An Efficient PIM Architecture for Computer Graphics

Jae Chul Cha and Sandeep K. Gupta

{jaecha, sandeep}@poisson.usc.edu

Abstract

Rapid advance of manufacturing technology has enabled us to build higher performance graphic processors with much smaller area. And, based on the current trends, we can predict that System-on-Chip (SOC) with the substantial number of graphic processors will emerge in the near future. The demands for higher performance graphic system have been deepened because of the incessant desire toward realism in graphics. In those regards, it will be worth while to explore single chip parallel architecture for such data-intensive application with the large number of processors. In this paper, we propose Processor-in-Memory (PIM) architecture tailored for computer graphics. And, we present efficient algorithms on partitioning and placement for given scene in the proposed PIM architecture and show that our approach can reduce searching space significantly, achieving high efficiency.

1. Introduction

Even though the advance of VLSI technologies has provided more capacities for computations and memories [1], most architectures fail to sufficiently exploit these advancements for the efficient parallel computations due to processor-memory bottleneck. Hence, PIM architectures that integrate processors and memory on the same chip have been proposed for many applications. Such integration enables efficient parallel computation and communication along with low power consumption. Other types of architectures have been also proposed to exploit VLSI scaling. [4][5][6][7][8]

Today's sophisticated 3D animations have workloads that are too large to process in real time. Therefore, most 3D animations are produced in a non-real manner – each scene is created separately instead of continuously and such digital scenes are converted to analog films, which are played continuously in theatres. This type of off-line approach is inevitable, since the computing power is too limited for handling substantially large size graphics in real-time. However, with advances in VLSI technologies, processors will be able to support real-time animations in the future if the memory bottleneck can be eliminated. Based on this perspective, we explore a PIM architecture tailored for massive parallel processing for graphics applications.

To achieve maximum performance, the workload must be evenly divided among memories, and the each placement must be determined to minimize the communication overhead between processors. Such two aspects are studied and discussed in details in this paper. And, we will show that our partitioning and placement methodologies can achieve significant performance improvement.

2. Proposed Architecture

We present a new PIM architecture to provide high performance for graphics applications. Figure 1 is an overall architecture of a typical main memory connected via a decoder tree. We modify (a) to obtain the new PIM architecture, namely, CIMM – computation-in-memory-modules. The main components for this architecture are memory blocks, embedded processors called embedded computing elements (ECE's), special decoders and controllers. Note that, to the each memory block, we added two processors, namely, a geometry processor (GP) and a rasterization processor (RAS), as also shown in the Figure 2.

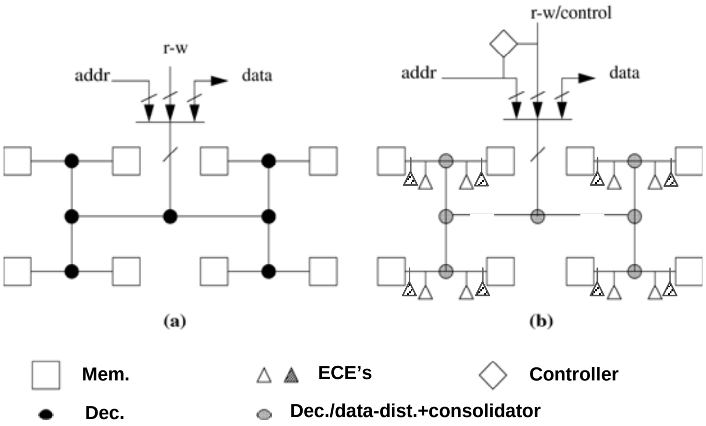


Figure 1 Organization of CIMM. (a) Organization of a classical main memory. (b) Organization of a CIMM, obtained by adding embedded computing elements (ECE's) within the classical main memory.

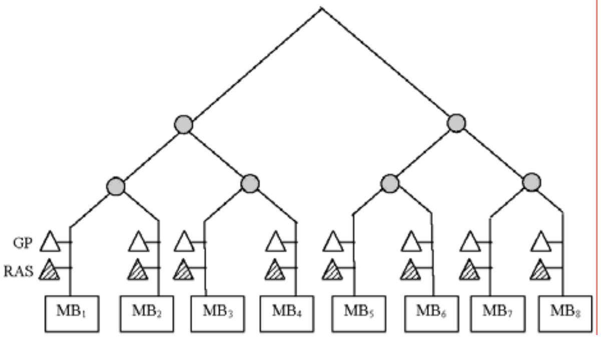


Figure 2 Simplified PIM architecture for graphics

The modified PIM architecture supports efficient communications to/from the external port, between memory blocks and between GP_i and RAS_j. We call this special decoder a DDDC – decoder, data-distributor, and consolidator. DDDC has many new data-distribution modes. For example, during a write operation, a node in DDDC can be controlled to activate both its children (two-way broadcast). DDDC can also activate its left child as the read mode and its right child as the write mode. In this case, the data forwarded to the DDDC by its left child is forwarded to its right child for writing.

3. Background on Computer Graphics

We introduce some basic concepts of computer graphics and discuss how to partition given scenes and assign the partitioned scenes to processors for efficient computation.

Most objects are represented using polygons, commonly triangles in most graphic systems. In order to add realism to the objects, each polygon is filled up with colors or textures, and effects such as shadows and lights are included. From a mathematical point of view, processing involves floating point and integer arithmetic operations as well as matrix operations, such as translation, rotation, scaling, shearing, tapering, twisting, and bending. When scenes have extremely high complexity, a single processor cannot meet the computational needs, especially in the pressure of real-time deadlines. In such parallel architectures, the performance is influenced significantly by how scenes are partitioned and assigned to processors.

Originally, an object is defined with respect to object-coordinate, which is attached at each body of object, then it is transformed into world-coordinate, and finally to the viewer-coordinate (Figure 3). After all other processing is completed, this object is mapped from 3D to 2D, since viewers see objects on a 2D screen.

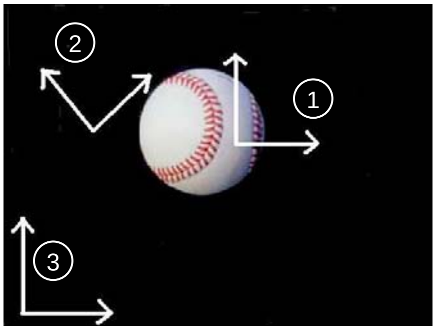


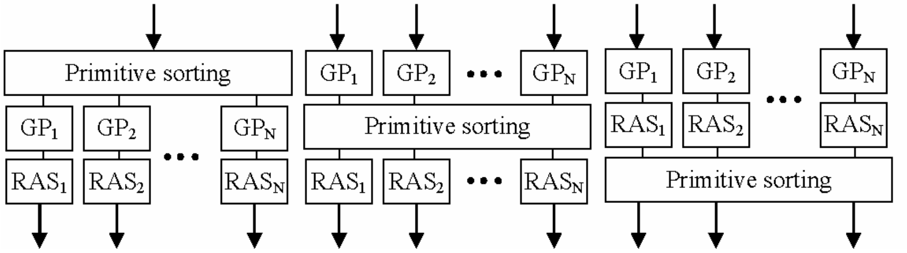
Figure 3. Space coordinates – 1.Object-coordinate 2.World-coordinate 3.Screen-coordinate

Graphic processors are composed of geometry processor and rasterization processor. Geometry processing transforms the primitives from one coordinates system to another. This is carried out by applying various matrix operations to each polygon. The main responsibility of the rasterization processor is to shade objects by filling up each polygon with the colors and textures and to map the 3D objects onto a 2D screen. The coordinates for geometry processing are called world-space (or object space), whereas those for rasterization processing are called image-space (or screen-space). Vertices of polygons are transformed by geometry processor and then undergo the rasterization process.

Initial input files to geometry processors contain vertex information for each polygon, such as the position of the vertex, the color of the vertex, the normal vector of each polygon, and the texture mapping information. The normal vector is used for lighting calculation by considering two vectors, namely, light direction and normal vector of surface. Vertices from the same polygon are assigned the same ID to keep track of the vertices that belong to a polygon.

Size of polygon doesn't affect the performance of geometry processing, since the performance of geometry processing is only related to the number of vertexes. But, the performance of rasterization processor depends on the size of polygons, since the complexity of shadings is a function of object size.

The current trend is that even small polygons are being replaced by tiny dots for much deeper details. Therefore, the demands for high performance computing systems will be higher as time passes by. And, thanks to the advance of the manufacturing technology, Graphical Processing Units (GPU) will be able to handle substantially large number of small polygons or dots on-line in the future.



(a) Sort-First (b) Sort-Middle (c) Sort-Last

Figure 4 Classification of communication types

Since polygons undergo spatial transformations, they need to be rearranged somewhere in the pipeline. According to the location of data rearrangement, we classify them into three types – sort first, sort middle, sort last [11], as shown in the Figure 4. Among three sorting strategies, our focus will be on sort middle.

We confine our discussion to the parallelizable portions in graphic procedures. Therefore, non-parallelizable interaction parts in a scene will be excluded from our discussion.

4. Pre-partition/placement phase

In our parallel processing architecture, the scenes in the world space are partitioned into sub-scenes and distributed to geometry processors at each node. Subsequently, the scenes in the image space are split into sub-scenes and allocated to rasterization processors (Figure 2). Overall, we decide the partitioning of the object-space and image-space first, then decide the placement of such partitioned jobs. The main objective of data partitioning is to balance the workload in parallel processors and the main objective of placement is to minimize the communication complexity between GP and RAS.

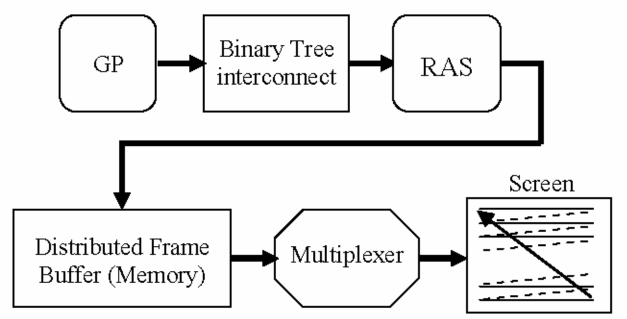


Figure 5. Overall procedures for producing final scenes

In this study, we assume to have distributed frame buffers at the end of rendering pipeline. The multiplexer between frame buffers and physical screen makes the ordered scans of picture lines to be possible on the physical screen regardless of the order of the placement of sub-frames in memory blocks (Figure 5). Therefore, the placement of rasterization processors can be chosen arbitrarily while we maintain the placement of distributed frame buffers to be same as that of rasterization processors. By doing so, no communication occurs between rasterization processor and frame buffers.

As temporally adjacent frames tend to be similar to each other, we use the information from the previous frame as a reference for partitioning and placement of the current frame. Of course, when abrupt change of scene occurs, our reference will be inevitably off the mark.

We construct 3D polygon distribution map that contains the information on the statistics of polygon distribution for both geometry processor and rasterization processor along with their mapping information between them. Processes of constructing map are: 3D scenes from both world space and image space are sliced into small cubes according to the given resolution and hardware specification. And, mapping information between cubes from the world space and those from the image space are stored in Look-Up Table (Figure 6). In the table, it contains the information on the number of polygons and the area sum of polygons. Distribution statistics on the number of polygons is used for determining the partitioning of geometry processor whereas distribution statistics on the area sum of polygons is used for the determination of partitioning method for rasterization processor. After partitioning is finished, we decide optimal placement by finding the placement with the minimal communication cost. Referring to the LUT in the Figure 5, the number of vertexes in a particular cube in a world space can be computed by summing up all the row elements in a corresponding row. Likewise, the area sum of polygons in a single cube in the screen space can be calculated by adding up all the terms in a corresponding column.

RAS GP	Cube 1	Cube 2	Cube 3	…
Cube 1	C_{11}, A_{11}	C_{12}, A_{12}	…	
Cube 2	C_{21}, A_{21}	C_{22}, A_{22}	…	
Cube 3	C_{31}, A_{31}			
…				

※ Where C is the number of vertex count, A is the area sum of polygons.

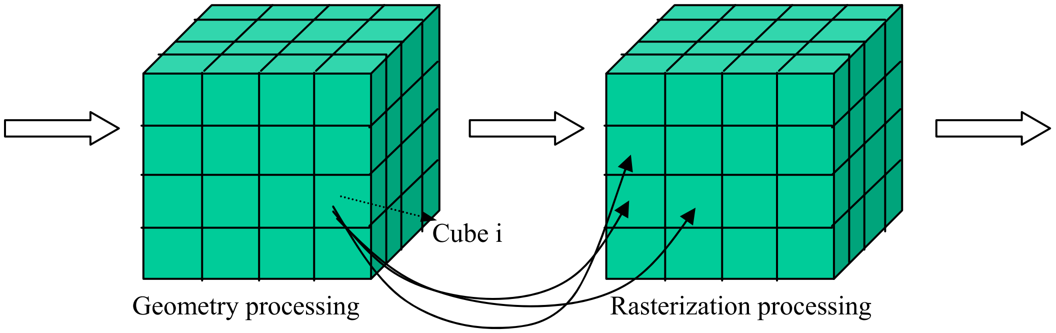


Figure 6. Look-Up Table for 3D mapping information between object space and image space

In three dimensional spaces, the area of triangle can be obtained by using pythagorean sum of the respective projection on the three principal planes. The computation takes 17 addition/subtractions, 10 multiplications, 1 square root. Assuming that the total number of polygons in a scene is n_p, the time complexity for computing the area sum in the cubes is simply $O(n_p)$. This amount of workload is insignificant compared to the complexities of GP and RAS procedures. Memory complexity for LUT is $O((n_c)^2)$ assuming that the number of cubes is n_c. Hardware complexity is – $(n_c)^2$ counters for vertex counts, $(n_c)^2$ adders for area sum, $6 \cdot n_c$ comparators for deciding the location of polygons in cubes, and one pipelined computational units used for the area calculation.

5. Optimal partitioning methodology

Basically, some of the known methodologies for creating partitions for 3D scenes are:

1. Fixed partitioning with respect to object space & screen space.
2. Adaptive partitioning with respect to object space & screen space.
3. Time slot/slice based distribution
4. Completely scattered distribution

Fixed partitioning methodology conducts recursive bi-partitioning cuts either in x or y or z direction. Adaptive partitioning can cut the whole 3D scene in any directions in any position. Time slot/slice based partitioning method is for one node to receive all the input data during a particular time interval. After the interval, another processor gets data as a next turn, and so forth. For the completely scattered distribution, every inputted polygon is allocated to each node one by one.

In this paper, our focus will be on the fixed partitioning methodology. The details of the method are:

1. Each cut produces equal sized bi-partitions (equi-partitioning)
2. Bi-partitioning cut is applied to all the existing partitions.

The partitioning objective is to find the most well-balanced partitioning cuts in terms of workload among all the possible fixed partitioning methods.

Generation of non-equivalent partitioning methods – Two step reduction

Searching space for finding desirable partitioning method becomes exponentially large as the number of processing elements is increased (more ahead). Therefore, we developed search space reduction algorithm, which consists of two parts, namely, scene independent part and scene dependent part.

<u>First reduction</u>: Scene independent

In case of fixed partitioning, bi-partitioning tree can be drawn for producing possible recursive cuts (Figure 7). Assuming that n is the number of processing units, the depth of tree is $\log_2{}^n$, since each recursive bi-partitioning is carried out to all the equi-partitions generated so far. In the Figure 7, the number of possible cuts in a complete partitioning tree including both dash line and solid line is $3^{\log_2^n}$, which is exponential. This makes the computational cost to be extremely high for system. However, we observe that, when a scene is divided by fixed partitioning methodology, the sequence of slicing doesn't matter, since each dimensional bi-partitioning is independent and the bi-partitioning is applied to all the existing partitions. For example, x-x-y cuts produce exactly the same result as x-y-x or y-x-x cuts. The systematic way to produce such distinct bi-partitioning cuts is: Branch x makes 3 children branches, namely x, y, z. Branch y is allowed to produce two branches, y, z. Lastly, branch z can produce only one child branch, z. Following the above instructions, we can build up the entire non-equivalent partition tree as shown in Figure 7. In this figure, solid lines are distinct paths that we need to explore, whereas dash lines are duplicative paths that we don't need to explore.

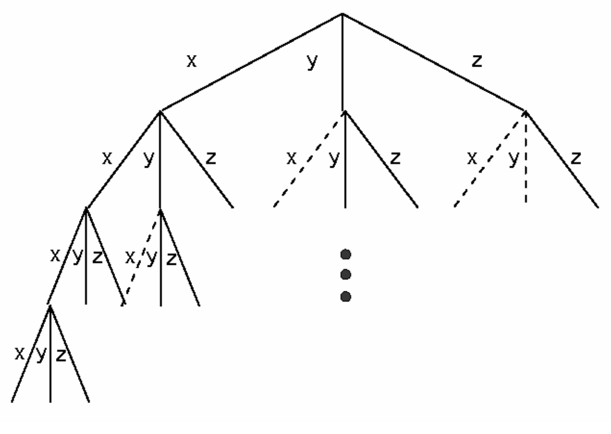


Figure 7 Fixed-Partitioning generation tree – without replication

Now, the total number of distinct leaves at the bottom of the non-replicated tree can be computed as $\frac{1}{2}(log_2^n + 1)\cdot(log_2^n + 2)$. Its derivation can be done by using the following formula.

$$n_x + n_y + n_z = \log_2^n$$

where, n is the total number of processing units, n_x is the number of x-directional cut, n_y, y-directional cut, n_z, z-directional cut. Assuming that we conduct k bi-partitionings, the recursive cuts produce 2^k partitions. By equating 2^k and n, we obtain k=$\log_2^n$ that corresponds to the term in the right-hand side of the above equation. Now, we need to count the number of distinct sets, namely, {n_x, n_y, n_z}. We can solve the above equation like:

When $n_x = 0, \quad n_y + n_z = \log_2^n$

When $n_x = 1, \quad n_y + n_z = \log_2^n - 1$

When $n_x = 2, \quad n_y + n_z = \log_2^n - 2$

…

Therefore, the number of combinations of (n_x, n_y, n_z) will be:

$$\begin{aligned} T &= (\log_2^n + 1) + (\log_2^n) + (\log_2^n - 2) + \bullet\bullet\bullet \\ &= \frac{1}{2}(\log_2^n)^2 + \frac{3}{2}\log_2^n + 1 \\ &= \frac{1}{2}(\log_2^n + 1)\cdot(\log_2^n + 2) \end{aligned}$$

Compared with the complete tree with replicate leaves, the number of bottom leaves from non-replicated tree is much smaller than that of the replicated tree. As an example, when n=1024, the total number of distinct bottom leaves are 66 whereas that of replicated tree is 58884. Therefore, we can achieve a significant improvement.

The fact that the ordering of slicing sequence doesn't make any difference is not affected by scene characteristics. Thus, we can say that the process of reduction of tree branches is ***scene-independent.***

<u>Second reduction</u>: Scene dependent

Furthermore, we can reduce the number of searching paths in a tree by considering the worst case and the best case scenario. We start by a simple example. Suppose that the scene has 1000 vertexes in total, and we have 8 processing units. Assume that x-directional cut at the top of tree split the polygons into (100, 900), y-directional cut (500,500), z-directional cut (150, 850). Since x and z cuts are heavily unbalanced, we explore y-directional cut first with a higher priority over other cuts. In the next step, suppose that z-directional cut after the first y-directional cut (i.e., y-z cut) produced (250, 250, 230, 270) while y-directional cut after the y-directional cut (i.e., y-y cut) resulted in (200, 300, 100, 400). Since the maximum value of (250, 250, 230, 270) is 270 whereas that of (200,300, 100, 400) is 400, we choose z-directional cut as a next exploration. Here, we used maximum value, not minimum value, since the computational time depends on the processor with the highest load, not the one with the lightest load. Then, as a last step, another z-directional cut is applied (i.e., y-z-z cuts), since z node can produce only z-directional branch as a child. Assume that (130, 120, 150, 100, 110,120, 130,140) was obtained finally. Now, we can realize that we don't need to traverse the unexplored x-directional cut and z-directional cut at the top of tree. The reason is as follows. The best scenario after the first x-directional cut at the top of the tree is that the rest of the subsequent slicing operations result in an equal load partitioning, i.e., (25, 25, 25, 25, 225, 225, 225, 255), producing a maximum value of 225, which is still larger than the maximum of (130, 120, 150, 100, 110,120, 130, 140) that was obtained by a y-z-z traverse. More formally,

$$\frac{1}{2^{0}} \bullet \mathit{The\ max\ load\ value\ in\ the\ currently\ exploring\ node}$$
$$\leq \frac{1}{2^{\mathit{remaining\ depth\ for\ the\ node}}} \bullet \mathit{the\ max\ load\ value\ at\ other\ unexplored\ node}$$

The term in the left side represents the worst case of the currently exploring node. The worst case scenario of the current node is that further partitionings after current partitioning do not partition the number of vertexes anymore. The term in the right side represents the best case of the other intermediate nodes to be explored. The best case scenario of the nodes is that further partitionings divide the number of vertexes equally. If the above inequality holds, then, we don't need to traverse the intermediate node, since the best case scenario of intermediate node to be explored will still give out worse result than the worst case scenario of the currently residing node. If that doesn't hold, we have to continue to search the path. We can explain it pictorially as shown in the Figure 8. Suppose that we choose x directional cut first among x, y, z cuts. Then, we select y-cut at the next level. After that, conduct y cut.

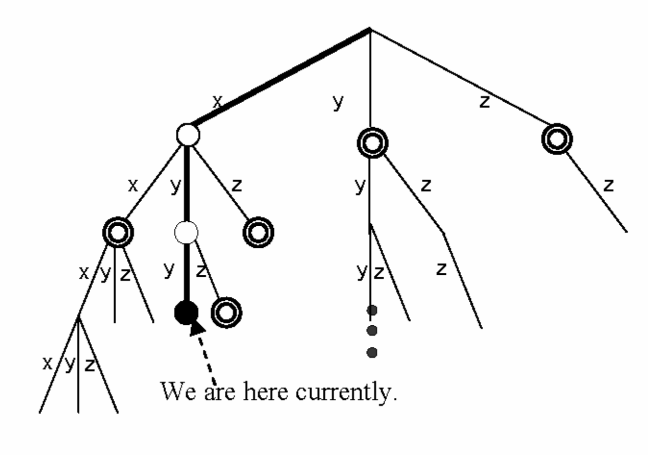


Figure 8 Intermediate states while creating a searching tee for fixed partitioning – ◎ : intermediate end-node, ○ : intermediate non-ending node, ● : currently exploring node

In the Figure 8, we compare the worst-case-scenario-values at the current node ● with the best-scenario-case values at nodes ◎. Here, the best case scenario is that the subsequent partitioning results in equal partitioning. For example, equal subsequent partitioning of (10, 40) is (5, 5, 20, 20). The worst case scenario is that the subsequent cutting results in no further workload division. For example, (10, 40) is partitioned into (10, 0, 40, 0). If the best scenario case value at the node ◎ still gives out worse results than the worst scenario case value at the node ●, then, we don't need to explore the node ◎ later.

This reduction technique is ***scene-dependent***, since this can or can not reduce the search space depending on the scene's characteristics.

In conclusion, by combining the scene-independent and the scene-dependent technique we developed, we can find the optimal partitioning methodology efficiently and quickly.

6. Optimal placement methodology

Suppose that we found the optimal partitioning method at both object space (GP) and image space (RAS) by using the fixed partitioning technique. Then, we decide the placement of the partitioned jobs in the GP and RAS that minimizes the communication cost between GP and RAS. Since PIM architecture has various communicational distances between nodes by its own nature, the degrees of the utilization of parallelism are different for each placement.

As mentioned earlier, sorting procedures for polygons are inevitable, since the object space for geometry processing and the images space for rasterization processing are different from each other. Sorting can occur before geometry processor (sort-first), or after rasterization processor, (sort-last) or in-between (sort-middle). In general, the most common sorting methodologies are sort-middle and sort-last. Here, we use sort-middle technique for our architecture.

Assume that there are n nodes in the PIM architecture. Then, the possible placements at the geometry processors are n factorials. Likewise, those at rasterization processor are n factorials. Therefore, in total, there are $(n!)^2$ placement pairs of geometry and rasterization processor. While this is a substantially large number, we can reduce the number of searching space by discarding the equivalent placement pairs in PIM architecture. We show some examples how to distinguish equivalent pairs and non-equivalent pairs.

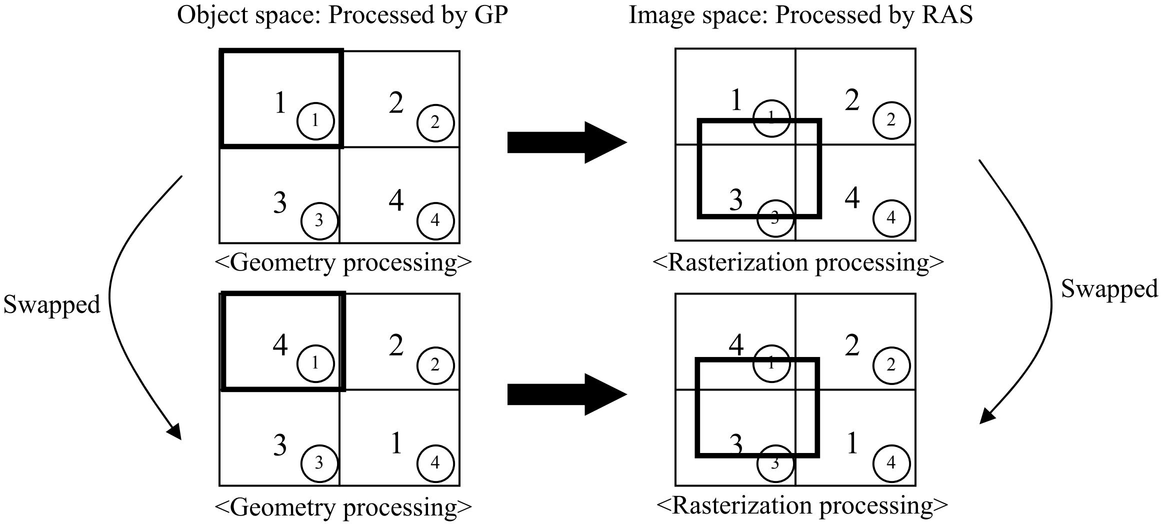

Object space: Processed by GP
Image space: Processed by RAS
1
2
3
4
<Geometry processing>
<Rasterization processing>
Swapped
Swapped
4
2
3
1
<Geometry processing>
<Rasterization processing>

Figure 9 Example of swapping process that produces non-equivalent placement in terms of communication cost

In the Figure 9, the number in a small circle shows an intrinsic sub-block id while the number in the center of sub-block indicates at which node the sub-block will be placed in. Partitioning methodologies for upper and low pair are exactly same, but, the placements are different. Bottom pair is a swapped version of upper pair by interchanging the data in the node 1 and 4 for both geometry and rasterization processor. Originally, sub-block 1 to be processed by GP was located at node 1, and sub-block 4 to be processed by GP was located at node 4. Likewise, sub-block 1 to be processed by RAS was located at node 1, and sub-block 4 to be processed by RAS was located at node 4. In swapped version, sub-block 1 to be processed by GP is located at node 4, and sub-block 4 to be processed by GP is located at node 1. Likewise, sub-block 1 to be processed by RAS is located at node 4, and sub-block 4 to be processed by RAS is located at node 1. Suppose that the rectangle with a thick line indicates the mapping relation between geometry processor and rasterization processor, i.e., the polygons in the rectangular box at the object space for geometry processing should be transferred to the rectangular box area at image space for rasterization processor. Referring to the swapped version pair, we know that most polygons that are processed by node 4 for geometry processing will be transferred to the neighboring place, namely, node 3 for the subsequent rasterization processing. And, some of the polygons need to be transferred to the node 1,2, and 4 for rasterization processing as seen above. Now, we see the original case in the Figure 9. Most polygons in node 1 for geometry processing will have to be transferred to the node 3 for rasterization processing, which results in higher communication cost than the swapped version of placements. Therefore, {1, 4} swapping did make difference with respect to communication cost. As another example, suppose we swap the data in the locations of node 1 and node 2 (Figure 10).

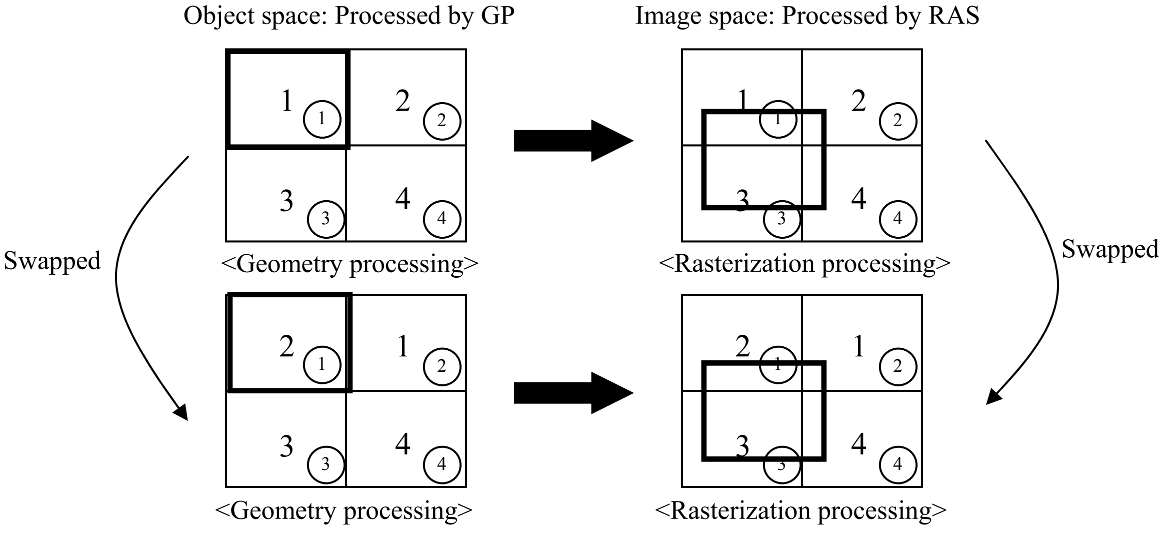

Object space: Processed by GP
Image space: Processed by RAS
1
2
3
4
<Geometry processing>
<Rasterization processing>
Swapped
Swapped
2
1
3
4
<Geometry processing>
<Rasterization processing>

Figure 10 Example of swapping process that produces equivalent placement in terms of communication cost

Then, in a swapped version, most polygons in node 2 that were processed by GP should be transferred to node 3 for rasterization procedure. This communication cost is same as that from the original version, where most polygons in node 1 are to be transferred to node 3. Now, we observe that certain swapping patterns create same results while others don't. In the above example, we can interpret the situation as neighboring and non-neighboring swaps. In the above examples, {1, 2} was a neighboring swap, whereas {1, 4} was a non-neighboring swap. But, the neighboring swap is not the only way to make the results exactly same. We will generalize the swaps that produce non-distinct pairs.

As a further example, suppose that the placement of the geometry processor is fixed, and the data in the node 1 and 2 at the rasterization processor is swapped. Then, even though {1, 2} are neighbors each other, the communication cost of the original placement pair is different from that of the swapped version of placement pair. In order to make same pairs, the data at both geometry and rasterization processor should be swapped.

Now, we present a concept called ***'Branch alternation'*** that is defined as swapping with respect to branching point. For example, referring to the Figure 11, data swapping between node 1 and node 2 doesn't affect the communication cost at all as long as both geometry and rasterization processor swap the two data altogether. The same phenomenon occurs for the swapping of data in node 3 and node 4. With the same reason, swapping between set {node 1, node 2} and set {node 3, node 4} doesn't make any difference in terms of communication cost when the data at both geometry and rasterization processor are swapped concurrently.

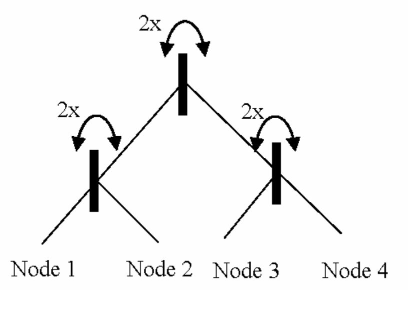


Figure 11 Illustration of types of branch alternation in case of 4 processing units

In the above example, 3 types of branch alternations are possible, which produce equivalent pairs. Therefore, 2×2×2=8 pairs will have exactly the same communication cost. In case of 4 processing units, branch alternation can reduce search space from $(4!)^2$ to $(4!)^2/8$. In general, since the number of splitting branches in the PIM architecture is 2^{n-1}, the total distinct placement pairs of geometry processor and rasterization processor will be $(n!)^2/2^{n-1}$.

Now, changing our view point, we can decompose $n!^2/2^{n-1}$ into two ways – $n! \times n!/2^{n-1}$ or $n!/2^{n-1} \times n!$. The former one means that the placements of geometry processor are just naïve permutations of n nodes whereas those of rasterization processor are distinct permutations with equivalent placements eliminated. The latter one means that the placements of rasterization processor are just naïve permutations of n nodes whereas those of geometry processor are distinct permutations with equivalent terms eliminated. As an example for the former case, the set {1,2,3,4,5,6,7,8} and the set {2,1,3,4,5,6,7,8} are different from each other in geometry processor (naïve permutation) whereas those two are considered to be same in rasterization processor, since we can match the two placements through a branch alternation.

We consider how to produce distinct permutations, any of which should not be matched through the branch alternations. Back to our original example of 8 processing element case, suppose that we divide the set {1,2,3,4,5,6,7,8} into two halves, namely, {1,2,3,4} and {5,6,7,8}. Then, find non-equivalent permutations from {1,2,3,4} and {5,6,7,8} respectively. After finished, do cross-product two findings, producing distinct permutations as a result (situation 1). Next, pick element 4 from {1,2,3,4} and pick 5 from {5,6,7,8}, and swap the two. Then, the updated sets will be {1,2,3,5} and {4,6,7,8}. And, in the same manner, find non-equivalent permutations for both of the sets, and cross-product them (situation 2). Note that the permutations produced from the situation 1 can not match with those from situation 2 even if we apply any sequences of branch alternations. Therefore, we can say that the former ones (distinct permutations of {1,2,3,4} × distinct permutations of {5,6,7,8}) and the latter ones (distinct permutations of {1,2,3,5} × distinct permutations of {4,6,7,8}) are non-equivalent, i.e., independent sets. In the exactly same manner, we can conduct two element swapping. However, consider 3 element swaps such as swapping between 1,2,3 from {1,2,3,4} and 5,6,7 from {5,6,7,8}. In this case, 3 element swapping is exactly same as 1 element swapping (swapping between element 4 from {1,2,3,4} and element 8 from {5,6,7,8}). Therefore, in general, when we have n elements in the set, we only need to consider 1, 2, 3, …, n/2-1, n/2 element swaps. Now, we prove that the number of non-equivalent permutations produced through the above swapping processes is exactly same as $(n!)^2/2^{n-1}$.

Define the number of non-equivalent placements of n nodes as M(n). Then,

$$M(n) = M(\frac{n}{2})\cdot M(\frac{n}{2}) + {}_{\frac{n}{2}}C_1 \cdot {}_{\frac{n}{2}}C_1 \cdot M(\frac{n}{2})\cdot M(\frac{n}{2}) + {}_{\frac{n}{2}}C_2 \cdot {}_{\frac{n}{2}}C_2 \cdot M(\frac{n}{2})\cdot M(\frac{n}{2}) + \bullet\bullet\bullet + {}_{\frac{n}{2}}C_{\frac{n}{4}} \cdot {}_{\frac{n}{2}}C_{\frac{n}{4}} \cdot M(\frac{n}{2})\cdot M(\frac{n}{2})\cdot \frac{1}{2} \qquad (1)$$

The first term in the right side means that no element swap happened between two sets of size n/2. Second term in the right side came from one element swap between two halves. Same explanation applies for the rest of terms.

Rearranging the above equation, we get:

$$\frac{M(n)}{M(\frac{n}{2})^2} = {}^1\!/\!_2\left[{}_{\frac{n}{2}}C_0 \cdot {}_{\frac{n}{2}}C_0 + {}_{\frac{n}{2}}C_1 \cdot {}_{\frac{n}{2}}C_1 + {}_{\frac{n}{2}}C_2 \cdot {}_{\frac{n}{2}}C_2 + \bullet\bullet\bullet + {}_{\frac{n}{2}}C_{\frac{n}{2}-1} \cdot {}_{\frac{n}{2}}C_{\frac{n}{2}-1} + {}_{\frac{n}{2}}C_{\frac{n}{2}} \cdot {}_{\frac{n}{2}}C_{\frac{n}{2}} \right] \qquad (2)$$

And,

$$(x+y)^n = {}_nC_0 \cdot x^0 \cdot y^n + {}_nC_1 \cdot x^1 \cdot y^{n-1} + {}_nC_2 \cdot x^2 \cdot y^{n-2} + \bullet\bullet\bullet + {}_nC_{n-1} \cdot x^{n-1} \cdot y^1 + {}_nC_n \cdot x^n \cdot y^0 \qquad (3)$$

By squaring both of the terms in the equation (3), we obtain:

$$(x+y)^{2n} = \left({}_nC_0 \cdot x^0 \cdot y^n + {}_nC_1 \cdot x^1 \cdot y^{n-1} + {}_nC_2 \cdot x^2 \cdot y^{n-2} + \bullet\bullet\bullet + {}_nC_{n-1} \cdot x^{n-1} \cdot y^1 + {}_nC_n \cdot x^n \cdot y^0\right)^2 \qquad (4)$$

In (4), the coefficient of $x^n \cdot y^n$ term in the left side can be computed as ${}_{2n}C_n$. And, from the right side of the above equation, the coefficients of $x^n \cdot y^n$ term can be obtained as:

$${}_nC_0 \cdot {}_nC_n + {}_nC_1 \cdot {}_nC_{n-1} + {}_nC_2 \cdot {}_nC_{n-2} + \bullet\bullet\bullet \quad = \quad {}_nC_0{}^2 + {}_nC_1{}^2 + {}_nC_2{}^2 + \bullet\bullet\bullet = {}_{2n}C_n \qquad (5)$$

We use (4) and (5) to transform (2) into:

$$\frac{M(n)}{M(n/2)^2} = \frac{1}{2}\,{}_nC_{\frac{n}{2}} \qquad (6)$$

And,

$$\frac{M(n)}{M(n/2)^2} = \frac{n!^2}{2^{n-1}} \bigg/ \left(\frac{(n/2)!^2}{2^{n/2-1}}\right)^2 = \frac{1}{2}\cdot\frac{n!}{\left(\frac{n}{2}!\right)^2} = \frac{1}{2}\,{}_nC_{\frac{n}{2}} \qquad (7)$$

Now, we know that the value from (6) and (7) are same each other. Therefore, we proved that the number of non-equivalent permutation pairs that are produced by our method is exactly same as n!^{2}/2^{n-1}.

General procedures to make non-equivalent permutations are: It starts from the highest level swapping, and gradually going down to the lower level swaps as follows.

First, pick and swap k_1 element(s) each from the first n/2 set and the second n/2 set. $(0 \leq k_1 \leq n/4)$

Second, pick and swap k_2 element(s) each from the first n/4 set and the second n/4 set. $(0 \leq k_2 \leq n/8)$

Third, pick and swap k_3 element(s) each from the third n/4 set and the fourth n/4 set. $(0 \leq k_3 \leq n/8)$

Fourth, pick and swap k_4 element(s) each from the first n/8 set and the second n/8 set. $(0 \leq k_4 \leq n/16)$

Fifth, pick and swap k_5 element(s) each from the third n/8 set and the fourth n/8 set. $(0 \leq k_5 \leq n/16)$

Sixth, pick and swap k_6 element(s) each from the fifth n/8 set and the sixth n/8 set. $(0 \leq k_6 \leq n/16)$

Seventh, pick and swap k_7 element(s) each from the seventh n/8 set and eighth n/8 set. $(0 \leq k_7 \leq n/16)$

And so forth.

Our algorithm for producing non-equivalent permutations significantly reduce the searching space, i.e., from n! to $(n!)/2^{n-1}$. But, the number of reduced search space is still considerably large, since the magnitude of n! is substantial intrinsically . Therefore, we invented some approximation algorithm for generating more promising pairs first with higher priority rather than generating all the placement pairs. And later, we will conduct performance comparison between approximation method with the optimal solution through exemplary simulation results.

7. Approximation approach for finding placements.

Searching all the possible placements are extremely time consuming processes as mentioned earlier. For example, suppose that we have 32 processing units in PIM architecture. Then, the total number of possible non-equivalent placement pairs is 3.224×10^{61}. And, we also have to compute estimated communication cost for each placement pairs, which takes additional time. Therefore, we consider an approximation method for finding some satisfactory placement pairs within a very short time. We explain how approximation method is developed.

Top down approximation

As an alternative method for exhaustive search, we develop a top-down approximation, which conducts swaps, starting from the highest level, all the way down to the lowest level according to the cost function. The highest level swap is the swapping between blocks with the longest distance each other. As an example, see the Figure 12. The distance between A_1 and A_7 is one of the longest ones in the architecture. It is same for between A_4 and A_7. Communication distance between A_1 and A_2 is one of the shortest ones as shown in the figure. Since the highest level data communication deprives PIM architecture of parallelism severely, we try to minimize such situation as possible. After handled properly, try to reduce the next highest level communication. These procedures are continued until the lowest level is reached.

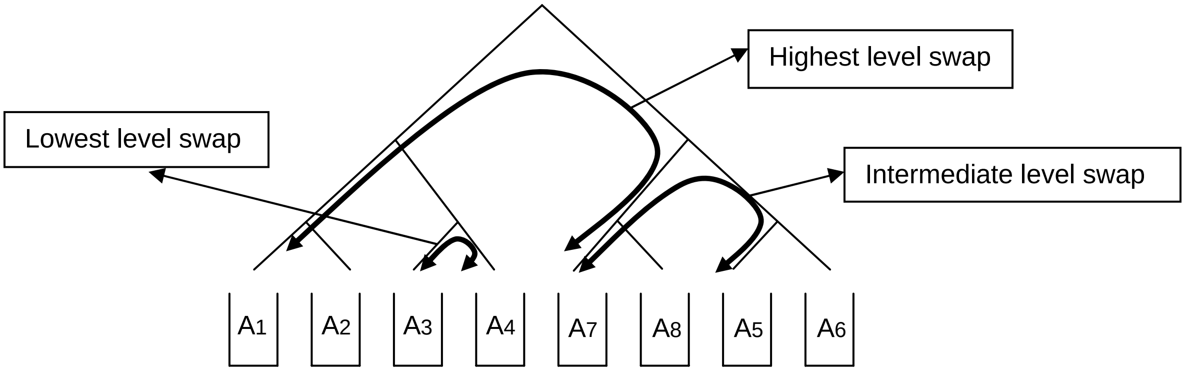


Figure 12 Illustration of the differences in communicational distance

When we apply this approximation method, the placement assignment of geometry processor is fixed, and the placement of rasterization processor is updated according to the cost function, starting from the highest level (level $\log_2{}^n$) to the lowest level (level 0). The cost function is defined as:

$$
\begin{aligned}
&\mathit{Cost(k, i)} = \sum D_{m,i} - \sum D_{p,i} \\
&\mathit{where,} \\
&\left\{ D_{m,i} \middle| m \in \left\{ \mathit{node\ ID\ with\ the\ communication\ distance\ of\ level\ } \log_2^n - k \mathit{\ between\ node\ m\ and\ node\ i} \right\} \right\} \\
&\left\{ D_{p,i} \middle| p \notin \{ m \} \right\} \\
&k = 0,1,2,3,\ldots
\end{aligned}
$$

Where, $D_{i,j}$ is the number of polygons that stays at node i for geometry processing and are transferred to node j for rasterization processor.

The meaning of our cost function is that we find an element that has less relationship with its currently neighboring elements, and rather, has close relations with the elements in a far-away side. The first term in the right side at the general cost function formula is to sum up the far-away side elements, whereas the second term is to sum up the values with currently neighboring elements. Therefore, the more positive for the cost function, the more unstable for the current assignment.

Placements of each partition are rearranged as top-down approach proceeds.

Fixed placement

RAS GP	node 1	Node 2	node 3	…
node 1	D_{11}	D_{12}	…	
node 2	D_{21}	D_{22}	…	
node 3	D_{31}			
…				

Figure 13 LUT used for approximation method for finding near-optimal placement – Fixed placement for geometry processor and progressively updated placement for rasterization processor.

For further clarification, we see the case of 8 processing units. We first compute the cost values for each element in {1, 2, 3, 4} and {5, 6, 7, 8} respectively – for example, the cost value of node 1 is $D_{51}+D_{61}+D_{71}+D_{81}-D_{11}-D_{21}-D_{31}-D_{41}$. Likewise, that of node 7 is $D_{17}+D_{27}+D_{37}+D_{47}-D_{57}-D_{57}-D_{67}-D_{87}$. (Figure 13) Compute all the cost values of 8 nodes in the same manner. Then, choose the two largest ones, one from {1, 2, 3, 4} and one from {5, 6, 7, 8}. If the sum of two is positive, swap two elements. Then, choose the second largest one from {1, 2, 3, 4} and the second largest one from {5, 6, 7, 8}. If the sum of them is also positive, swap them. Continue until the sum of them becomes negative. When everything is processed, go down to one level lower. Do the same procedure for {1, 2} and {3, 4}. When swapping procedures of the two sets are done, then, work on the swapping between {5, 6} and {7, 8} based on the cost function. Once finished, go down to the lowest level, and check the possible swapping between {1} and {2}. Do the same procedure for the rest of the lowest level swaps.

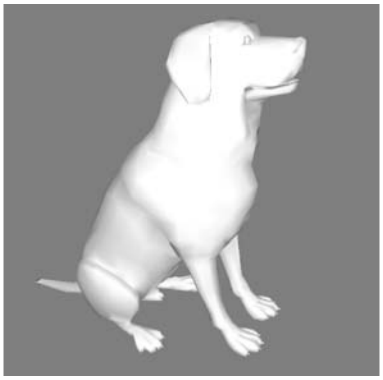
In the above approximation method, we fixed the placement in geometry processor while we update the placement in rasterization processor. But, conversely, the placement assignment of rasterization processor can be fixed, and the placement of geometry processor is determined according to the approximation method.

This top-down method was experimented for a particular case, and produced a good result. Refer to the simulation result of the subsequent section.

8. Experiment result

We conducted simulations for a simple dog object as shown in the Table 1. The total number of vertex for the dog object at below is 1556, and the average size of polygon is 0.051747. This object consists of a triangle polygon mostly. In the experiments, we assumed that the polygon sizes were approximately same for the computational convenience. And, the number of processing nodes in PIM architecture was eight. Since the scene is static, the past, present and future information on this scene is time-invariant. Therefore, the preciseness of 3D distribution statistics along with their mapping relations between GP and RAS won't be an issue here (Figure 6).

Table 1. Simulation data to be used for finding an optimal partitioning method in object space and image space along with polygon mapping information between the chosen partitioning methods

	Vertex counst for each partition method w.r.t. geometry processor	
	*x-x-x cut*	*10, 9, 60, 116, 272, 180, 511, 398*
	x-x-y cut	*19, 174, 2, 370, 82, 301, 608*
	x-x-z cut	*10, 9, 89, 87, 232, 220, 456, 453*
	x-y-y cut	*143, 50, 2, 514, 157, 154, 536*
	x-y-z cut	*98, 95, 1, 1, 343, 328, 345, 345*
	x-z-z cut	*36, 63, 61, 35, 215, 473, 460, 213*
	y-y-y cut	*551, 106, 141, 66, 46, 110, 269, 267*
	y-y-z cut	*335, 322, 106, 101, 80, 76, 266, 270*
	y-z-z cut	*204, 237, 222, 201, 47, 299, 299, 47*
	z-z-z cut	*70, 181, 251, 285, 274, 247, 179, 69*

Vertex counts for each partition method w.r.t. rasterization processor	
x-x-x cut	*11, 15, 101, 236, 180, 383, 209, 418*
x-x-y cut	*26, 11, 326, 102, 461, 611, 16*
x-x-z cut	*26, 176, 161, 332, 231, 12, 615*
x-y-y cut	*11, 180, 172, 551, 162, 170, 307*
x-y-z cut	*11, 176, 176, 37, 676, 307, 170*
x-z-z cut	*78, 98, 125, 62, 168, 176, 541, 305*
y-y-y cut	*231, 320, 100, 73, 137, 213, 288, 191*
y-y-z cut	*551, 37, 136, 212, 138, 271, 208*
y-z-z cut	*37, 405, 282, 246, 237, 261, 85*
z-z-z cut	*87, 159, 122, 152, 312, 354, 240, 127*

3D Mapping LUT (z-z-z to y-y-y)								
RAS *GP*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*
1	*0*	*0*	*0*	*1*	*0*	*1*	*12*	*55*
2	*0*	*14*	*9*	*16*	*23*	*28*	*12*	*79*
3	*17*	*55*	*33*	*11*	*9*	*28*	*49*	*48*
4	*56*	*105*	*17*	*7*	*21*	*10*	*60*	*9*
5	*79*	*90*	*13*	*7*	*18*	*11*	*56*	*0*
6	*64*	*40*	*10*	*7*	*26*	*39*	*60*	*0*
7	*15*	*16*	*17*	*23*	*28*	*41*	*39*	*0*
8	*0*	*0*	*1*	*1*	*12*	*55*	*0*	*0*

In the above simulation results, the data in the upper right shows the vertex counts at each geometry processor with respect to each fixed partitioning method, and the data in the lower left indicates the vertex counts at each rasterization processor for each partitioning method. Based on the statistics obtained from the experiments, we conclude:

1. Recommended partitioning of geometry processor: z-z-z
2. Recommended partitioning of rasterization processor: y-y-y

The data in the lower right represents the polygon mapping information between z-z-z partitions of geometry processor and y-y-y partitions of rasterization processor. Based on the mapping table, we apply top-down approach and get the results:

GP Block ID	*Placement in PIM*	*RAS Block ID*	*Placement in PIM*
1	*1*	*1*	*5*
2	*2*	*2*	*4*
3	*3*	*3*	*3*
4	*4*	*4*	*1*
5	*5*	*5*	*7*
6	*6*	*6*	*8*
7	*7*	*7*	*6*
8	*8*	*8*	*2*

The detailed procedures for obtaining the above placement result are explained at the Appendix A. The following data is the globally optimal solution when exhaustive search is applied:

GP Block ID	*Placement in PIM*	*RAS Block ID*	*Placement in PIM*
1	*1*	*1*	*6*
2	*2*	*2*	*5*
3	*3*	*3*	*3*
4	*4*	*4*	*2*
5	*5*	*5*	*7*
6	*6*	*6*	*8*
7	*7*	*7*	*4*
8	*8*	*8*	*1*

Relative communication cost from the top-down approach was 121%, assuming that the communication cost of the globally optimized placement is 100%. Considering time efficiency of our top-down approach and near-optimality of the algorithm, our approach looks very promising. Even in this simple case of 8 processing units, the exhaustive searches for all the 12,700,800 possible pairs of placements, including the computation of communication costs for each case, do take significant amount of time.

9. Computation of overall processing time.

Once we decide certain partitioning and placement method, we can compute overall processing time for each stage. Defining g_i as the number of vertexes processed by geometry processor i, the computation time for geometry processing can be obtained:

$$T_{GP} = \alpha_{GP} \cdot \max\{g_1,\ g_2,\ g_3,\ \bullet\bullet\bullet,\ g_n\}$$

When the work-load for each geometry processor is different, computation time on geometry processing stage depends on the heavily loaded node, so that max function can be seen at the above equation. In the equation, α_{GP} takes into account stalling in pipelining stages. Graphic processing requires numerous arithmetic operations from addition/subtraction, multiplication to division of integers or floating point numbers. The latency of FPU (Floating point unit) like multiplication or divisions is much higher than that of integer units. Under those circumstances, stalling can occur due to dependencies. Those factors are considered in α_{GP}. With the same analogy to the geometry processing time, rasterization processing time can be computed as:

$$T_{RAS} = \alpha_{RAS} \cdot \max\{r_1,\ r_2, r_3,\ \bullet\bullet\bullet,\ r_n\}$$

Where, r_i is defined as the area sum of polygons processed by rasterization processor I, since rasterization process is size-dependent of polygon.

Now, we consider communication cost that comes from the stage between the end of geometry processor and the beginning of rasterization processor (sort-middle). We start from 8 processing unit case as an example. The definition of $a_{i,j}$ is the number of polygon that is processed by geometry processor i, and transferred to rasterization processor j. The communication cost can be computed as follows.

Define the level 1 communication as adjacent nodes' data exchange. Similarly, define the level 2 communication as the communication through the same grand-parental branch. Keep defining the level of communication until it reaches up to the highest level, $\log_2^n$.

Now, we can estimate communication cost like:

Level 1: $\max(a_{1,2}, a_{3,4}, a_{5,6}, a_{7,8}) + \max(a_{2,1}, a_{4,3}, a_{6,5}, a_{8,7})$

Level 2: $\max(a_{1,3}, a_{5,7}) + \max(a_{3,1}, a_{7,5}) + \max(a_{1,4}, a_{5,8}) + \bullet\bullet\bullet$

Level 3: $\max(a_{1,5}) + \max(a_{1,6}) + \max(a_{1,7}) + \bullet\bullet\bullet$

Summing up all the values from each level produces overall communication cost between geometry processor and rasterization processor. In general, assuming that we have n processing units, we can compute each level's cost:

Level 1: $max\left\{\ a_{i,j} \middle|\ i = 2k-1,\ j = 2k \ where\ k = 1,2,3,...,\frac{n}{2}\ \right\}$

Level 2:

$$\begin{aligned}
&max\left\{\, a_{i,j} \,\middle|\, i = 4k-3,\ j = 4k-1 \text{ where } k = 1,2,3,\ldots,\frac{n}{4} \right\} + \\
&max\left\{\, a_{i,j} \,\middle|\, i = 4k-3,\ j = 4k \quad \text{ where } k = 1,2,3,\ldots,\frac{n}{4} \right\} + \\
&max\left\{\, a_{i,j} \,\middle|\, i = 4k-2,\ j = 4k-1 \text{ where } k = 1,2,3,\ldots,\frac{n}{4} \right\} + \\
&max\left\{\, a_{i,j} \,\middle|\, i = 4k-2,\ j = 4k \quad \text{ where } k = 1,2,3,\ldots,\frac{n}{4} \right\}
\end{aligned}$$

$$\vdots$$

Level p: $$\sum_{0\le m,n\le 2^{p-1}-1} max\left\{\, a_{i,j} \,\middle|\, i = 2^p k - 2^p + 1 + m,\ j = 2^p k - 2^{p-1} + 1 + n \text{ where } k = 1,2,3,\ldots,\frac{n}{2^p} \right\}$$

As for a relative time-complexity of geometry processing and rasterization processing, when 3D scenes are delineated by coarse polygons, rasterization process will dominate the overall time, since rasterization processors will spend the substantial amount of time for shading large polygons whereas the number of vertexes in the scenes becomes relatively small. In the case, we can say that the system is rasterization dominant. But, modern 3D animations draw very fine and detailed objects with very small sizes of polygons for producing extremely high quality scenes. In this case, the workload for shading single polygon is very small. Therefore, the time complexity of geometry processing will be comparable to that of rasterization processors. In this situation, it is hard to say that which one is dominant.

10. Summary and Discussion

In this paper, we explored efficient partitioning and placement methodologies for PIM graphic architecture and showed that our algorithms for partitioning and placement methodologies are efficient in terms of computational time and hardware complexity. We focused on fixed-partitioning methodologies, and developed search space reduction algorithms – One is scene-dependent and another one is scene-independent. As for placement, we could cut down on the number of search space substantially by using the concept of branch alternation. Since the size of search space for placement is substantially large by its own nature even after applying placement reduction techniques, we developed an approximate method in order to search close-to-optimal placement very efficiently and quickly (we call it a top-down approach). According to the simulation result, top-down approach performed satisfactorily.

We constructed a special informational unit that contains a distributional statistics on polygons for both geometry processing and rasterization processing along with their mapping information. By using this informational unit, we could decide what would be an efficient partitioning and placement methodologies for given scene. Also, it enabled us to estimate overall processing times in parallel PIM architecture.

In this paper, we limited our studies to binary interconnect types, but we plan to expand those ideas for other interconnect type such as grid and cube.

11. Reference

[1] International Technology Roadmap for Semiconductors, www.itrs.net/

[2] Keith Diefendorff, et al., How Multimedia Workloads Will Change Processor Design, *IEEE Computer*, Vol. 30, No 9, p.43-45, September 1997.

[3] D. Burger, et al., Memory Bandwidth Limitations of Future Microprocessors, In Proceedings of the 23rd International Symposium on Computer Architecture, p.78–89, 1996.

[4] Patterson D, et al., A Case for Intelligent DRAM: IRAM, IEEE Micro, April 1997.

[5] Mark Oskin, et al., Active Pages: A Computation Model for Intelligent Memory, In Proceedings of the 23rd. International Symposium on. Computer Architecture, p.192-203, 1998

[6] Yi Kang, et al., FlexRAM: Toward an Advanced Intelligent Memory System, IEEE International Conference on Computer Design, p.192, 1999.

[7] Jung-Yup Kang, et al., An Efficient PIM (Processor-In-Memory) Architecture for Motion Estimation. In 14th IEEE International Conference on Application-Specific Systems, Architectures, and Processors, p.282-292, 24-26 June 2003.

[8] Jung-Yup Kang, et al., Accelerating the Kernels of BLAST with an Efficient PIM (Processor-In-Memory) Architecture, In 3rd International IEEE Computer Society Computational Systems Bioinformatics Conference, p.552-553, 16-19 August 2004.

[9] John Montrym, et al., The GeForce 6800, IEEE Micro**,** Vol 25, No 2, p. 41-51, Mar 2005

[10] Emmett Kilgariff, et al., The GeForce 6 Series GPU Architecture, download.nvidia.com/developer/GPU_Gems_2/GPU_Gems2_ch30.pdf

[11] Molner, et. al., A sorting classification of parallel rendering , Computer Graphics and Application, IEEE, Vol 14, No 4, p.23-32, 1994

[12] S. Whitman, Dynamic load balancing for parallel polygon rendering, IEEE Computer Graphics and Applications, vol 14, No 4, p41-48, 1994.

[13] S. Whitman, Parallel Graphics Rendering Algorithms, Proc. 3rd Eurographics Workshop on Rendering, Consolidation Express, Bristol, UK, pp.123-134, May 1992

[14] Tahsin M. Kurc, et al., Object-Space Parallel Polygon Rendering on Hypercubes, *Computers & Graphics* , vol. 22, No 4, p.487-503, 1998

Appendix A. *Procedure of top-down approach from section 7*

1. Original setup

	1	2	3	4	5	6	7	8
1	0	0	0	1	0	1	12	55
2	0	14	9	16	23	28	12	79
3	17	55	33	11	9	28	49	48
4	56	105	17	7	21	10	60	9
5	79	90	13	7	18	11	56	0
6	64	40	10	7	26	39	60	0
7	15	16	17	23	28	41	39	0
8	0	0	1	1	12	55	0	0

2. Swap 1 and 8.

	8(1)	2	3	4	5	6	7	1(8)
1	55	0	0	1	0	1	12	0
2	79	14	9	16	23	28	12	0
3	48	55	33	11	9	28	49	17
4	9	105	17	7	21	10	60	56
5	0	90	13	7	18	11	56	79
6	0	40	10	7	26	39	60	64
7	0	16	17	23	28	41	39	15
8	0	0	1	1	12	55	0	0

3. Swap 2 and 4

	8(1)	4(2)	3	2(4)	5	6	7	1(8)
1	55	1	0	0	0	1	12	0
2	79	16	9	14	23	28	12	0
3	48	11	33	55	9	28	49	17
4	9	7	17	105	21	10	60	56
5	0	7	13	90	18	11	56	79
6	0	7	10	40	26	39	60	64
7	0	23	17	16	28	41	39	15
8	0	1	1	0	12	55	0	0

4. Swap 6 and 8

	8(1)	4(2)	3	2(4)	5	1(6)	7	6(8)
1	55	1	0	0	0	0	12	1
2	79	16	9	14	23	0	12	28
3	48	11	33	55	9	17	49	28
4	9	7	17	105	21	56	60	10
5	0	7	13	90	18	79	56	11
6	0	7	10	40	26	64	60	39
7	0	23	17	16	28	15	39	41
8	0	1	1	0	12	0	0	55

5. Swap 5 and 7

	8(1)	4(2)	3	2(4)	7(5)	1(6)	5(7)	6(8)
1	55	1	0	0	12	0	0	1
2	79	16	9	14	12	0	23	28
3	48	11	33	55	49	17	9	28
4	9	7	17	105	60	56	21	10
5	0	7	13	90	56	79	18	11
6	0	7	10	40	60	64	26	39
7	0	23	17	16	39	15	28	41
8	0	1	1	0	0	0	12	55

6. Swap 1 and 2

	4(1)	8(2)	3	2(4)	7(5)	1(6)	5(7)	6(8)
1	1	55	0	0	12	0	0	1
2	16	79	9	14	12	0	23	28
3	11	48	33	55	49	17	9	28
4	7	9	17	105	60	56	21	10
5	7	0	13	90	56	79	18	11
6	7	0	10	40	60	64	26	39
7	23	0	17	16	39	15	28	41
8	1	0	1	0	0	0	12	55

Swap 5 and 6

	4(1)	8(2)	3	2(4)	1(5)	7(6)	5(7)	6(8)
1	1	55	0	0	0	12	0	1
2	16	79	9	14	0	12	23	28
3	11	48	33	55	17	49	9	28
4	7	9	17	105	56	60	21	10
5	7	0	13	90	79	56	18	11
6	7	0	10	40	64	60	26	39
7	23	0	17	16	15	39	28	41
8	1	0	1	0	0	0	12	55

Done