size. Thus, the performance difference between these two operators essentially arises from the difference of these two terms as the arriving interval increases.

We also compare these two *over* operators on the image size of 2048^2 and 3072^2 using the 3D brain dataset with varying arriving intervals and numbers of input images arriving in the pre-sorted order, and plot their performance measurements in (4.8). We observe that with such an "ideal" arriving order, the proposed operator still outperforms the original one.

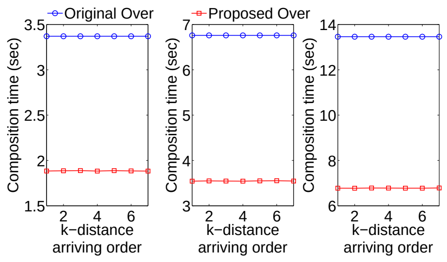
We also conduct experiments to evaluate the performance of the proposed V4BD architecture. We consider three scenarios: i) visualization with the pipelining structure using the proposed rendering and composition methods, ii) visualization with the pipelining structure using only the proposed composition method, and iii) visualization using the traditional rendering and composition methods. In the same computing environment, we execute the above three visualization scenarios with different image sizes and plot the results in Fig. 4.9, which shows the performance gains brought by the proposed image composition operator, per-ray integration method, and pipelined scheme, respectively.



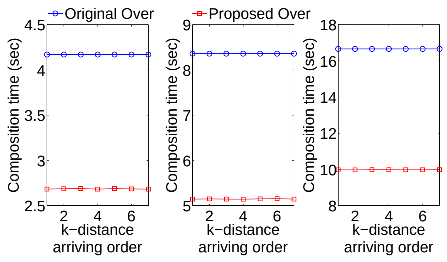
(a) arriving interval 0s for image size of 2048^2.



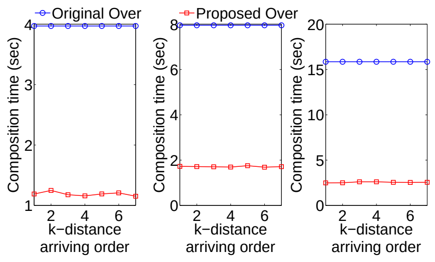
(b) arriving interval 0.1s for image size of 2048^2.



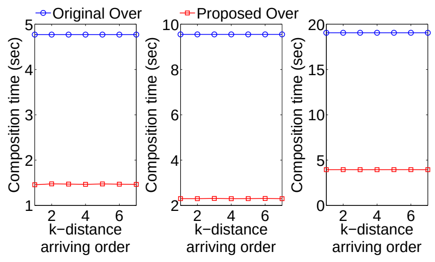
(c) arriving interval 0s for image size of 2048^2.



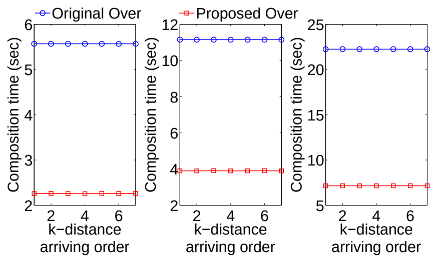
(d) arriving interval 0.1s for image size of 2048^2.



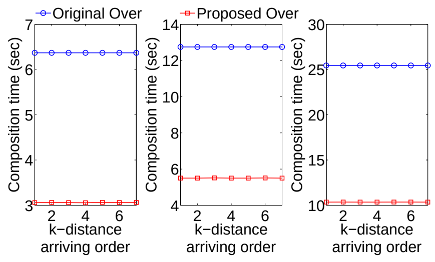
(e) arriving interval 0s for image size of 3072^2.



(f) arriving interval 0.1s for image size of 3072^2.

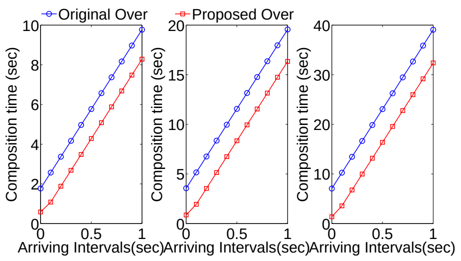


(g) arriving interval 0s for image size of 3072^2.

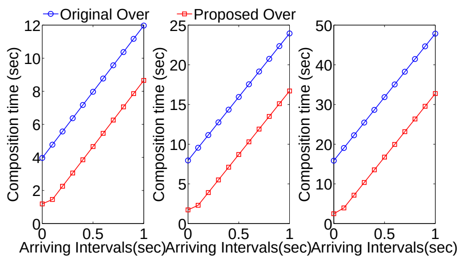


(h) arriving interval 0.1s for image size of 3072^2.

Figure 4.7 Comparisons of two *over* operators on the image size of 2048^2 in subfigures (a,b,c,d) and 3072^2 in subfigures (e,f,g,h) using the 3D brain dataset with varying arriving intervals and numbers of input images. The three figures in each subfigure correspond to the cases of 8, 16, and 32 input images, respectively, from left to right.



(a) Image size of 2048^2.



(b) Image size of 3072^2.

Figure 4.8 Comparisons of two *over* operators on the image size of 2048^2 in subfigure (a) and 3072^2 in subfigure (b) using the 3D brain dataset with varying arriving intervals and numbers of input images arriving in an "ideal" order. The three figures in subfigures (a) and (b) correspond to the cases of 8, 16, and 32 input images, respectively, from left to right.

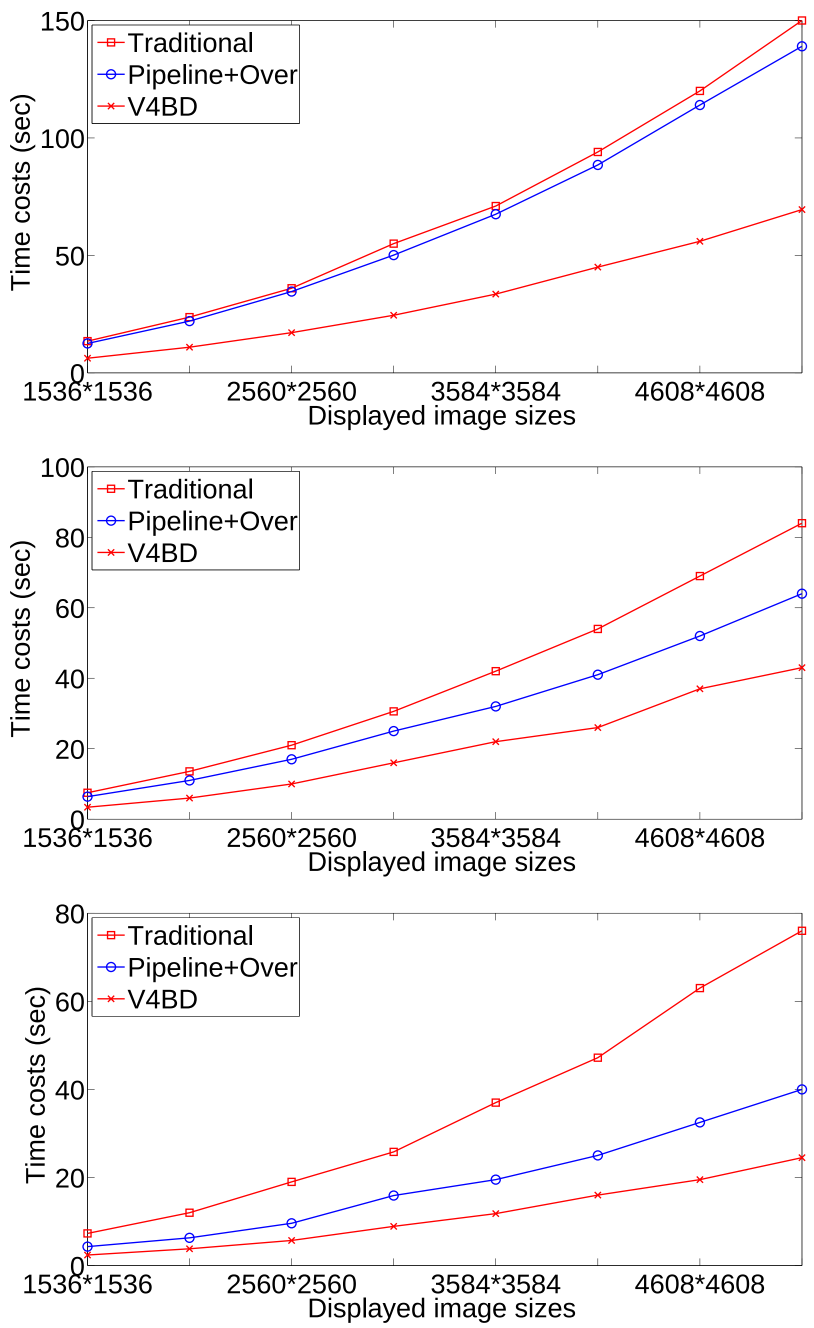


Figure 4.9 Comparison of the proposed and existing visualization pipeline with 8, 16, and 32 rendering/composition nodes, where "Traditional" represents the traditional sort-last pipeline, "Pipeline+*over*" represents the tile-based pipelining structure using only the proposed *over* operator, and "V4BD" represents the tile-based pipelining structure using the proposed rendering and composition methods.

CHAPTER 5

QR DECOMPOSITION

It has been a long history since QR was first proposed [63, 26]. Along its way of development during the past over one hundred years, it constantly attract attentions from researchers for its construction of orthogonal bases for the input matrix and the benefiting to other linear algebraic problems. Such long-term and large-scale focus place QR decomposition at a important role in nowadays scientific computing, also result in huge number of works targeting at different aspects of the decomposition problem, such as decomposition method itself, time efficiency or stability studying of one given decomposition method, application of the decomposition results and so on. The application of QR decomposition is briefly covered in the chapter of Introduction, we devote our research efforts here mainly to different decomposition methods and their time efficiency, stability analysis, especially in the context of parallelization. A brief survey of existing mainstream QR decomposition methods and their corresponding analytics are given as follows.

5.0.5 Existing Mainstream Sequential QR Decomposition Methods

We enumerate 5 existing mainstream methods in sequential scenario here: Classical Gram-Schmidt(CGS) method, as in Algorithm 27, Modified Gram-Schmidt(MGS) method, as in Alg. 28, HouseHolder method as in Alg. 29, Givens method as in Algorithm 30, Choelsky-based factorization as in Algorithm 31. Comparing these 5 methods, they are of the same order of time complexity(efficiency), differing on stability, where HouseHolder has strongest stability, MGS method follows, Choelsky-based method follows and CGS comes last. Details of these analytics are given in Table 5.1.

Function 27 Classical Gram-Schmidt(a_1,a_2, $\cdots$, a_n)
Input:n independent vectors a_1,a_2, $\cdots$, a_n
Output: Normalized orthogonal matrix $Q = (q_1,q_2, \cdots , q_n)$ and upper triangular matrix R, where $A = Q \cdot R$

```
1: for $j = 1$ to $n$ do
2:     $v_j = a_j$;
3:     for $i = 1$ to $j - 1$ do
4:         $r_{ij} = q_i^T \cdot a_j$;
5:         $v_j = v_j - r_{ij} q_i$;
6:     $r_{ij} = \|v_j\|_2$;
7:     $q_j = v_j / r_{jj}$;
```

Function 28 Modified Gram-Schmit(a_1,a_2, $\cdots$, a_n)
Input:n independent vectors a_1,a_2, $\cdots$, a_n
Output: Normalized orthogonal matrix $Q = (q_1,q_2, \cdots , q_n)$ and upper triangular matrix R, where $A = Q \cdot R$

```
1: for $j = 1$ to $n$ do
2:     $v_j = a_j$;
3: for $i = 1$ to $n$ do
4:     $r_{ii} = \|v_i\|_2$;
5:     $q_i = v_i / r_{ii}$;
6:     for $j = i + 1$ to $n$ do
7:         $r_{ij} = q_i^T \cdot v_j$;
8:         $v_j = v_j - r_{ij} q_i$;
9: for $i = 1$ to $n$ do
10:     for $k = i$ to $n$ do
11:         $R(i, k) = a_i \cdot q_k$;
```

Function 29 Householder(a_1,a_2, $\cdots$, a_n)
Input:n independent vectors a_1,a_2, $\cdots$, a_n
Output: Normalized orthogonal matrix $Q = (q_1,q_2,\cdots,q_n)$ and upper triangular matrix R, where $A = Q \cdot R$

```
 1: $Q = I_n$;[a]
 2: $R = Z_{n,n}$;[b]
 3: for $i = 1$ to $n$ do
 4:    $\sigma_i = \|E_{i-1} \cdot a_i\|$;[c]
 5:    $v_i = E_{i-1} \cdot a_i - \sigma_i e_i$;[d]
 6:    $H_i = I_n - \frac{2}{v_i^T v_i} v_i v_i^T$;
 7:    $A = H_i \cdot A$;
 8:    $Q = Q \cdot H_i$;
 9: for $j = 1$ to $n$ do
10:    for $k = j$ to $n$ do
11:       $R(j,k) = a_j \cdot q_k$;
```

[a] $I_n = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix}_{n\times n}$

[b] $Z_{n,n} = \begin{bmatrix} 0 & \cdots & 0 \\ \vdots & \vdots & \vdots \\ 0 & \cdots & 0 \end{bmatrix}_{n\times n}$

[c] $E_n^i = \begin{bmatrix} Z_{i,i} & Z_{i,n-i} \\ Z_{n-i,i} & I_{n-i} \end{bmatrix}_{n,n}$

[d] $e_i = \begin{array}{ccccc} [0 & \cdots & 1 & \cdots & 0]^T \\ 1 & \cdots & \text{i} & \cdots & \text{n} \end{array}$

Function 30 Givens_QR(a_1,a_2, $\cdots$, a_n)
Input:n independent vectors a_1,a_2, $\cdots$, a_n
Output: Normalized orthogonal matrix $Q = (q_1,q_2, \cdots , q_n)$ and upper triangular matrix R, where $A = Q \cdot R$

```
1:  $Q = I_n$;
2:  $R = A$;
3:  for $j = 1$ to $n$ do
4:      for $i = m : -1 : j + 1$ do
5:          $r = \sqrt{A(i-1, j)^2 + A(i, j)^2}$;
6:          $c = A(i-1, j)/r$;
7:          $s = -A(i, j)/r$;
8:          Create $G(i-1, i, j)$;[a]
9:          $Q = QG(i-1, i, j)$;
10:         $R = G^T(i-1, i, j)R$;
```

[a] $G(i-1, i, j) = \begin{bmatrix} 1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \cdots & c & \cdots & -s & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \cdots & s & \cdots & c & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & 1 \end{bmatrix}$ where $c = \frac{A(i-1,j)}{\sqrt{A(i-1,j)^2+A(i,j)^2}}, s = \frac{A(i-1,j)}{\sqrt{A(i-1,j)^2+A(i,j)^2}}$ appear at the intersections ith and $i-1$th row, other elements are determined by $g_{k,k} = 1$, for $k \neq i, i-1$, $g_{i,i} = c$, $g_{i-1,i-1} = c$, $g_{i-1,i} = -s$, $g_{i,i-1} = s$.

Table 5.1 Stability Analysis of Four Methods in Comparison

Algorithm	$\|I - Q^TQ\|_2$ bound	Assumption on $\kappa(A)$[a]	References
Householder QR	$O(\epsilon)$	None	[25]
MGS	$O(\epsilon\kappa(A))$	None	[6]
CholeskyQR	$O(\epsilon\kappa(A)^2)$	None	[65]
CGS	$O(\epsilon\kappa(A)^{n-1})$	None	[64, 33]

[a] $\kappa(A) = \|A\|_2\|A^{-1}\|_2$ as in [29]

$proj_u(a) = \frac{<a,u>}{<u,u>}u$

$$\begin{aligned}
u_1 &= a_1 & e_1 &= \frac{u_1}{\|u_1\|} \\
u_2 &= a_2 - proj_{u_1}(a_2) & e_2 &= \frac{u_2}{\|u_2\|} \\
u_3 &= a_3 - proj_{u_1}(a_3) - proj_{u_2}(a_3) & e_3 &= \frac{u_3}{\|u_3\|} \\
u_4 &= a_4 - proj_{u_1}(a_4) - proj_{u_2}(a_4) - proj_{u_3}(a_4) & e_4 &= \frac{u_4}{\|u_4\|} \\
&\vdots \\
u_k &= a_k - \sum_{j=1}^{k-1} proj_{u_j}(a_k) & e_k &= \frac{u_k}{\|u_k\|}
\end{aligned} \tag{5.1}$$

$$Q = [e_1, e_2, \cdots, e_n], R = \begin{pmatrix} < e_1, a_1 > & < e_1, a_2 > & < e_1, a_3 > & \cdots \\ 0 & < e_2, a_2 > & < e_2, a_3 > & \cdots \\ 0 & 0 & < e_3, a_3 > & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix}$$

Function 31 CholeskyQR(a_1,a_2, $\cdots$, a_n)
Input:n independent vectors a_1,a_2, $\cdots$, a_n
Output: Normalized orthogonal matrix $Q = (q_1,q_2, \cdots, q_n)$ and upper triangular matrix R, where $A = Q \cdot R$

```
1: $W = A^TA$;
2: Compute the Cholesky factorization $L \cdot L^T$ of $W$ as in Algorithm 32;
3: $Q = AL^{-T}$;
4: $R = L$;
```

Function 32 CholeskyDecomposition(a_1,a_2, $\cdots$, a_n)
Input:n independent vectors a_1,a_2, $\cdots$, a_n
Output: Cholesky factorization $L \cdot L^T$ such that $A = L \cdot L^T$

```
1: for $i = 1 : n$ do
2:   for $j = 1 : i$ do
3:     if i==j then
4:       $sum = 0$;
5:       for $k = 1 : j$ do
6:         $sum+ = a[j, k] * a[j, k]$;
7:       $L[j, j] = \sqrt{a[j, j] - sum}$;
8:     else
9:       $sum = 0$;
10:      for $k = 1 : j$ do
11:        $sum+ = a[i, k] * a[j, k]$;
12:      $L[i, j] = \frac{1.0}{L[j,j]} \sqrt{a[i, j] - sum}$;
```

5.1 Parallelization for QR Decomposition

Parallelization here are oriented to the few previously mentioned sequential methods. Unlike a single computing unit within consideration, there requires more complex underlying architecture supporting such procedure. We first illuminate all requisite supportive components during such procedure, then list multiple parallelization instances of the mainstream sequential methods.

5.1.1 Requisite Components in Parallel QR Factorization

Data Partition There are two commonly used methods, row-wise partitioning, column-wise partitioning, block-wise partitioning, block-wise cyclic partitioning and so on. We provide definition of row-wise partitioning and block-wise cyclic partitioning as follows. The others can be derived in a similar way.

Definition 5.1.1. *Given $P(\leq n)$ processors, denoted ρ_0,ρ_1, $\cdots$, ρ_{p-1} and an integer β where $1 \leq \beta \leq n/P$, a one-dimensional row-wise partitioning of panel size β, is a partitioning where for all i,$0 \leq i < P$, ρ_i is assigned row $k\beta P + i\beta + q$, for all $0 \leq k \leq \frac{n}{\beta P}$ and $1 \leq q \leq \beta$.*

Definition 5.1.2. *Given $P(\leq n^2)$ processors, denoted $\rho_{i,j}$,where $1 \leq i, j \leq \sqrt{P}$ and an integer G where $1 \leq G \leq n/\sqrt{P}$, a two-dimensional block cyclic partitioning of granularity size G assigns to each processor $\rho_{i,j}$ the matrix items $a_{l,m}$ where $\frac{n}{G}k +$*

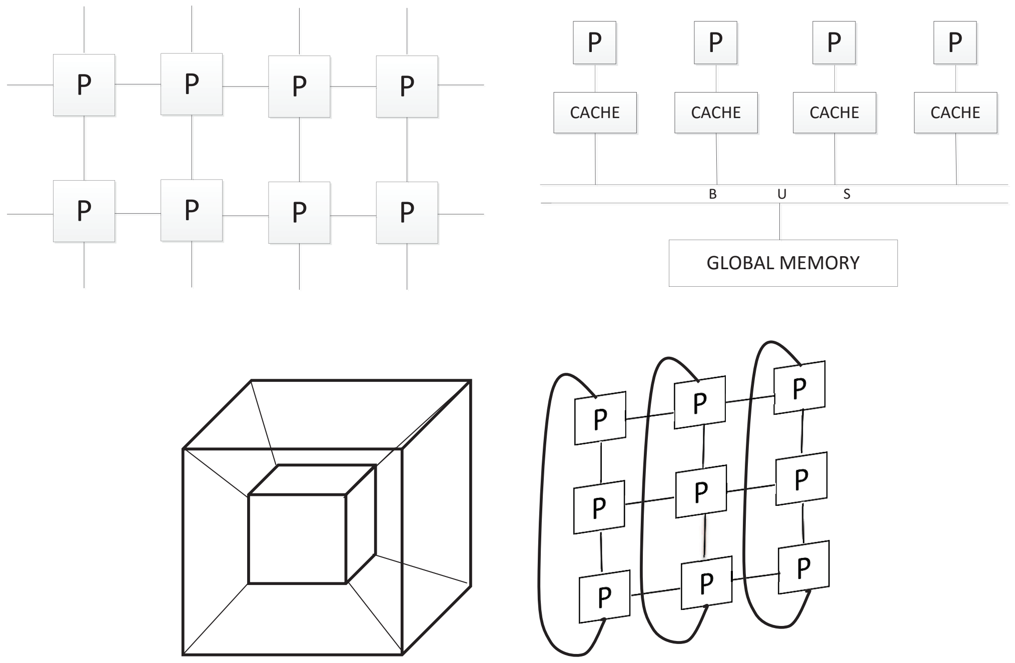


Figure 5.1 Commonly-used parallel processor architecture.

$\frac{n}{G\sqrt{P}}(i-1)+1 \leq l \leq \frac{n}{G}k + \frac{n}{G\sqrt{P}}i, \frac{n}{G}k + \frac{n}{G\sqrt{P}}(j-1)+1 \leq m \leq \frac{n}{G}k + \frac{n}{G\sqrt{P}}j$ *where* $0 \leq k < G$.

Topology of Computing Units and Communication Models The paradigm of parallel computing units varies a lot. We list a few commonly used ones here, such as Cellular Automata, pRAMS, Hypercubes, Meshes, Butterflies, sorting networks, and further provide illustrating figures for selected ones of them in Figure 5.1.

Given a specific topology of parallel computing units, we consider the LogP model [14] to be the responsive communication model, which is defined as follows:

- L: Latency or delay incurred in communicating a message containing a word from the source processor to the target one.
- o: Overhead, the length of time that a processor is either transmitting or receiving a message. During this time, the processor cannot perform other operations;

- g: the gap defined as the minimum time interval between consecutive message sending or consecutive message receiving at a processor;
- P: number of processor/memory pairs running in parallel.

Note that there is an upper bound for numbers of messages, L/g, to send/receive through the network.

5.1.2 Parallelized QR Decomposition Methods and Their Performances

Based on the above platform of parallelization, we did a survey for each sequential method with different data partitioning schemes and computing unit topologies. For CGS method, there exist parallelization instances following row-wise column wise partitioning in [70], running on topology of MIMD in [46] and pRAMS in [70]. For MGS method, there exist parallelization instances following the column-wise partitioning in [52], the row-wise partitioning in [8], block-wise partitioning in [66], running on topology of hypercube and ring in [51]. For HouseHolder method, there exist parallelization instances following the column partition in [58], the row and block partition in [51], running on topology of ring, hypercube in [51], and torus in [19]. For Givens method, there exist parallelization instances following the block partition in [17], row-wise partition in [22], running on topology of shared-memory machine in [61][12], torus in [19], ring in [50], hypercube in [57]. For Cholesky based decomposition, there exist parallelization instances following the block and row partition in [62], running on topology of hypercube in [24] and ring in [53]. The most efficient parallelization across all possible data partition schemes and topologies for each individual sequential algorithm is given in Table 5.2.

5.2 Supportive Theorems for Parallel QR Decomposition

The performance of existing parallelized decomposition methods is still limited in terms of time efficiency and stability. To the best of our knowledge, there does not exist a single parallelization method that addresses both time efficiency and stability

Table 5.2 Latency Performance Analysis of Four Methods in Comparison[15]

Parallel Algorithm	Flops on critical path	Messages	Comm. volume
Householder QR	$\frac{3n^3}{P}$	$\log(P)$	$n^2 \log(P)$
MGS	$\frac{2n^3}{P}$	$\log(P)$	$\frac{n^2}{2} \log(P)$
CholeskyQR	$\frac{2n^3}{P} + \frac{n^3}{3}$	$\log(P)$	$\frac{n^2}{2} \log(P)$
CGS	$\frac{2n^3}{P}$	$\log(P)$	$\frac{n^2}{2} \log(P)$

issues. Towards this goal, we propose a new parallelized QR decomposition method. Currently, we have derived two theorems that pave a promising way to such a goal: one provides an explicit representation of each vector in the Q matrix generated by the HouseHolder method as in Theorem 5.2.2. and the other provides a new distinctive QR decomposition method as in Theorem 5.2.4. Proofs for these two theorems are as follows, where one key formula required in Theorem 5.2.2 is proved in Lemma 5.2.1 in advance for convenience of reference.

Lemma 5.2.1. *For the i-th row (column) of* $I - \frac{2}{v_i^T v_i} v_i v_i^T$, *i.e.,* $I - \frac{2}{v_i^T v_i} v_i v_i^T(:, i)$, *there exists the following equation:*

$$I - \frac{2}{v_i^T v_i} v_i v_i^T(:, i) = \frac{a_i^{(i)}}{\|a_i^{(i)}\|}, \tag{5.2}$$

$$\text{where } a_i^{(i)} = E_i(I - \frac{2}{v_{i-1}^T v_{i-1}} v_{i-1} v_{i-1}^T) \cdots (I - \frac{2}{v_1^T v_1} v_1 v_1^T) a_i, \ v_i = a_i^{(i)} - \gamma_i e_i,$$

$$\gamma_i = \|a_i^{(i)}\|_2, \ E_i = \begin{bmatrix} Z_{i,i} & Z_{i,n-i} \\ Z_{i,n-i}^T & I_{n-i} \end{bmatrix}, Z_{i,j} = \begin{bmatrix} 0 & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 \\ \vdots & & & \\ 0 & 0 & \cdots & 0 \end{bmatrix}_{i \times j} .$$

Proof.

$$\begin{aligned}
I - \frac{2}{v_i^T v_i} v_i v_i^T(:,i) &= e_i - \frac{(a_i^{(i)}[i] - \|a_i^{(i)}\|)(a_i^{(i)} - \|a_i^{(i)}\| e_i)}{\|a_i^{(i)}\|^2 - \|a_i^{(i)}\| \cdot a_i^{(i)}(i)} \\
&= e_i - \frac{(\|a_i^{(i)}\|^2 - \|a_i^{(i)}\| \cdot a_i^{(i)}(i)) e_i - (a_i^{(i)}[i] - \|a_i^{(i)}\|) a_i^{(i)}}{\|a_i^{(i)}\|^2 - \|a_i^{(i)}\| \cdot a_i^{(i)}(i)} \\
&= e_i - e_i + \frac{a_i^{(i)}}{\|a_i^{(i)}\|} = \frac{a_i^{(i)}}{\|a_i^{(i)}\|}
\end{aligned} \tag{5.3}$$

□

Theorem 5.2.2. *Matrix $Q = (q_1, \cdots, q_k \cdots, q_n)$, where q_k is calculated according to 5.4 is numerically equivalent to the counterpart generated from HouseHolder method.*

$$q_k = \frac{q_k^{'}}{\|q_k^{'}\|}, q_k^{'} = a_k - \sum_{j=1}^{k-1} < a_k^T \cdot q_j > q_j \tag{5.4}$$

Proof. According to the definition of Householder, we have

$$q_i = (I - \frac{2}{v_1^T v_1} v_1 v_1^T) \cdots (I - \frac{2}{v_{i-1}^T v_{i-1}} v_{i-1} v_{i-1}^T)(I - \frac{2}{v_i^T v_i} v_i v_i^T(:,i)). \tag{5.5}$$

Plugging (5.2) in (5.5) and setting $V_i = I - \frac{2}{v_i^T v_i} v_i v_i^T$, we have

$$\begin{aligned}
q_i &= V_1 \cdots V_{i-1} \cdot E_i \cdot V_{i-1} \cdots V_1 \cdot \frac{a_i}{\|a_i^{(i)}\|_2} \\
&= V_1 \cdots V_{i-1} \cdot V_{i-1} \cdots V_1 \cdot \frac{a_i}{\|a_i^{(i)}\|_2} + \sum_{j=1}^{i-1} V_1 \cdots V_{i-1} \cdot E_j^{'} \cdot V_{i-1} \cdots V_1 \cdot \frac{a_i}{\|a_i^{(i)}\|_2}
\end{aligned} \tag{5.6}$$

where

$$E_i = I_n - \sum_{j=1}^{i-1} E_j^{'}, (i \geq 2), E_i^{'} = \begin{bmatrix} Z_{i-1,n} \\ e_i \\ Z_{n-i,n} \end{bmatrix}.$$

There also exists that:

$$
\begin{aligned}
& V_1 \cdots V_{i-1} \cdot E_j^{'} \cdot V_{i-1} \cdots V_1 \cdot a_i \\
=& V_1 \cdots V_{i-1} \cdot E_j^{'} \cdot V_j \cdots V_1 \cdot a_i \\
=& V_1 \cdots V_{i-1} \cdot Q_j^{'} \cdot V_{j-1} \cdots V_1 \cdot a_i \\
=& V_1 \cdots V_{i-1} \cdot \begin{bmatrix} Z_{j-1,n} \\ q_j^T \\ Z_{n-j,n} \end{bmatrix} \cdot a_i \\
=& (q_j^T \cdot a_i) \cdot V_1 \cdots V_{i-1} \cdot e_j \\
=& (q_j^T \cdot a_i) \cdot V_1 \cdots V_{j-1} \cdot \frac{a_i^{(i)}}{\|a_i^{(i)}\|_2} \\
=& (q_j^T \cdot a_i) q_j
\end{aligned}
\tag{5.7}
$$

where

$$
Q_i^{'} = \begin{bmatrix} Z_{i-1,n} \\ I - \frac{2}{v_i^T v_i} v_i v_i^T (i,:) \\ Z_{n-i,n} \end{bmatrix}.
$$

Combining (5.6) and (5.7), the theorem is proved. □

Theorem 5.2.3. *Given matrix $A = (a_1, a_2, \cdots, a_n)$, its corresponding Q matrix for QR decomposition satisfies that*

$$
Q = \left(\frac{q_1^{'}}{|q_1^{'}|}, \frac{q_2^{'}}{|q_2^{'}|}, \cdots, \frac{q_n^{'}}{|q_n^{'}|} \right) \tag{5.8}
$$

where

$$q_i^{'} = \overrightarrow{a_1} \otimes \overrightarrow{a_2} \cdots \otimes \overrightarrow{a_i} = \begin{vmatrix} \overrightarrow{a_1} & \overrightarrow{a_2} & \cdots & \overrightarrow{a_i} \\ \overrightarrow{a_1} \cdot \overrightarrow{a_1} & \overrightarrow{a_2} \cdot \overrightarrow{a_1} & \cdots & \overrightarrow{a_i} \cdot \overrightarrow{a_1} \\ \overrightarrow{a_1} \cdot \overrightarrow{a_2} & \overrightarrow{a_2} \cdot \overrightarrow{a_2} & \cdots & \overrightarrow{a_i} \cdot \overrightarrow{a_2} \\ \vdots & & & \\ \overrightarrow{a_1} \cdot \overrightarrow{a_{i-1}} & \overrightarrow{a_2} \cdot \overrightarrow{a_{i-1}} & \cdots & \overrightarrow{a_i} \cdot \overrightarrow{a_{i-1}} \end{vmatrix}$$

$$\begin{aligned} &=(-1)^{1+1}\overrightarrow{a_1} \cdot A_1^{(i)} + (-1)^{1+2}\overrightarrow{a_2} \cdot A_2^{(i)} \\ &+ \cdots + (-1)^{1+i}\overrightarrow{a_i} \cdot A_i^{(i)} \\ &=(-1)^{1+1}\overrightarrow{a_1} \cdot \frac{\cdot A_1^{(i)}}{A_i^{(i)}} + (-1)^{1+2}\overrightarrow{a_2} \cdot \frac{A_2^{(i)}}{A_i^{(i)}} \\ &+ \cdots + (-1)^{1+i}\overrightarrow{a_i} \\ &=(-1)^{1+1}\overrightarrow{a_1} \cdot d_1^{(i)} + (-1)^{1+2}\overrightarrow{a_2} \cdot d_2^{(i)} \\ &+ \cdots + (-1)^{1+i}\overrightarrow{a_i}, \end{aligned} \tag{5.9}$$

$d_j^{(i)} = \frac{|A_j^{(i)}|}{|A_i^{(i)}|}(1 \leq j < i)$, $A_1^{(i)} = A_i^{iT} \cdot A_1^i$, $\cdots$, $A_j^{(i)} = A_i^{iT} \cdot A_j^i$, $\cdots$, $A_i^{(i)} = A_i^{iT} \cdot A_i^i$. $A_j^i = (a_1, \cdots, a_{j-1}, a_{j+1}, \cdots, a_i)$, $A_i^i = (a_1, a_2, \cdots, a_{i-1})$ and $|A_j^{(i)}|$ is the determinant of matrix $A_j^{(i)}$.

Proof for (5.8) is straightforward, and is thus skipped for brevity.

Theorem 5.2.4. *For the sequence of* $D^{(i)} = (d_1^{(i)}, \cdots, d_j^{(i)}, \cdots, d_{i-1}^{(i)})$ *where* $1 \leq j < i$, $d_j^{(i)} = \frac{|A_j^{(i)}|}{|A_i^{(i)}|}$, $A_j^{(i)} = A_i^{iT} \cdot A_j^i, A_j^i = (a_1, \cdots, a_{j-1}, a_{j+1}, \cdots, a_i)$, $A_i^i = (a_1, \cdots, a_{i-1})$, *there exists the following equation*

$$A_i^{(i)} \cdot D^{(i)} = A_i^{iT} \cdot a_i. \tag{5.10}$$

Proof of such theorem is constructed by simply applying Cramer's rule to (5.10), and is thus skipped for brevity.

Comparing to determining each vector in the Q matrix individually, we prefer determining all vectors in $D^{(i)}$ systematically. Actually, given $D^{(i)}$ and

$$A_i^{(i)} \cdot X = A_i^{iT} \cdot a_i, \tag{5.11}$$

there exists that the solution X to (5.11) is the expected sequence to $D^{(i)}$. We thus propose to determine $D^{(i)}$ by solving an equation set in (5.11). For convenience of reference in the rest of the dissertation, we denote the equation set in (5.11) as $D^{(i)}$'s companion equation set, i.e., $CEQ(D^{(i)})$.

5.2.1 Implementation and Experimental Results

To evaluate the performance of the proposed ParaQR method, we implement and compare it together with other five algorithms including CGS, MGS, HouseHolder, Givens, and Cholesky on a high-performance computing node. The computing node is equipped with one GTX580 GPU, dual 6-core processors at the speed of 2.3 GHz for each core and 64GB RAM. We select the GPU to run the parallelized algorithms. We test executable of each algorithm on multiple benchmark dataset of different sizes, collect experimental performance metrics of each algorithm and show them in Figure 5.2. It is valid to infer the following conclusions from Figure 5.2:

- 1: Proposed algorithm reveals an unanimous performance advantages over all existing methods on all considered data sizes. As the data size increases, the advantage extent enlarges further.
- 2: Given the same-sized dataset, performances of existing algorithms follow the order that HouseHolder<Cholesky<MGS<CGS<Givens<ParaQR methods,which complies with the established theoretical analysis and justifies performance superiority of the proposed algorithm.

Function 33 ParaQR($A_{m\times n}$)
Input: a non-singular matrix A of $m \times n$ and rank n
Output: two decomposed matrixes Q,R such that $A = QR$, where Q is $m \times n$ orthogonal, i.e., $Q^T \cdot Q = I_n$ and R is $n \times n$ upper triangular

```
 1: $M[1:n,1:n] = \begin{pmatrix} \overrightarrow{a_1}^T \\ \overrightarrow{a_2}^T \\ \cdots \\ \overrightarrow{a_n}^T \end{pmatrix} (\overrightarrow{a_1}, \overrightarrow{a_2}, \cdots, \overrightarrow{a_n});$
 2: $InversR[1:n,1:n] = \begin{pmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & & & \\ 0 & 0 & \cdots & 1 \end{pmatrix}_{n\times n};$
 3: $R[1:n,1:n] = \begin{pmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & & & \\ 0 & 0 & \cdots & 1 \end{pmatrix}_{n\times n};$
 4: $Q[1:n,1:n] = \begin{pmatrix} 0 & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 \\ \vdots & & & \\ 0 & 0 & \cdots & 0 \end{pmatrix}_{n\times n};$
 5: $Static\_M[1:n,1:n] = M[1:n,1:n];$
 6: for i=1:n do
 7:     Simultaneously do lines 8 to 9
 8:     $M[i+1:n,i:n] = M[i+1:n,i:n] - \frac{1}{M[i,i]} \cdot M[i+1:n,i:n] \cdot M[i,i:n];$
 9:     $M[1:i-1,i:n] = M[1:i-1,i:n] - \frac{1}{M[i,i]} \cdot M[1:i-1,i:n] \cdot M[i,i:n];$
10:     $InversR[1:i-1,i] = M[1:i-1,1:i-1]^{-1} \times Static\_M[1:i-1,i];$
11:     $q'_{i-1} = Static\_M[:,1:i-1] \cdot InversR[1:i-1,i];$
12:     $q_{i-1} = \frac{q'_{i-1}}{|q'_{i-1}|};$
13:     $R[1:i-1,i] = \left(q'_1, q'_2, \cdots, q'_{i-1}\right)^T \cdot \overrightarrow{a_1};$
14: $R = (InversR)^{-1} \times \begin{pmatrix} |q'_1| & 0 & \cdots & 0 \\ 0 & |q'_2| & \cdots & 0 \\ \vdots & & & \\ 0 & 0 & \cdots & |q'_n| \end{pmatrix};$
15: $Q = \left(\frac{q'_1}{|q'_1|}, \frac{q'_2}{|q'_2|}, \cdots, \frac{q'_n}{|q'_n|}\right);$
16: return $Q, R$
```

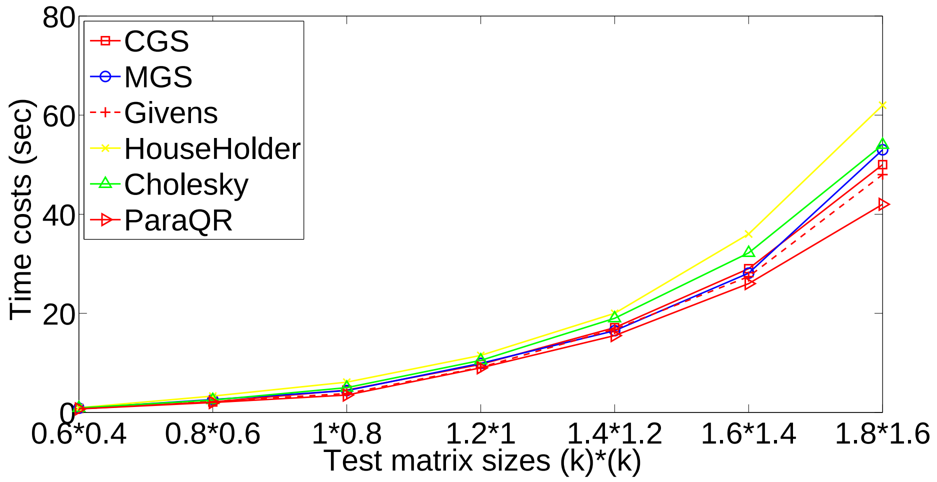


Figure 5.2 Performances of different algorithms on different sizes of data.

CHAPTER 6

CONCLUSION

We consider a few practical problems in the domain of scientific visualization and computing, briefly summarize each of them as follows, respectively.

6.1 Image Composition in Scientific Visualization

We proposed a Grouping More and Pairing Less (GMPL) method for sort-last parallel rendering. To reduce the image composition time, GMPL employs two approaches: prime factorization and Improved Direct Send. Prime factorization is considered based on a common pattern that underlies the three most widely used methods: DS, PP and BS, i.e., the more thoroughly the number of processors is factorized, the more latency performance gain may be achieved. ImprovedDS is considered in the case where the number of processors is already prime. In such a case, we designed a new sending/receiving scheme, which, ideally, is able to reduce the connection establishing time almost by half compared with the original DS method.

We parallelized the *over* operator for performance improvement by exploiting the parallelism in blending different channels and summing up partial blending results. For performance evaluation, we constructed cost models for both of the operators, and conducted rigorous theoretical analyses to justify the advantages of the generalized operator in terms of efficiency, which was confirmed by extensive experimental results. We also proposed an improved *over* operator that was capable of performing blending on any number of operands arriving in any arbitrary order. Such order independency overcomed the performance limitation of the traditional *over* operator. We implemented the proposed operator in both a sequential and a parallel mode, and applied it to the image composition problem in parallel volume visualization. Both theoretical analyses and extensive experimental results showed its performance superiority over the traditional one.

In addition, we proposed a volume visualization architecture that integrated three components, i.e., a parallelized per-ray integration method, a real-time *over* operator, and a pipelining scheme that overlapped rendering and composition. The proposed techniques overcame the performance limitations of traditional methods. We implemented the proposed architecture and evaluated the performances using real-life scientific datasets on a high-performance visualization cluster. Both theoretical analyses and extensive experimental results demonstrated their performance superiorities over existing ones.

6.2 QR Decomposition in Scientific Computing

In the QR decomposition procedure, we derived two theorems that illustrated the prototype of the proposed procedure and confirmed its stability, respectively. We further developed the prototype into a fully-fledged method, parallelized the corresponding procedure, justified its performance superiority in comparison with other existing methods.

BIBLIOGRAPHY

[1] K. Akeley and T. Jermoluk. High-performance polygon rendering. *SIGGRAPH Comput. Graph.*, 22(4):239–246, Jun. 1988.

[2] Alfonse. Blending. http://www.opengl.org/wiki/Blending, accessed on July 4th 2013.

[3] A. Appel. Some techniques for shading machine renderings of solids. In *Proc. of the April 30–May 2, 1968, Spring Joint Computer Conference*, pages 37–45, 1968.

[4] L. Bavoil and K. Myers. Order independent transparency with dual depth peeling. Technical report, NVIDIA OpenGL SDK, 2008.

[5] F. F. Bernardon, C. A. Pagot, J. L. Comba, and C. T. Silva. GPU-based tiled ray casting using depth peeling. *J. Graph. GPU Game Tools*, 11(4):1–16, 2006.

[6] A. Björck. Solving linear least squares problems by gram-schmidt orthogonalization. *BIT Num. Math.*, 7(1):1–21, 1967.

[7] A. Björck. Least squares methods. *Handbook of Numerical Analysis*, I:Solution of Equations in R (Part 2), 1991.

[8] A. Björck. Numerics of gram-schmidt orthogonalization. *Linear Algebra Appl.*, 197:297–316, 1994.

[9] P. Bunyk, A. Kaufman, and C. Silva. Simple, fast, and robust ray casting of irregular grids. In *Proc. of Scientific Visualization Conference*, pages 30–36, 1997.

[10] E. Catmull. *A subdivision algorithm for computer display of curved surfaces.* PhD thesis, Univ. of Utah, Salt Lake, Dec. 1974.

[11] X. Cavin, C. Mion, and A. Filbois. Cots cluster-based sort-last rendering: performance evaluation and pipelined implementation. In *Proc. of the 16th IEEE Visualization 2005*, pages 111–118, Oct 2005.

[12] M. Cosnard, J. M. Muller, and Y. Robert. Parallel QR decomposition of a rectangular matrix. *Numer. Math.*, 48(2):239–249, 1986.

[13] R. Crandall and C. Pomerance. *Prime Numbers: A Computational Perspective.* Springer, 2001.

[14] D. E. Culler, R. M. Karp, D. Patterson, A. Sahay, E. E. Santos, K. E. Schauser, R. Subramonian, and T. Von Eicken. Logp: A practical model of parallel computation. *Commun. ACM*, 39(11):78–85, 1996.

[15] J. Demmel, L. Grigori, M. Hoemmen, and J. Langou. Communication-avoiding parallel and sequential QR factorizations. *SIAM J. Sci. Comput.*, 34(1):A206–A239, 2012.

[16] J. D. Dixon. Asymptotically fast factorization of integers. *Math. Comp.*, 36(153):255–260, 1981.

[17] O. Egecioglu. Givens and householder reductions for linear least squares on a cluster of workstations. Technical report, Santa Barbara, CA, USA, 1995.

[18] S. Eilemann and R. Pajarola. Direct send compositing for parallel sort-last rendering. In *Proc. of the 7th Eurographics Conf. on Parallel Graphics and Visualization*, pages 29–36, 2007.

[19] L. Eldin. A parallel QR decomposition algorithm. *Report LiTh Mat-R-1988-02 Dept. of Math., Linkoping University.*

[20] W. Fang, G. Sun, P. Zheng, T. He, and G. Chen. Efficient pipelining parallel methods for image compositing in sort-last rendering. In *Network and Parallel Computing*, pages 289–298. 2010.

[21] G. Fasshauer. QR Factorization. *Numerical Linear Algebra/Computational Mathematics*, I.

[22] L. Fernandez and J. M. Garcia. The performance of fast givens rotations problem implemented with mpi extensions in multicomputer. *WIT Trans. on Info. Com. Tech.*, 18, 1997.

[23] S. Flannery and D. Flannery. *In Code: A Mathematical Journey.* Workman Publishing Company, 1st edition, 2000.

[24] G. A. Geist and M. T. Heath. Parallel cholesky factorization on a hypercube multiprocessor. *Report ORNL-6190 Oak Ridge Nat. Lab.*, page 51, 1985.

[25] G. H. Golub and C. F. Loan. *Matrix computations*, volume 3. 2012.

[26] J. P. Gram. Ueber die entwickelung reeler funtionen in reihen mittelst der methode der kleinsten quadrate. *J. Reine. Angew. Math.*, 94:71–73, 1908.

[27] S. Grimm, S. Bruckner, A. Kanitsar, and E. Groller. Memory efficient acceleration structures and techniques for cpu-based volume raycasting of large data. In *2004 IEEE Symposium on Volume Visualization and Graphics*, pages 1–8, 2004.

[28] H. Gruen and N. Thibieroz. OIT and indirect illumination using DX11 linked lists. In *Proc. of Game Developers Conference*, 2010.

[29] N. J. Higham. *Perturbation Theory for Linear Systems*, chapter 7, pages 119–137.

[30] M. Howison, E. W. Bethel, and H. Childs. Hybrid parallelism for volume rendering on large-, multi-, and many-core systems. *IEEE Trans. Vis. Comput. Graphics*, 18(1):17–29, Jan. 2012.

[31] H. Huang and T. Y. Tam. On the QR iterations of real matrices. *Linear Algebra Appl.*, 408:161–176, 2005.

[32] H. W. Lenstra Jr. Factoring integers with elliptic curves. *Annals of Mathematics*, 126(3):649–673, 1987.

[33] A. Kiełbasinski. Numerical analysis of the gram-schmidt orthogonalization algorithm. *Mat. Stos.*, pages 15–35, 1974.

[34] J. Kruger and R. Westermann. Acceleration techniques for GPU-based volume rendering. In *Proc. of the 14th IEEE Visualization 2003*, page 38, 2003.

[35] C. L. Lawson and R. J. Hanson. *Solving Least Squares Problems.* Classics in Applied Mathematics. 1995.

[36] T. Lee, J. Lee, H. Lee, H. Kye, Y. G. Shin, and S. H. Kim. Fast perspective volume ray casting method using GPU-based acceleration techniques for translucency rendering in 3D endoluminal CT colonography. *Comput. Biol. Med.*, 39(8):657–666, 2009.

[37] T. Y. Lee, C. S. Raghavendra, and J. B. Nicholas. Image composition schemes for sort-last polygon rendering on 2D mesh multicomputers. *IEEE Trans. Vis. Comput. Graphics*, 2(3):202–217, 1996.

[38] D. H. Lehmer and R. E. Powers. On factoring large numbers. *Bulletin of the American Mathematical Society*, 37(10):770–776, 1931.

[39] M. Levoy. Efficient ray tracing of volume data. *ACM Trans. Graph.*, 9(3):245–261, 1990.

[40] M. Levoy. A hybrid ray tracer for rendering polygon and volume data. *IEEE Comput. Graph.*, 10(2):33–40, 1990.

[41] C. F. Lin, S. K. Liao, Y. C. Chung, and D. L. Yang. A rotate-tiling image compositing method for sort-last parallel volume rendering systems on distributed memory multicomputers. *J. Inf. Sci. Eng.*, 20(4):643–664, 2004.

[42] K. L. Ma, J. S. Painter, C. D. Hansen, and M. F. Krogh. Parallel volume rendering using binary-swap compositing. *IEEE Comput. Graph. Appl.*, 14(4):59–68, 1994.

[43] S. Marchesin, C. Mongenet, and J. M. Dischler. Multi-GPU sort-last volume visualization. In *Proc. of the 8th Eurographics Conf. on Parallel Graphics and Visualization*, pages 1–8, 2008.

[44] G. Marmitt and P. Slusallek. Fast ray traversal of tetrahedral and hexahedral meshes for direct volume rendering. In *Proc. of the Eighth Joint Eurographics/IEEE VGTC conference on Visualization*, pages 235–242, 2006.

[45] H. Meshkin. Sort-independent alpha blending. Technical report, Perpetual Entertainment, 2007.

[46] B. Mildeand and M. Schneider. Parallel implementation of classical Gram-Schmidt orthogonalization on cuda graphics cards.

[47] S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs. A sorting classification of parallel rendering. *IEEE Comput. Graph. Appl.*, 14(4):23–32, 1994.

[48] K. Moreland, B. Wylie, and C. Pavlakos. Sort-last parallel rendering for viewing extremely large data sets on tile displays. In *Proc. of the IEEE 2001 Symposium on Parallel and Large-data Visualization and Graphics*, pages 85–92, 2001.

[49] C. Müller, M. Strengert, and T. Ertl. Optimized volume raycasting for graphics-hardware-based cluster systems. In *Proc. of the 6th Eurographics Conf. on Parallel Graphics and Visualization*, pages 59–67, 2006.

[50] D. P. O'Leary, G. W. Stewart, and R. V. Geijn. *DOMINO: A message passing environment for parallel computation.* 1986.

[51] P. D. O'Leary and P. Whitman. Parallel QR factorization by Householder and modified Gram-Schmidt algorithms. *Parallel Comput.*, 16(1):99–112, 1990.

[52] S. Oliveira, L. Borges, M. Holzrichter, and T. Soma. Analysis of different partitioning schemes for parallel Gram-Schmidt algorithms. *Parallel Algorithms Appl.*, 14(4):293–320, 2000.

[53] B. Parhami. Periodically regular chordal ring networks for massively parallel architectures. In *Proc. of the Fifth Symposium on the Frontiers of Massively Parallel Computation*, FRONTIERS '95, pages 315–323, 1995.

[54] A. Patney, S. Tzeng, and J. Owens. Fragment-parallel composite and filter. In *Proc. of the 21st Eurographics Conf. on Rendering*, EGSR'10, pages 1251–1258, 2010.

[55] T. Peterka, D. Goodell, R. Ross, H. W. Shen, and R. Thakur. A configurable algorithm for parallel image-compositing applications. In *Proc. of the ACM/IEEE Int. Conf. on High Performance Computing Networking, Storage and Analysis*, pages 4:1–4:10, 2009.

[56] T. Porter and T. Duff. Compositing digital images. *SIGGRAPH Comput. Graph.*, 18(3):253–259, 1984.

[57] A. Pothen, S. Jha, and U. Vemulapati. Orthogonal factorization on a distributed memory multiprocessor. 1986.

[58] A. Pothen and P. Raghavan. Distributed orthogonal factorization: Givens and Householder algorithms. *SIAM J. Sci. Stat. Comp.*, 10(6):1113–1134, 1989.

[59] S. D. Roth. Ray casting for modeling solids. *Comput. Vision Graph.*, 18(2):109–144, 1982.

[60] R. Samanta, T. Funkhouser, K. Li, and J. P. Singh. Hybrid sort-first and sort-last parallel rendering with a cluster of PCs. In *Proc. of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 97–108, 2000.

[61] A. H. Sameh and D. J. Kuck. On stable parallel linear system solvers. *J. ACM.*, 25(1):81–91, 1978.

[62] E. E. Santos and P. Y. Chu. Efficient and optimal parallel algorithms for Cholesky decomposition. *J.M.M.A.*, 2(3):217–234, 2003.

[63] E. Schmidt. Zur theorie der linearen und nichtlinearen integralgleichungen. iii. teil. *Math. Ann.*, 65(3):370–399, 1908.

[64] A. Smoktunowicz, J. L. Barlow, and J. Langou. A note on the error analysis of classical Gram–Schmidt. *Numer. Math.*, 105(2):299–313, 2006.

[65] A. Stathopoulos and K. Wu. A block orthogonalization procedure with constant synchronization requirements. *SIAM J. Sci. Comput.*, 23(6):2165–2182, 2002.

[66] G. W. Stewart. Block Gram-Schmidt orthogonalization. *SIAM J. Sci. Comput.*, 31(1):761–775, 2008.

[67] A. Takeuchi, F. Ino, and K. Hagihara. An improved binary-swap compositing for sort-last parallel rendering on distributed memory multiprocessors. *Parallel Comput.*, 29(11-12):1745–1762, Nov. 2003.

[68] P. Thomas and D. Tom. Compositing digital images. *SIGGRAPH Comput. Graph.*, 18(3):253–259, Jan. 1984.

[69] J. P. Tignol. *Galois' Theory of Algebraic Equations.* World Scientific, Singapore, 2001.

[70] T. Yokozawa, D. Takahashi, T. Boku, and M. Sato. Efficient parallel implementation of classical Gram-Schmidt orthogonalization using matrix multiplication.

[71] H. Yu, C. Wang, and K. L. Ma. Massively parallel volume rendering using 2-3 swap image compositing. In *Proc. of the ACM/IEEE Int. Conf. on High Performance Computing Networking, Storage and Analysis*, pages 1–11, 2008.

[72] J. Zhang and K. Wong. Fast QR decomposition of matrix and its applications to numerical optimization.

[73] M. Zhu, M. Guo, L. Wang, and Y. Dai. A ray casting accelerated method of segmented regular volume data. In *Advanced Research on Electronic Commerce Web Application and Communication in Computer and Information Science*, volume 144, pages 7–12, 2011.