International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:2, No:2, 2008

A PIM (Processor-In-Memory) for Computer Graphics: Data Partitioning and Placement Schemes

Jae Chul Cha, and Sandeep K. Gupta

***Abstract*—**The demand for higher performance graphics continues to grow because of the incessant desire towards realism. And, rapid advances in fabrication technology have enabled us to build several processor cores on a single die. Hence, it is important to develop single chip parallel architectures for such data-intensive applications. In this paper, we propose an efficient PIM architectures tailored for computer graphics which requires a large number of memory accesses. We then address the two important tasks necessary for maximally exploiting the parallelism provided by the architecture, namely, partitioning and placement of graphic data, which affect respectively load balances and communication costs. Under the constraints of uniform partitioning, we develop approaches for optimal partitioning and placement, which significantly reduce search space. We also present heuristics for identifying near-optimal placement, since the search space for placement is impractically large despite our optimization. We then demonstrate the effectiveness of our partitioning and placement approaches via analysis of example scenes; simulation results show considerable search space reductions, and our heuristics for placement performs close to optimal – the average ratio of communication overheads between our heuristics and the optimal was 1.05. Our uniform partitioning showed average load-balance ratio of 1.47 for geometry processing and 1.44 for rasterization, which is reasonable.

***Keywords*—**Data Partitioning and Placement, Graphics, PIM, Search Space Reduction.

I. Introduction

EVEN though advances in VLSI technology have provided higher computational capabilities and larger memories [1], most classical architectures fail to sufficiently exploit these advancements for efficient parallel computations due to processor-memory bottlenecks [35]. Hence, PIM architectures, which 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 similar types of architectures have been proposed to exploit VLSI scaling [4][5][6][7][8].

Today's sophisticated 3D animation films have work-loads that are too large to process in real time. Therefore, they are rendered in a non-real-time (off-line) manner [32] – i.e., each scene is created separately, and such digital scenes are converted to analog films, which are played in theatres. However, with advances in VLSI technologies, graphics processors will be able to support real-time animations in the future if the bottlenecks can be eliminated. Based on this perspective, we explore a PIM architecture tailored for parallel graphics processing.

Authors are with Department of Electrical Engineering, University of Southern California, Los Angeles, CA90089.

To achieve maximum performance, the workload must be evenly divided among processors, and data placement and task assignment must minimize the communication overhead between processors. We propose partitioning and placement methodologies that require low hardware complexities, but provide good performance.

The rest of paper is organized as follows. Section II briefly describes the proposed architecture. Section III explains some of the important computer graphics concepts for better understanding of the partitioning and placement. Section IV describes how we obtain the information required to identify efficient partitioning and placement. Section V explains our efficient partitioning and placement schemes. Section VI presents the performance results on the load balancing and communication overheads. Section VII summarizes our results and proposes future work.

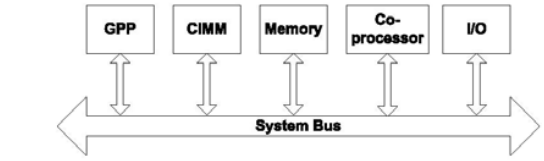
II. Proposed Architecture

We envision that the overall system is organized to include GPP (general purpose processor), a CIMM (computation-in-memory-modules), co-processor(s), memory, and I/O devices as shown in Fig. 1(a). CIMM consists of ECME (embedded computing memory element) modules, DDDR (decoder/data-distributor + router), and controller as shown in Fig. 1(b). Fig. 1(b)(i) shows a classical decoder tree that connects memory modules to external port, whereas Fig. 1(b)(ii) shows how we modify the organization of a classical memory to obtain the CIMM.

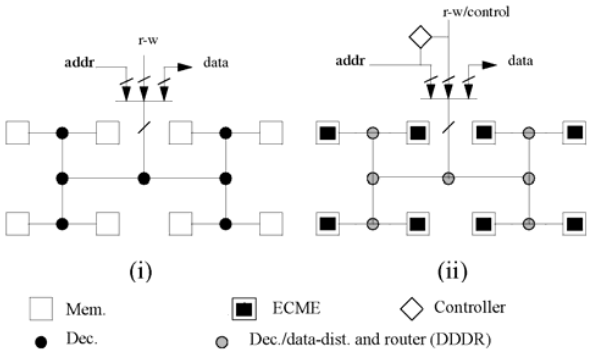
DDDR – decoder, data-distributor, and router – in the ECME memory decoder tree enables efficient communications to/from the external port as well as between memory blocks. When the memory is used in the normal mode, each DDDR operates as a classical decoder, i.e., when activated, each decoder selects and activates its left or right child, depending on whether the corresponding address symbol is a '0' or a '1'. However, a DDDR can also work in one of many new data-distribution modes. For example, during a write operation, if a DDDR is activated with an address symbol '*', it selects and activates both its children, resulting in a two-way broadcast of the data to be written. This mode facilitates data replication, which is used to duplicate graphic primitives that cross boundaries in partitioned scenes without incurring performance overheads. DDDR can be also activated with an address symbol 'lr' to select and activate its left child in the read mode and its right

child in the write mode. In this mode, the data forwarded to the DDDR by its left child is forwarded to its right child for writing. Such operations can be processed in parallel at multiple DDDR nodes in the decoder tree, which expedites inter-memory transfer of graphics primitives.

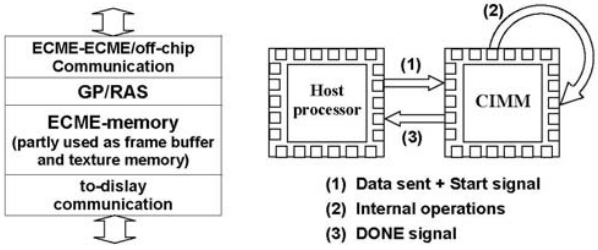
Fig. 1(c) shows structure of the ECME – each ECME consists of a processor that can act either as a GP (geometry processor) or a RAS (rasterization processor), a DRAM (memory block) and a controller. The details of internal and external data flows in ECME are explained in Fig. 2. By designing ECME's with suitably parameterized controls, much of the control logic can be migrated from individual ECME's to controllers at higher levels of the decoder tree due to commonality in their control logic. The controller shown at the input part in Fig. 1(b) embodies such control logic.



(a) Overall Configurations



(b) Internal Organization of CIMM



(c) Inside of ECME (d) CIMM Execution Flow

Fig. 1 Data intensive computing Architecture

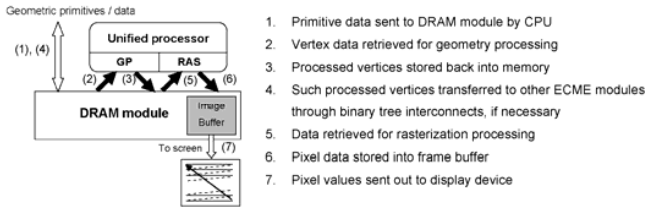


Fig. 2 Internal/external data flows in ECME

Fig. 1(d) depicts the flow of execution for kernels – the host processor sends all the vertex data of a scene to memory modules within the CIMM. The host processor then commands the CIMM to start the kernel operations. Each ECME begins to execute the given operations once it receives signal from CIMM controller. Each ECME notifies the CIMM when it completes its job. CIMM then notifies the host processor that all the jobs are finished and that the contents of frame buffers are ready to be rendered in the display device.

In our architecture, instead of incorporating separate GP and RAS processors, we use a *unified* processor, which can be used for either GP or RAS [36][37]. In many recent graphic processing units with *non-unified* architectures, i.e., architectures that use distinct processors for GP and RAS, the ratio of the number of vertex shaders in GP stage and that of pixel shaders in RAS stage are 1:3 approximately [38]. This is because, in many cases, computational needs for RAS stage are higher than those for GP in about this proportion. Therefore, if vertex shaders are overloaded for some scenes, performance will drop regardless of the number of available pixel shaders. In contrast, a unified architecture can alleviate such bottlenecks by allocating all the shader resources for vertex shading as well as pixel shading as necessary.

Graphic processing requires intensive processor-memory communication for data transfers, texturing, and buffering. Thus, a PIM architecture with low memory access time can boost system's performance.

III. Background on Computer Graphics

Most objects are represented using polygons, typically triangles. In order to add realism to the objects, each polygon is filled up with colors and textures as well as effects like shadows and light. 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 [23]. When scenes have extremely high complexities, a single processor cannot meet the computational needs, especially in the presence of real- time deadlines [26]. In 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. Subsequently, the representation is transformed into world-coordinate, where every object is defined relative to a common origin. Finally, it is transformed to the viewer-coordinate as defined by the user's perspective [24]. Such transformations are carried out by applying various matrix operations to each vertex during GP stage.

Once GP stage completes its operations on vertices, the RAS stage takes those transformed vertices and shades polygons by filling in the interiors of polygons with the colors and textures and converts such primitives into pixel values that are stored in the frame buffer for display. The coordinates for GP processing are called object space, whereas those for RAS processing are called image-space [23].

Initial inputs to GP 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 [23]. The normal vector is used for the lighting calculation by considering two vectors, namely, the light direction and the normal vector of surface. The vertices from the same polygon are assigned the same ID to keep track of the vertices that belong to a polygon [24].

The size of a polygon doesn't affect the computational complexity of GP, since the complexity of GP is only related to the number of vertices. However, the complexity of RAS depends on the size of a polygon, since the complexity of shading (and other operations if carried out) is a function of the polygon's size [33]. As smaller primitives (polygons) provide higher quality images, the computational needs for graphics will continue to increase for the foreseeable future.

Polygons undergo numerous spatial transformations, so that they need to be rearranged somewhere within the pipeline in a parallel architecture. According to the location of data rearrange- ment, three types of schemes exist – sort first (before geometry processing), sort middle (between geometry and rasterization processing), sort last (after rasterization) [11]. Among three sorting strategies, sort-middle has thus far resulted in many practical implementations [18][19][20][21][22]. Therefore, we use sort-middle in our architecture.

As for a relative time-complexity of GP and RAS processing, when 3D scenes are delineated by coarse polygons, RAS process will dominate the overall time, since RAS processors will spend the substantial amount of time for shading large polygons whereas the number of vertices in the scenes becomes relatively small. In such a case, we can say that the system is rasterization dominant. In contrast, if system is required to draw very fine and detailed objects with very small polygons for producing extremely high quality scenes, the workload for shading a single polygon is very small. In this case, the time complexity of GP processing can be comparable to that of RAS processing.

IV. Pre-partitioning and Placement Phase – Construction of Reference

Parallel processing is capable of providing significant performance gains in data intensive applications. However, the efficiency of parallel processing is most frequently dependent on partitioning and placement of the data. In section V, we will discuss efficient partitioning and placement methodologies. In this section, we describe how we gather information that will be used for partitioning and placement.

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 [34]. Such an approach will be effective except when an abrupt change of scene occurs.

We construct a 3D polygon distribution map when the 3D space is partitioned into small cubes (Fig. 3). This distribution map contains (a) the number of polygons in each fraction of the space (cube) for geometry and rasterization processors, and (b) the mapping which captures the number of polygons that are transferred from cube i during geometry processing to cube j during rasterization. To construct this map, 3D scenes in the object-space and the image-space are partitioned into small cubes, based on the desired resolution and hardware availability. The mapping information between cubes from the object-space and those from the image-space are stored in a look-up table (LUT) as depicted in Fig. 3. The table contains the number of polygons and the sum of areas of polygons. The numbers of polygons are used for partitioning frames for GP processing whereas the area sums of polygons are used for partitioning for RAS processing. Once we decide the partitionings for GP and RAS, we identify placements with minimal communication costs.

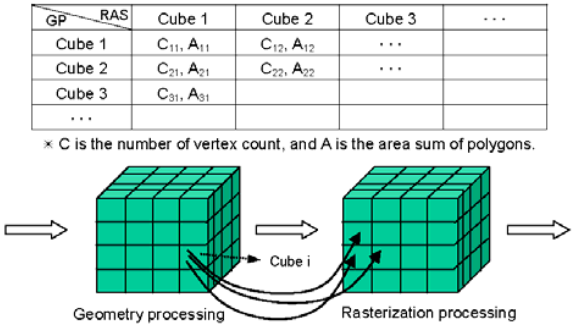


Fig. 3 Look-up table (LUT) construction that contains 3D mapping relations between object space and image space. – Vertex counts are used for determining the workload division on GPs whereas area sums are used for determining the workload division on RASs

In a three dimensional space, the area of a triangle can be obtained by using pythagorean sum of the respective projection on the three principal planes. This computation takes 17 additions /subtractions, 10 multiplications, and 1 square root operation. Assuming that the total number of polygons in a scene is n_p, the time complexity for computing the area sum stored in the cubes is simply $O(n_p)$. This amount of workload is insignificant compared to the complexities of GP and RAS procedures. The memory complexity of LUT is $O((n_c)^2)$ assuming that the number of cubes is n_c. The 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 unit used for the area calculation.

V. Partitioning and Placement Phase

We determine the partitioning of the object-space and that of image-space independently each other to achieve workload balancing and then determine the placement that minimizes the communication cost between GP and RAS. Such a decision making process – determine the partitioning first and then determine the placement – may not produce globally optimal results. However, this allows us to find good solutions efficiently and quickly at relatively low complexity by reducing the search space.

A. Partitioning Methodology

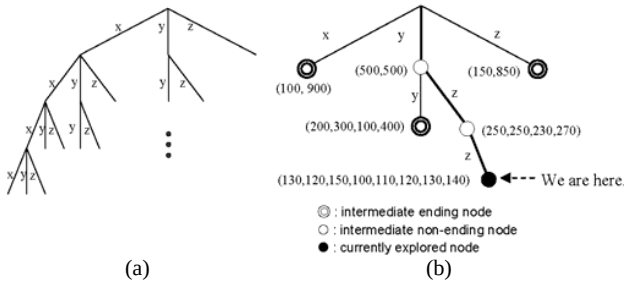
The partitioning procedure can be static or adaptive [31]. The static method is to divide the scene in a deterministic way, whereas adaptive method decides its partitioning dynamically [13][27][28][29][30]. In this study, we present a hybrid

partitioning methodology – only x-, y-, z-directional cuts are allowed (static), while the directions of cuts are determined dynamically according to the scene's characteristics (adaptive) – subject to the following partitioning rules.

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

The size of search space for all the possible partitioning methods increases exponentially with the number of processing elements. Thus, we present an approach to reduce search space. This approach consists of two parts, namely, a scene independent scheme and a scene dependent bounding.

A bi-partitioning tree can be drawn for generating recursive cuts (Fig. 4(a)) for our hybrid method. 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. The number of possible cuts in a complete partitioning tree can be computed as $3^{\log_2(n)}$. However, we observe that the sequence of cuts doesn't matter, since bi-partitioning in each dimension is independent and bi-partitioning at each step is applied to all the existing partitions. For example, the sequence of cuts x-x-y produces exactly the same result as x-y-x or y-x-x. A systematic way to produce such distinct bi-partitioning cuts is: Branch x has 3 children branches, namely x, y, z. Branch y is allowed to have two branches, y, z. Lastly, branch z can produce only one child branch, z. We can hence limit ourselves to non-replicated partition tree shown in Fig. 4(a).



(a) (b)

Fig. 4 partitioning tree – (a) Scene-independent non-replicated tree generation (b) Scene dependent searching paths (Example)

The total number of distinct leaves at the bottom of the non-replicated tree is $\frac{1}{2}(\log_2(n)+1)\cdot (\log_2(n)+2)$. This can be derived by using the formula, $n_x+n_y+n_z= \log_2(n)$, where n is the total number of ECME's and n_x, n_y, and n_z are the numbers of cuts in x-, y-, and z-directions, respectively. 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 on the right-hand side of the above equation. The number of distinct sets of $\{n_x, n_y, n_z\}$ is then computed as $\frac{1}{2}(\log_2(n)+1)\cdot(\log_2(n)+2)$. The number of leaves for the non-replicated tree increases logarithmically with the number of processing elements while that for the replicated tree grows exponentially. Therefore, our approach has resulted in a significant improvement. This approach does not depend on the characteristics of the scene, so we refer to it as *scene-independent.*

We can further reduce the number of paths searched in a tree by considering the worst case and the best case scenarios. Let us illustrate using a simple example (Fig. 4(b)). Suppose that a scene has 1000 vertices, and we have 8 ECME's. Assume that an x-directional cut at the top of tree divides the polygons into *(100, 900)*, a y-directional cut *(500,500)*, and a 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) produces *(250, 250, 230, 270)*, while a y-directional cut after the y-directional cut (i.e., y-y cut) results in *(200, 300, 100, 400)*. Since the maximum value in *(250, 250, 230, 270)* is 270 whereas that in *(200,300, 100, 400)* is 400, we choose z-directional cut as a next exploration. Here, we used the maximum value, and not the minimum value, since the computational time depends on the processor with the highest load. Then, in the 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. We can now determine that we don't need to traverse the unexplored x-directional cut and z-directional cut at the top of the tree. The reasoning 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., *(100, 900)* → *(100/2, 100/2, 900/2, 900/2)* → *(100/4, 100/4, 100/4, 100/4, 900/4, 900/4, 900/4, 900/4) = (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 traversal.

In general, we use the following inequality to reduce search space;

$1/2^{\text{remaining number of cuts}} \times$ *The max load value in the non-leaf node currently being explored > The max load value at an alternative non-leaf node*

The term on the left side represents the best case for the non-leaf node currently being explored. The best case scenario of the node is obtained by assuming that further partitioning could divide the number of vertices equally. The term on the right side represents the worst case of an alternative non-leaf node to be explored. If the above inequality holds, then, we don't need to traverse the non-leaf node that we are currently exploring, since the best case scenario for that node will give worse result than the worst case scenario of an alternative node. If the above inequality doesn't hold, we continue to search the path. A similar comparison can be also made between the current node and the best leaf-node identified so far. This reduction technique is *scene-dependent*, since it may or may not help reduce the search space depending on the scene's characteristics. By combining the scene-independent and the scene-dependent techniques we proposed, the optimal partitioning method can be found in a logarithmic time.

B. Placement Methodology

Suppose that we have identified the optimal partitioning in object space (GP) as well as image space (RAS) using the above reduction schemes. We then determine the placement of

the partitioned jobs (and associated polygons) in the memory modules for GP and RAS that minimizes the communication cost between them. Placements are critical for optimizing the communication cost, since the balancing as well as concurrency of inter module communications depends on the locations of source and destination nodes.

As mentioned earlier, we use sort-middle technique in our architecture. Given *n* ECME's at CIMM, the number of possible placements at the geometry processors is *n* factorial. Likewise, those at rasterization processor are n factorial. Therefore, in total, there are $(n!)^2$ possible placements. While this is an extremely large number, we can reduce the search space by discarding the equivalent placement pairs.

We present a concept called *branch alternation* that is defined as a swapping with respect to a branching point. For example, as shown in Fig. 5, data swapping between node 1 and node 2 causes no change in communication cost so long as both geometry and rasterization processors swap their data in their memory blocks (along with corresponding jobs). The same phenomenon occurs for the swapping of data in node 3 and node 4. For the same reason, swapping between the set {node 1, node 2} and the set {node 3, node 4} causes no change if the data at both geometry and rasterization processors are swapped concurrently. However, data swapping between node 1 and node 3 changes the communication cost, regardless of whether both GP and RAS are swapped or not.

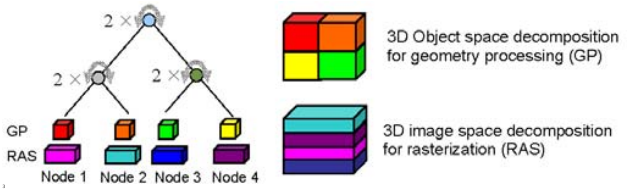


Fig. 5 Types of branch alternation for 4 ECME's

In the above example, 3 types of branch alternations are possible, which produce equivalent pairs. Therefore, $2\times2\times2=8$ pairs of placements will have exactly the same communication cost. We then reduce the search space from $(4!)^2$ to $(4!)^2/8$. In general, since the number of splitting branches in our PIM architecture with n leaf nodes is n-1, the total distinct placement pairs of geometry processor and rasterization processor will be $(n!)^2/2^{n-1}$.

We can now decompose $n!^2/2^{n-1}$ into two ways, namely, $n!\times n!/2^{n-1}$ or $n!/2^{n-1}\times n!$. By the former we mean that the placements for 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 the two are considered identical in rasterization processor, since we can match the two placements through a branch alternation. We have developed an approach to search for placements that takes full advantage of branch alternations and enumerates exactly $(n!)^2/2^{n-1}$ candidates. The details of the algorithm and the proof of its correctness can be found in our technical report [17].

Our algorithm for producing non-equivalent permutations significantly reduces the searching space, i.e., from $(n!)^2$ to $(n!)^2/2^{n-1}$. However, the reduced search space is still considerable. Therefore, we propose an approximate algorithm (heuristics) for generating the more promising pairs early in this search process. We will describe the approximate method and compare the solution identified by this approximate method with the optimal solution through exemplary simulation results.

C. Heuristics for Placement

Enumerating all possible placements is extremely time consuming. For example, suppose that we have 32 ECME's in CIMM. Then, the total number of possible non-equivalent placement pairs is 3.224×10^{61}. Therefore, we developed an approximate method called top-down heuristic for finding satisfactory placement pairs at low complexity, as an alternative to exhaustive search. Basically, we conduct swaps, starting from the highest level, all the way down to the lowest level according to the cost function (more ahead). The highest level swap is between blocks with the longest distance from each other. As an example, see Fig. 6. The distance between A_1 and A_7 is one of the longest ones in the decoder tree. The distance between A_4 and A_7 is also identical. The communication distance between A_3 and A_4 is one of the shortest. As excessive data communication at the highest level severely limits the parallelism of inter-block transfers and increases its delay, we try to minimize such communications by using our heuristics.

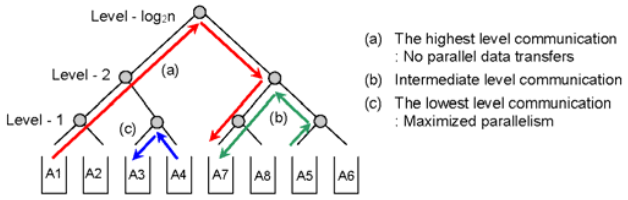


Fig. 6 Illustration of the different communication distances

When we apply this approximate method, the placement assignment of geometry processor is fixed, and the placement of rasterization processor is updated according to the cost function. Suppose that we want to calculate the cost function of level k with respect to the node i. First, we need to identify which nodes are at the distance of the level k from the perspective of the node i and which nodes are at the distance less than the level k from the point of the node i. Second, we compute the number of polygons to be transferred from the node i to the nodes that are at the distance of the level k. Likewise, compute the number of polygons to be moved from node i to the nodes that are at the distance less than the level k. Third, we compute the difference between the two quantities, which is defined as the cost function as follows;

$$Cost(k, i) = \sum_{x\in S_1} D_{x,i} - \sum_{y\in S_2} D_{y,i}$$

where, $D_{i,j}$ is defined as the number of polygons that are at node i (for GP) and then are transferred to node j (for RAS). Also, S_1

= { node IDs at the distance of the level k in the view of node i } and S_2 = { node IDs at the distance that is less than the level k in the view of node i }

The purpose of our cost function is to identify an element that has fewer communications with its current neighbors in decoder tree but has more communications with distant nodes. The first term on the right side at the above cost function sums up the communications with distant nodes, whereas the second term sums up the communications with current neighbors. Therefore, the more positive the cost function, the more inferior is the current assignment.

Our heuristics use this cost function to progressively update the placements of the blocks one by one, starting from the highest level down to the lowest level. For clarification, we see the case of 8 ECME's. We compute the cost values of the highest level for each element in the set {1, 2, 3, 4} and the set {5, 6, 7, 8}, respectively; as an instance, the cost value of node 1 with the highest level (level 3) is $D_{51}+D_{61}+D_{71}+D_{81}-D_{11}-D_{21}-D_{31}-D_{41}$ (see Fig. 7). Likewise, that of node 7 is $D_{17}+D_{27}+D_{37}+D_{47}-D_{57}-D_{67}-D_{77}-D_{87}$. Compute the cost values of all other 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 the two is positive, swap the two elements, since the current assignment is unstable. Next, pick the second largest one from {1, 2, 3, 4} and the second largest one from {5, 6, 7, 8}. If the sum of the two is positive, swap them. Continue until the sum becomes negative. Then, go down to the next lower level, which is level 2, where we compute the cost values of each element in {1,2} and {3,4}. We determine which node elements will be exchanged between the set {1,2} and the set {3,4}. Same procedure is applied for {5,6} and {7,8}. Once we arrive at the lowest level, we are finished. This top-down approach was evaluated empirically through example scenes and shown to produce good results (Section VI)

(b)

(a) GP \ RAS	Partition 1	Partition 2	Partition 3	···
Partition 1	D_{11}	D_{12}	···	
Partition 2	D_{21}	D_{22}	···	
Partition 3	D_{31}			
···				

Fig. 7 A table that shows data-transfers between GP and RAS partitions. Used for finding near-optimal placement through top-down approximation – Fixed placement for geometry processor (a) and progressively updated placement for rasterization processor (b)

VI. Experimental Results

We evaluated our entire methodology using the three example objects shown in Appendix A using a detailed c++ simulation of our PIM architecture. A PIM architecture with eight ECME's is studied. Since the scene is static, the preciseness of 3D distribution statistics along with their mapping relations between GP and RAS are not at issue (Fig. 3).

Table I shows the results of search space reduction ratio when the proposed scene independent and dependent techniques are applied. The scene independent technique reduces the search space in a manner that can be computed analytically, whereas the scene dependent bounding condition performs differently depending on the scene's characteristics. In the example cases, the overall search time was reduced by 71.2% for GP and 62.5% for RAS on average compared to the naïve searching scheme when the two reduction schemes were applied.

TABLE I
Search Space Reduction Ratio for Partitioning

	Stage	Object1	Object 2	Object 3
Total reduction ratio of search space	GP	79.5%	64.2%	69.3%
	RAS	56.5%	64.2%	66.7%
Reduction ratio by scene independent scheme	GP/RAS	51.2%	51.2%	51.2%
Reduction ratio by scene dependent bounding	GP	57.9%	26.3%	36.8%
	RAS	10.5%	26.3%	31.5%

Based on the search results, we conclude that the recommended partitionings for GP are z-z-z (object 1), z-z-z (object 2), and x-x-z (object 3). For RAS, x-x-x, x-x-y, and y-z-z are the recommended partitionings for the object 1, object 2, and object 3, respectively.

After determining the partitioning of both GP and RAS, mapping tables are constructed, representing the polygon mapping relations between z-z-z partitioning for GP and x-x-x partitioning for RAS (object 1, Table II(a)), z-z-z partitioning for GP and x-x-y partitioning for RAS (object 2, Table II(b)), and x-x-z partitioning for GP and y-z-z partitioning for RAS (object 3, Table II(c)). Based on the mapping information, we apply our top-down heuristics to find near-optimal placements for each partitioned scene. We also apply exhaustive search to identify globally optimal placements (Table III). The detailed procedures for obtaining the top-down placement are explained in our technical report [17].

The communication costs incurred by data rearrangements between GP and RAS for placements obtained by top-down heuristics are normalized by communication overheads for globally optimal placements obtained via exhaustive searches. Table III data shows such relative communication costs. This shows that our heuristics provide communication overheads that are within 7% of the optimal.

Partitioning methods that we have developed are limited to uniform partitioning, i.e., they do not consider arbitrary partitions. To ensure that this does not result in significant sub-optimality, we checked the performance of such constrained partitioning methods by using the load balance [31], which is defined as the ratio of the maximum processor's load over the average load. In [31], the author considers a load-balance reasonable if the maximum/average load ratio is 1.5 or less. The load-balances for the partitioning obtained by our approach of the three objects are in the reasonable range, as shown in Table IV, and comparable to the results of complicated adaptive load-balancing schemes presented in [31], which are much more costly and not suitable for real-time systems.

For your reference, we added plots on load-balances for each possible type of partitioning in Appendix. This illustrates the importance of determining appropriate partitioning types.

TABLE II
MAPPING RELATION FOR THE CHOSEN GP AND RAS PARTITIONING FOR EACH OBJECT

(a) Object 1

GP \ RAS	1	2	3	4	5	6	7	8
1	8	60	1	0	0	0	0	0
2	40	43	28	5	55	10	0	0
3	11	14	37	12	103	73	0	0
4	20	6	25	20	1490	152	5	9
5	20	7	35	20	38	141	6	6
6	12	28	64	19	85	39	0	0
7	61	22	20	21	55	0	0	0
8	64	0	0	4	1	0	0	0

(b) Object 2

GP \ RAS	1	2	3	4	5	6	7	8
1	17	144	31	0	0	0	0	0
2	53	319	122	81	240	1	0	0
3	17	38	190	262	143	0	126	0
4	51	19	148	315	12	128	232	0
5	127	32	192	358	34	152	203	23
6	78	21	120	517	61	101	256	0
7	14	12	6	468	20	0	313	1
8	8	37	0	235	18	0	276	0

(c) Object 3

GP \ RAS	1	2	3	4	5	6	7	8
1	336	80	0	0	0	0	0	0
2	203	0	6	1	125	83	0	0
3	0	0	147	0	72	0	105	0
4	176	0	8	0	138	77	0	0
5	0	0	122	0	61	0	168	0
6	13	0	0	0	405	0	0	0
7	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0

*The row and column numbers (1~8) are IDs of blocks that are generated by partitioning, not the actual placement in the CIMM.

TABLE III
PLACEMENTS RESULTS OF TOP-DOWN HEURISTIC AND EXHAUSTIVE SEARCH

			Top-down heuristic								Optimal placement								†† Relative communication cost
ECME node ID			1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8	
Placements	† GP (fixed)		1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8	
	RAS	Object 1	2	8	6	5	4	3	7	1	8	2	6	7	4	5	1	3	1.06
		Object 2	2	5	3	8	6	1	4	7	7	2	3	6	4	5	8	1	1.07
		Object 3	1	2	3	6	7	5	4	8	1	2	3	7	6	4	5	8	1.01

† The placements in GP are fixed, while those in RAS are varied.
†† Relative communication cost is defined as the ratio between the communication cost for globally optimal placement and that for top-down heuristics.

TABLE IV
PERFORMANCE DATA FOR THE CHOSEN PARTITIONING METHODS

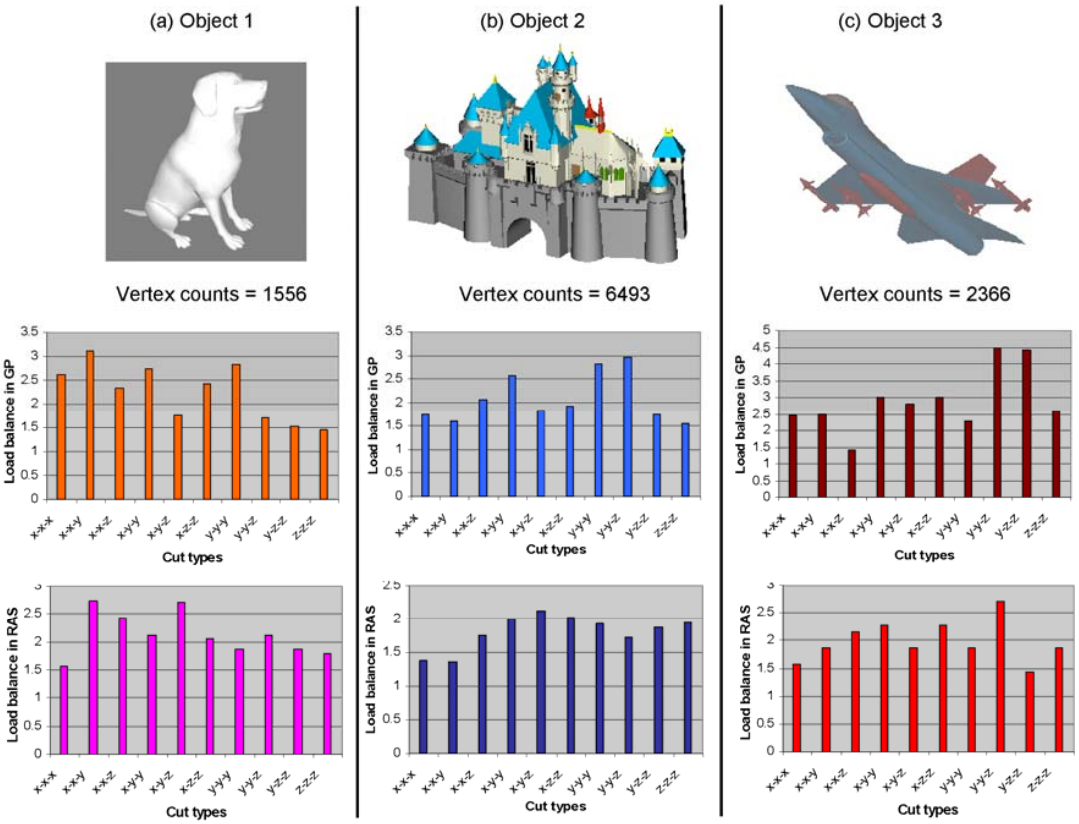
		object 1	object 2	object 3
Load-Balance	GP	1.45	1.56	1.41
	RAS	1.56	1.35	1.42

VII. SUMMARY AND FUTURE WORKS

In this paper, we proposed an efficient PIM architecture intended for computer graphics and explored methods for efficient partitioning and placement under the uniform partitioning constraints. Sophisticated 3D graphics requires intensive memory accesses and many current architectures suffer processor-memory bottlenecks. Therefore, PIM architecture that reduces memory access latency can alleviate such bottlenecks and thus can be an ideal candidate for high quality computer graphics. We used a hybrid-partitioning method and proposed search space reduction algorithms – one scene-dependent and one scene-independent. Scene independent reduction scheme reduces computational complexity in an analytically quantifiable manner, while scene dependent bounding condition reduces search space depending on the characteristics of the scene. From the simulation results for example objects, the average of the reduction ratio was 71.2% for GP and 62.5% for RAS when the two schemes were applied. As for placement, we reduced the search space by using our branch alternation approach, which reduces search space by exponential number. However, since the search space for placement is intrinsically large, we also developed a top-down heuristic to identify near-optimal placements efficiently. According to the simulation results, our top-down heuristic performed close-to-optimal – the ratio of communication-cost between our heuristic and the optimal placement was less than 1.07. We also performed simulations on the load-balance among processors under uniform partitioning, and it showed reasonable performances within or close to 1.5.

We are currently developing realistic analytical models for general types of graphics architectures and PIM architecture for two distinct cases, namely, unified processors and non-unified processors. By doing so, we will be able to identify which parts in the given architecture are bottlenecks, and resolve the problems appropriately. Also, for the given hardware constraints, we will be able to predict and compare the performances of various architectures using higher level models.

APPENDIX



REFERENCES

[1] International Technology Roadmap for Semiconductors , www.itrs.net/
[2] Keith Diefendorff, et al., How Multimedia Workloads Will Change Processor Design, IEEE Computer, p.43-45, 1997.
[3] D. Burger, et al., *Memory Bandwidth Limitations of Future Microprocessors*, In Proceedings of the 23rd Inter-national Symposium on Computer Architecture, p.78–89, 1996.
[4] Patterson D, et al., *A Case for Intelligent DRAM: IRAM*, IEEE Micro, 1997.
[5] Mark Oskin, et al., *Active Pages: A Computation Model for Intelligent Memory*, In Proceedings of the 23rd. Inter-national Symposium on. Computer Architecture, p.192-203, 1998.
[6] Yi Kang, et al., *FlexRAM: Toward an Advanced Intelligent Memory System*, In proceedings of 1999 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 proceedings of the 14th IEEE International Conference on Application-Specific Systems, Architectures, and Processors, p.282-292, 2003.
[8] Jung-Yup Kang, et al., *Accelerating the Kernels of BLAST with an Efficient PIM (Processor-In-Memory) Architecture*, In proceedings of the 3rd International IEEE Computer Society Computational Systems Bioinformatics Conference, p.552-553, 2004.
[9] John Montrym, et al., *The GeForce 6800*, IEEE Micro**,** p.41-51, 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, p.23-32, 1994.
[12] S. Whitman, *Dynamic load balancing for parallel polygon rendering*, IEEE Computer Graphics and Applications, p.41-48, 1994.
[13] S. Whitman, *Parallel Graphics Rendering Algorithms*, In Proceedings of 3rd Eurographics Workshop on Rendering, Consolidation Express, Bristol, UK, p.123-134, 1992.
[14] Tahsin M. Kurc, et al., Object-Space Parallel Polygon Rendering on Hypercubes, Compu-ters & Graphics , p.487-503, 1998.
[15] B. Wei, et al., *Performance Issues of a Distributed Frame Buffer on a Multicomputer*. In Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics Hardware, p.87 -96, 1998.
[16] Vineet Kumar. *A Host Interface Architecture for HIPPI*. In Proceedings of Scalable High Performance Computing Conference, p.142–149, 1994.
[17] Jae C. Cha, et al., Technical Report CENG-2007-6.
[18] Akeley, Kurt. *RealityEngine Graphics.* In Proceedings of SIGGRAPH '93, New York, p.109-116, 1993.
[19] Thomas W. Crockett, et al., *Rendering Algorithm for MIMD Architectures*, In Proceedings of the 1993 Parallel Rendering Symposium, p.35-42,1993.
[20] Deering, et al., *A System for Cost Effective 3D Shaded Graphics*. In Proceedings of SIGGRAPH '93, p.101-108, 1993.
[21] Ellsworth, et al.,. *A New Algorithm for Interactive Graphics on Multicomputers*. *IEEE Computer Graphics & Applications*, p.33-40, 1994.
[22] Fuchs, Henry, et al., *Pixel-Planes 5: A Heterogeneous Multiprocessor Graphics System Using Processor-Enhanced Memories*. In Proceedings of SIGGRAPH '89, p.79-88, 1993.
[23] J. D. Foley, et al., *Computer Graphics, Principles and Practice*. Addison-Wesley, 2nd edition, 1996.

[24] Francis S Hill Jr., et al., *Computer Graphics Using OpenGL*, Prentice Hall, 3rd edition, 2006.
[25] Tomas Akenine-Moller, et al., *Real-Time Rendering*, 2nd edition, A.K. Peters Ltd, 2002.
[26] Thomas W. Crockett, *An Introduction to Parallel Rendering*, Parallel Computing, p.819-843, 1997.
[27] D.R. Roble, *A Load Balanced Parallel Scanline Z-Buffer Algorithm for the iPSC Hypercube*, In Proceedings of the 1st International Conference PIXIM 88, p.177-192, 1998.
[28] D.S. Whelan, *Animac: A Multiprocessor Architecture for Real time Computer Animation*, Ph.D. dissertation, California Institute of Technology, Pasadena, CA, 1985.
[29] Carl Mueller, *Hierarchical Graphics Databases in Sort-First*, In Proceedings of the IEEE Symposium on Parallel Rendering, p.49-57, 1997.
[30] David Ellsworth, *A Multicomputer Polygon Rendering Algorithm for Interactive Applications*, In Proceedings of the 1993 Parallel Rendering Symposium, p.43-48, 1993.
[31] Carl Mueller, *The sort-first rendering architecture for high-performance graphics*, In Proceedings of the 1995 symposium on Interactive 3D graphics, p.75-ff., Monterey, 1995.
[32] The Cg Tutorial: *The Definitive Guide to Programmable Real-Time Graphics*, NVDIA, http://developer.nvidia.com/CgTutorial.
[33] Dirk Bartz, *Rendering and Visualization in Parallel Environments*, In SIGGRAPH 2000 Course.
[34] Frederico Abraham et al., *A Load-Balancing Strategy for Sort-First Distributed Rendering*, In Proceedings of SIGGRAPH '04, p.292-299, 2004.
[35] Wulf, Wm.A and McKee, S.A. *Hitting the Memory Wall: Implications of the Obvious.* ACM Computer Architecture News. Vol.23, No.1, 1995.
[36] http://www.nvidia.com/page/8800_tech_specs.html
[37] http://www.xbox.com/en-AU/support/xbox360/manuals/xbox360specs.htm
[38] http://techreport.com/articles.x/10039/1