Chapter 10

Multi-Level Adaptive Cross Approximation

In Chapter 9, basis functions were clustered into spatially localized groups of roughly equal size using the K-means algorithm, thereby imposing a block structure on the MoM matrix. It was then shown that the off-diagonal blocks in the system matrix, as well as the corresponding off-diagonal **LU** matrix blocks, can be compressed via the ACA+QR/SVD technique (Section 9.2.2). Because K-means is not a hierarchical algorithm, it creates a single set of basis-function groups. We therefore refer to this approach as the single-level ACA, or simply the ACA.

It was pointed out in [1] that the number of degrees of freedom (DoF) remains asymptotically constant when the sizes of the source and testing groups are varied inversely. This concept was used to further extend the ACA in [2, 3] through a recursive subdivision of the basis functions in each group. This approach splits blocks into column sub-blocks, which are compressed separately via ACA. The resulting **U** matrices, which contain the singular values and left-singular vectors of the compressed sub-blocks, are stacked and recursively compressed via ACA. This multi-level approach, called the Multi-Level ACA (MLACA), was shown to greatly improve the overall compression ratio at the expense of increased computation time. In that paper, however, only the system matrix was compressed, and an iterative solver was used for the solution. We then showed in [4] that the MLACA is also capable of compressing the **LU** matrix blocks very efficiently, making a direct solution feasible.

In this Chapter, we present a further refinement of our algorithm in [4], which fuses elements from the ACA in Chapter 9 with the MLACA as presented in [2]. The geometry is partitioned using semi-hierarchical K-means clustering, and both the MoM system matrix **Z** and the **LU** matrix are compressed. Unlike in the ACA, the off-diagonal *and* diagonal blocks are compressed in the MLACA, yielding a much greater level of compression.

Many concepts and algorithms from the single-level ACA in Chapter 9, such as the block-LU factorization and block-RHS solution algorithms, apply to the MLACA with almost no modification. Other implementation details, such as software changes and

additional classes needed for MLACA support, will be discussed in detail.

10.1 MLACA Compression of Matrix Blocks

In this section, we discuss hierarchical clustering of sub-groups and the MLACA compression algorithm. We will then apply it to the direct solution of the MoM matrix system via LU factorization in Section 10.2.

10.1.1 MLACA Fundamentals

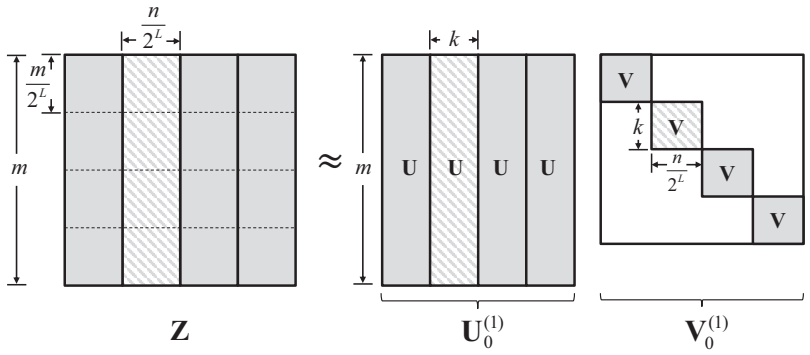
MLACA is based on the idea that the number of degrees of freedom (DoF) remains asymptotically constant when the sizes of well-separated source and testing groups are inversely varied. Thus, an L-level MLACA further divides the basis functions in each single-level ACA group into 2^L spatially localized sub-groups. Each off-diagonal block $\mathbf{Z}^{m\times n}$, with m rows and n columns, now comprises 2^L row and column sub-blocks, each having approximately $\frac{m}{2^L}$ and $\frac{n}{2^L}$ rows and columns. To begin the MLACA compression, the ACA is first applied to each block column of $\mathbf{Z}$ independently. The $\mathbf{U}$ matrices of the compressed block columns, which contain the singular values and left-singular vectors, are then joined in pairs column-wise and divided in half row-wise. The ACA compression process is then repeated for the upper and lower halves, recursively, until only a single block row and column remains.

To illustrate, consider an $L = 2$ level MLACA, where operations on the first level $l = 0$ are depicted in Figure 10.1a. Each block column of $\mathbf{Z}$ is compressed separately using ACA+QR/SVD, yielding four $\mathbf{U}$ and $\mathbf{V}$ block matrix pairs. The singular values from the recompression step are retained in each $\mathbf{U}$ block. The $\mathbf{U}$ blocks, each comprising approximately m rows and k columns, where $k << n$, are stacked together to create the notional block matrix $\mathbf{U}_0^{(1)}$. The smaller $\mathbf{V}$ blocks, forming the notional block-diagonal matrix $\mathbf{V}_0^{(1)}$, each comprise approximately k rows and $\frac{n}{2^L}$ columns. These $\mathbf{V}$ blocks are fully compressed and stored. We then proceed as in Figure 10.1(b), where we now combine pairwise the $\mathbf{U}$ column blocks and divide the result in half row-wise, yielding four smaller blocks, each having approximately $\frac{m}{2}$ rows and $2k$ columns. We then recurse to level $l = 1$, where the block columns of each half, each having approximately $\frac{m}{2^l}$ rows and $2k$ columns, are compressed in the same way as on the previous level. This recursion is repeated until the finest level is reached, where there is only a single block row and column. After applying the ACA on this level, the $\mathbf{U}$ and $\mathbf{V}$ matrix pair are stored and the MLACA compression of $\mathbf{Z}$ is complete.

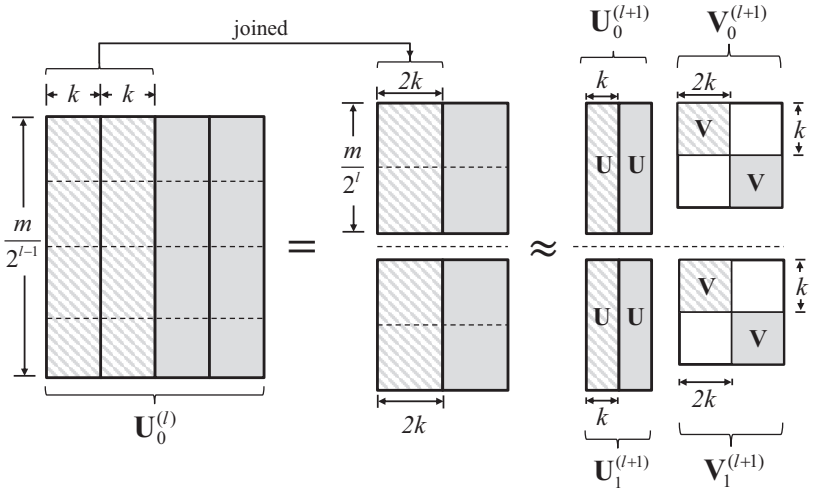
Because this algorithm stacks and recursively compresses the $\mathbf{U}$ blocks, we refer to it as a U-type MLACA. An error analysis of MLACA was presented by other authors in [5]. Finally, we note that for $L = 0$, MLACA is identical to the single-level ACA.

10.1.1.1 SVD-Based Compression on Higher Levels

Section 9.2.1.3 presented a failure case for the ACA in which the source and testing groups comprised coplanar groups of facets on a cube face. This problem was solved by arranging all electric basis functions in a group first, followed by any magnetic basis



(a) Operations on Level $l = 0$



(b) Operations on Level $l + 1$

Figure 10.1: 2-Level U-Type MLACA

functions. Hints were then passed to the ACA driver, forcing the ACA to search the upper and lower halves of the matrix.

This potential failure case remains when using the MLACA and must therefore be addressed. When performing compressions on level $l = 0$, it can be avoided in the same manner as in the single-level ACA, by ordering the electric basis functions in each group block sub-column first, followed by magnetic functions. This is done during initialization when clustering the sub-groups, and the hints provided to the ACA driver differ for each column block. On levels $l > 0$, this problem persists, with zero sub-blocks appearing in the upper and lower halves of the $\mathbf{U}_0^{(l)}$ matrix. In this case, however, there is no way to predict *a priori* where these locations will appear, only that it may happen, so it is impossible to provide any hints to the ACA driver.

Fortunately, knowing which sub-blocks can exhibit this behavior allows us to han-

dle all possible cases. If either (or both) of the top-level source and testing sub-groups have only electric basis functions, we know it will not happen and we use the ACA+QR/SVD for compression at all levels. However, when both sub-groups have magnetic basis functions, we know the failure is possible. Thus, we use ACA+QR/SVD on level $l = 0$, as hints are available for the ACA driver. On levels $l > 0$, we instead apply the SVD directly. This approach would be inefficient on level $l = 0$ if applied to the column blocks of $\mathbf{Z}^{m\times n}$, each approximately of size $m \times \frac{n}{2^L}$. However, the blocks on levels $l > 0$ are much smaller, being approximately of size $\frac{m}{2^l} \times 2k$. Thus, the use of SVD on the higher levels does not incur unreasonable overhead, and in practice compression times were found to be higher by only 5 percent, or less, when applying the SVD directly versus ACA+QR/SVD.

10.1.2 Hierarchical Clustering of Sub-Groups

In MLACA, block columns are paired together before recursing to a higher level. Therefore, the basis functions in each pair must be clustered so that their union is itself spatially local. Similarly, when the joined blocks are split in half row-wise, the upper and lower halves should each be spatially local. Thus, we see a clear parent-child relationship between the block rows and columns as the algorithm moves between levels. This suggests that a binary tree-based subdivision of groups, where sub-groups on each level are divided in half spatially, is needed for the MLACA.

We must consider carefully how this subdivision is performed. In [2] a tree-based subdivision of the basis functions in angle space was used, based on the solid angle of a source group as seen by a testing group, and vice versa. It was said that this solid-angle approach is critical for the MLACA to obtain its gains in computational complexity and memory savings. However, this approach results in a local re-ordering of the rows and columns in each block that are necessarily different for other blocks. When using an iterative solver, this does not present any issues, as only elements from the input and output vectors are shuffled during the matrix-vector product with a block. However, this will cause difficulties in performing steps 2, 5, and 7 in Algorithm 7, as $\mathbf{Z}$, $\mathbf{L}$, and $\mathbf{U}$ will need to be unshuffled prior to each matrix product so that the global ordering is followed, and the results shuffled again prior to recompression. This would be very inefficient.

Our approach is to apply the same K-means algorithm used to create the single-level ACA groups, but in a recursive way. We first generate the single-level groups, as in the ACA. For a problem with N basis functions, this results in $M = N/N_g$ single-level groups, where N_g is the target number of basis functions per group. For each group, we then apply K-means to create two spatially local sub-groups, and then recursively divide these in half via K-means, and so on, to a maximum level L. This produces a per-group reordering, so the global ordering can be adjusted to match and no later shuffling is required. This strategy was compared to the tree-based solid angle method when compressing off-diagonal blocks for a variety of test geometries. It was observed that the size of the compressed blocks differed by less than 2 percent in each case, validating the approach.

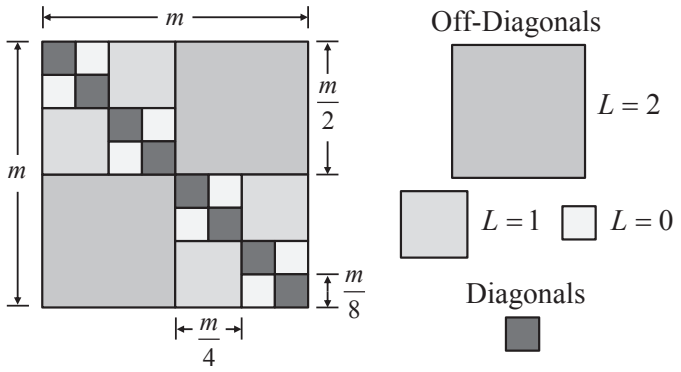


Figure 10.2: MLACA Compression of Diagonal Block For $L = 3$

10.1.3 Compression of Diagonal Blocks

Another advantage of the hierarchical clustering of sub-groups is that it permits compression of the diagonal blocks. Consider the example diagonal block in Figure 10.2, which has m rows and m columns, and is compressed using an $L = 3$ level MLACA. The compression comprises a recursive walk of the binary tree starting at the root ($l = 0$). On levels $l > 0$, the off-diagonal sub-blocks are compressed using an $(L - l)$ level MLACA, and recursion to a higher level occurs only for the diagonal sub-blocks. On the finest level, the diagonal sub-blocks, now each having roughly $\frac{m}{2^L}$ rows and columns, are stored in full form and the compression is complete.

10.2 Direct Solution of MLACA-Compressed Matrix System

The LU factorization of the MLACA-compressed MoM system matrix follows Algorithm 7, with a few modifications, as summarized in Algorithm 12. Each diagonal block is first expanded to full form prior to its update in step 2 and later re-compressed in step 5, and the full version of $[\mathbf{LU}]_{bb}$ from step 4 is retained for the triangular back-substitution in steps 7 and 9. We now discuss the reconstruction of MLACA-compressed matrix blocks, and their products, as these must be optimized to yield a fast and efficient algorithm.

10.2.1 MLACA Block Reconstruction

Prior to performing the block updates in steps 3, 7 and 9 of Algorithm 12, we must first expand $\mathbf{Z}_{bb}$, $\mathbf{Z}_{sb}$, and $\mathbf{Z}_{bs}$ to full size. For MLACA-compressed blocks this is done via recursion, where nodes on level l reconstruct their portion of the matrix $\mathbf{U}_{0,1}^{l-1}$ on the previous level. On level $l = 0$ (the coarsest level), once $\mathbf{U}_0^{(1)}$ is available, the full

Algorithm 12 MLACA-Compressed Block LU Factorization

for $b = 1$ to M **do**
  Reconstruct $\mathbf{Z}_{bb}$
  $\mathbf{A} = \mathbf{Z}_{bb} - \sum_{s=1}^{b-1} [\mathbf{L}_u \mathbf{L}_v]_{bs} [\mathbf{U}_u \mathbf{U}_v]_{sb}$
  $[\mathbf{LU}]_{bb} = \mathrm{LU}(\mathbf{A})$ (scalar LU factorization)
  Compress and store $[\mathbf{LU}]_{bb}$ via MLACA, retain full $[\mathbf{LU}]_{bb}$
  for $s = b + 1$ to M **do**
    $\mathbf{L}_{sb} = \left[[\mathbf{Z}_u \mathbf{Z}_v]_{sb} - \sum_{p=1}^{b-1} [\mathbf{L}_u \mathbf{L}_v]_{sp} [\mathbf{U}_u \mathbf{U}_v]_{pb} \right] \mathbf{U}_{bb}^{-1}$
    Compress and store $\mathbf{L}_{sb}$ via MLACA
    $\mathbf{U}_{bs} = \mathbf{L}_{bb}^{-1} \left[[\mathbf{Z}_u \mathbf{Z}_v]_{bs} - \sum_{p=1}^{b-1} [\mathbf{L}_u \mathbf{L}_v]_{bp} [\mathbf{U}_u \mathbf{U}_v]_{ps} \right]$
    Compress and store $\mathbf{U}_{bs}$ via MLACA
  end for
end for

matrix is then obtained via the product

$$\mathbf{Z} = \mathbf{U}_0^{(1)} \mathbf{V}_0^{(1)}. \tag{10.1}$$

Operations on each level are fast because each reconstruction step consists of matrix products in rank-reduced form.

10.2.2 Matrix Product and V-Type MLACA

Matrix products in the ACA block updates, discussed in Section 9.4.1 are very fast as they operate in rank-reduced form, requiring only a few calls to the Level 3 BLAS. However, now consider the product $\mathbf{AB}$ of two U-type MLACA matrices on level $l = 0$, as shown in Figure 10.3. Prior to performing the product, the compressed $\mathbf{U}$-column blocks of each matrix have been reconstructed. It is clear that the innermost product $\mathbf{A}_v \mathbf{B}_u$ is not efficient, as each small diagonal block of $\mathbf{A}_v$ multiplies only a small portion of the block columns of $\mathbf{B}_u$. For the $L = 2$ level MLACA depicted, this product requires $2^L = 4$ `cgemm` calls.

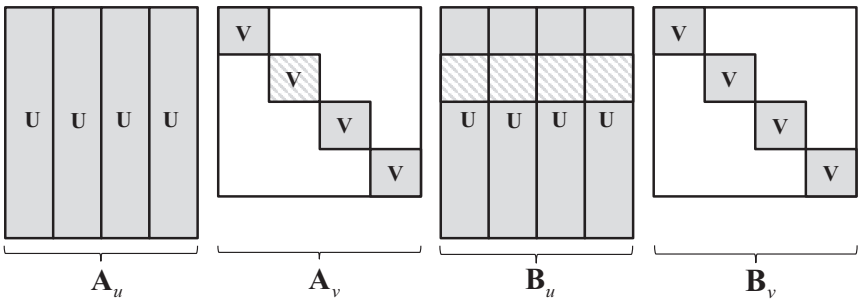


Figure 10.3: Product of U-Type MLACA Matrices

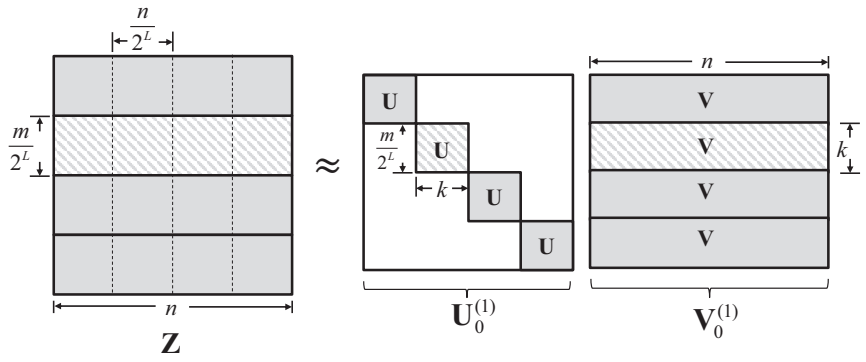


Figure 10.4: V-Type MLACA

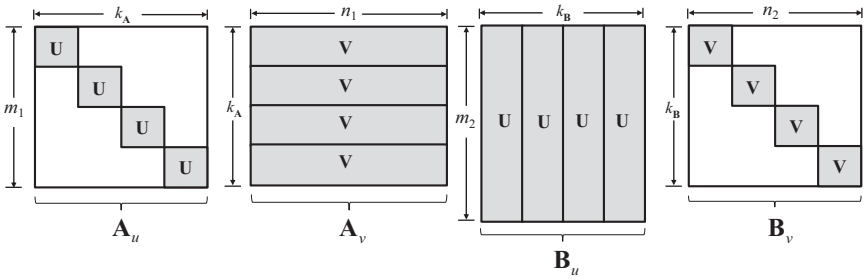


Figure 10.5: Product of V and U-Type Matrices

This innermost product can be reduced to a single operation if we modify the MLACA algorithm used to compress $\mathbf{A}$. Consider again the level $l = 0$ operations depicted in Figure 10.1(a). If we instead compress the block rows of $\mathbf{Z}$, we now have the configuration in Figure 10.4. The smaller diagonal $\mathbf{U}$ blocks of $\mathbf{U}_0^{(1)}$ are stored, and the larger $\mathbf{V}$ blocks comprising $\mathbf{V}_0^{(1)}$ are paired row-wise and split in half column-wise. These left and right halves are then recursively compressed in a manner analogous to the U-type MLACA, resulting in a variant we call the V-type MLACA.

The resulting product of V and U-type MLACA matrices is depicted in Figure 10.5, where the innermost product has been reduced to a single operation. We note that we can force all MLACA matrix products to have this form by compressing all blocks below the diagonal using V-type MLACA, and all blocks above the diagonal using U-type MLACA.

In [4], the innermost product was performed by first fully reconstructing $\mathbf{A}_v$ and $\mathbf{B}_u$, prior to computing any intermediate products. This approach is summarized in Section 10.2.2.1. Since the publication of [4], a more efficient method was identified, wherein the first intermediate product is computed directly without needing to compute $\mathbf{A}_v$ or $\mathbf{B}_u$. This approach is presented in Section 10.2.2.2.

10.2.2.1 Top-Level Matrix Product

The "top-level" matrix product begins by reconstructing the compressed row blocks of $\mathbf{A}_v$ and the compressed column blocks of $\mathbf{B}_u$, using the recursive approach in Section 10.2.1. The innermost product in Figure 10.5 then comprises a single operation, yielding

$$\mathbf{T}_1^{k_\mathbf{A} \times k_\mathbf{B}} = \mathbf{A}_v^{k_\mathbf{A} \times n_1} \mathbf{B}_u^{m_2 \times k_\mathbf{B}}, \tag{10.2}$$

where $\mathbf{T}_1^{k_\mathbf{A} \times k_\mathbf{B}}$ is a $2^L \times 2^L$ block matrix, and $k_\mathbf{A}$ and $k_\mathbf{B}$ are the sum of the ranks of all the compressed sub-blocks in $\mathbf{A}$ and $\mathbf{B}$, respectively. As in Section 9.4.1, whether the left or right product is performed first depends on $\min(k_\mathbf{A}, k_\mathbf{B})$. If $k_\mathbf{A} < k_\mathbf{B}$, we compute $\mathbf{T}_2 = \mathbf{T}_1 \mathbf{B}_v$ yielding

$$\mathbf{T}_2^{k_\mathbf{A} \times n_2} = \mathbf{T}_1^{k_\mathbf{A} \times k_\mathbf{B}} \mathbf{B}_v^{k_\mathbf{B} \times n_2}, \tag{10.3}$$

and then

$$[\mathbf{LU}]^{m_1 \times n_2} = \mathbf{A}_u^{m_1 \times k_\mathbf{A}} \mathbf{T}_2^{k_\mathbf{A} \times n_2}. \tag{10.4}$$

Otherwise, we compute $\mathbf{T}_3 = \mathbf{A}_u \mathbf{T}_1$, resulting in

$$\mathbf{T}_3^{m_1 \times k_\mathbf{B}} = \mathbf{A}_u^{m_1 \times k_\mathbf{A}} \mathbf{T}_1^{k_\mathbf{A} \times k_\mathbf{B}}, \tag{10.5}$$

and then

$$[\mathbf{LU}]^{m_1 \times n_2} = \mathbf{T}_3^{m_1 \times k_\mathbf{B}} \mathbf{B}_v^{k_\mathbf{B} \times n_2}. \tag{10.6}$$

As $\mathbf{A}_u$ and $\mathbf{B}_v$ each comprise block diagonal matrices of 2^L blocks, the product $\mathbf{AB}$ is thus reduced to $2^{L+1} + 1$ matrix products.

10.2.2.2 Bottom-Level Matrix Product

The "top-level" approach outlined in Section 10.2.2.1 is reasonably efficient, however it becomes less attractive as the size of the groups is increased (see Section 10.4.2.1). This is because as n_1 and m_2 grow, the workspace needed to store $\mathbf{A}_v^{k_\mathbf{A} \times n_1}$ and $\mathbf{B}_u^{m_2 \times k_\mathbf{B}}$ becomes excessive. This is especially problematic when working on the GPU, as each host thread must have its own workspace for $\mathbf{A}_v$ and $\mathbf{B}_u$ as well as the other intermediate products. In particular, $\mathbf{T}_2^{k_\mathbf{A} \times n_2}$ and $\mathbf{T}_3^{m_1 \times k_\mathbf{B}}$ are of similar size, further increasing the GPU memory burden. Finally, the approach from Section 9.6.2.2, where all $\mathbf{L}_v$ left of the diagonal and $\mathbf{U}_u$ above the diagonal are pre-staged on the GPU, quickly depletes GPU memory and limits the number of threads that can work on the device.

However, we note that $\mathbf{A}_v$ and $\mathbf{B}_u$ comprise temporary matrices which are discarded after computing the intermediate product $\mathbf{T}_1$ in (10.2). Similarly, when reconstructing $\mathbf{A}_v$ and $\mathbf{B}_u$, at higher levels of recursion the matrices $\mathbf{U}_{0,1}^l$ (U-type) and $\mathbf{V}_{0,1}^l$ (V-type) are also temporary and discarded after reconstructing their portion of $\mathbf{U}_{0,1}^{l-1}$ or $\mathbf{V}_{0,1}^{l-1}$ on the next higher level. Only at the finest level L are the matrices $\mathbf{U}_{0,1}^L$ and $\mathbf{V}_{0,1}^L$ are actually stored.

Therefore, we can instead construct the product $\mathbf{T}_1$ directly by walking the tree structure of $\mathbf{A}$ and $\mathbf{B}$ simultaneously, from bottom to top, starting on the finest level. To illustrate the order of operations, we will first work from the top level down to the

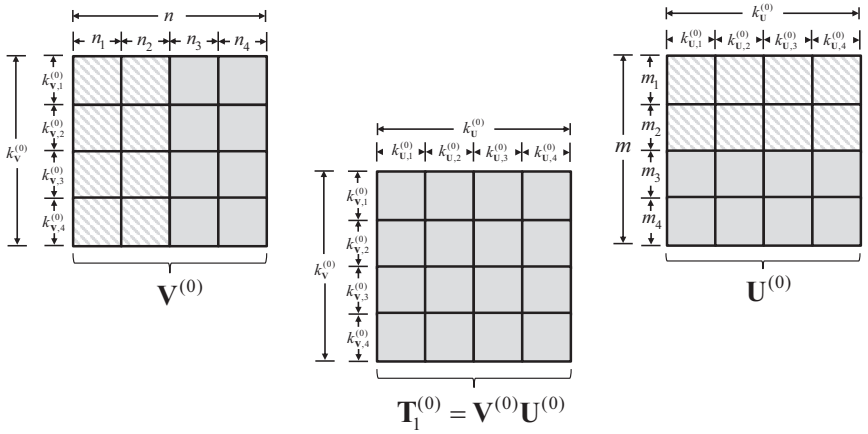


Figure 10.6: Reconstruction of $\mathbf{T}_1^{(0)}$

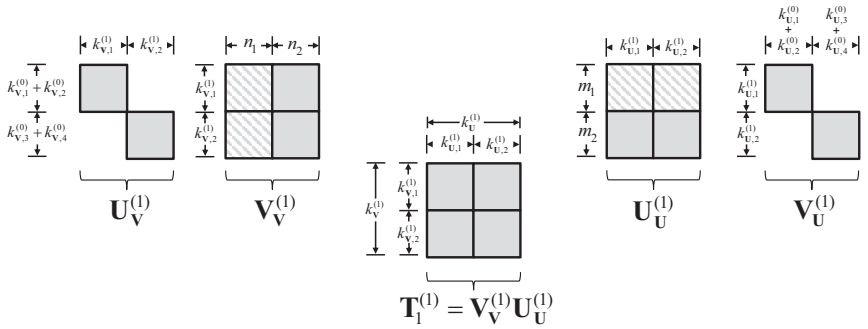


Figure 10.7: Reconstruction of $\mathbf{T}_1^{(1)}$

finest level. Let us define $\mathbf{V}^{(0)} = \mathbf{A}_v$ and $\mathbf{U}^{(0)} = \mathbf{B}_u$, and as depicted in Figure 10.6, the product $\mathbf{T}_1^{(0)} = \mathbf{V}^{(0)}\mathbf{U}^{(0)}$ on level $l = 0$. Let us now consider the compression of the left half of $\mathbf{V}^{(0)}$ and upper half of $\mathbf{U}^{(0)}$, as depicted. This corresponds to following only one side of the binary tree. The result on level $l = 1$ is shown in Figure 10.7, where $\mathbf{U}_{\mathbf{V}}^1$ and $\mathbf{V}_{\mathbf{U}}^1$ are stored, $\mathbf{V}_{\mathbf{V}}^{(1)}$ and $\mathbf{U}_{\mathbf{U}}^{(1)}$ are temporary matrices to be compressed, and $\mathbf{T}_1^{(1)} = \mathbf{V}_{\mathbf{V}}^{(1)}\mathbf{U}_{\mathbf{U}}^{(1)}$. Recursing in the same way again yields the result on level $l = 2$, shown in Figure 10.8. In this case, $\mathbf{U}_{\mathbf{V}}^{(2)}$, $\mathbf{V}_{\mathbf{V}}^{(2)}$, $\mathbf{U}_{\mathbf{U}}^{(2)}$, and $\mathbf{V}_{\mathbf{U}}^{(2)}$ are all stored and the recursion is finished. Note that at each step in the recursion, there is a second path which compresses the right half of $\mathbf{V}_{\mathbf{V}}^l$ and the bottom half of $\mathbf{U}_{\mathbf{U}}^l$, which would be depicted similarly.

Having reached the bottom of the tree, we now work our way back to the top. Starting on level $l = 2$, we first compute

$$\mathbf{T}_1^{(2),k_{\mathbf{V},1}^{(2)} \times k_{\mathbf{U},1}^{(2)}} = \mathbf{V}_{\mathbf{V}}^{(2),k_{\mathbf{V},1}^{(2)} \times n_1} \mathbf{U}_{\mathbf{U}}^{(2),m_1 \times k_{\mathbf{U},1}^{(2)}}, \tag{10.7}$$

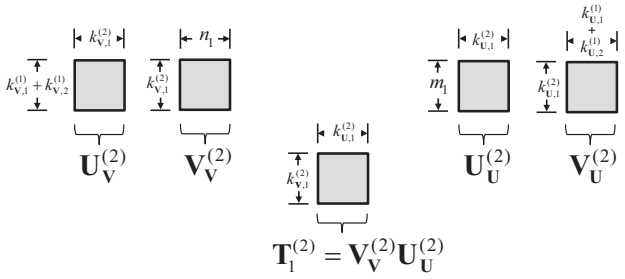


Figure 10.8: Reconstruction of $\mathbf{T}_1^{(2)}$

and then

$$\mathbf{T}_2^{(2),k_{\mathbf{V}}^{(1)}\times k_{\mathbf{U},1}^{(2)}} = \mathbf{U}_{\mathbf{V}}^{(2),k_{\mathbf{V}}^{(1)}\times k_{\mathbf{V},1}^{(2)}}\mathbf{T}_1^{(2),k_{\mathbf{V},1}^{(2)}\times k_{\mathbf{U},1}^{(2)}}, \tag{10.8}$$

where $k_{\mathbf{V}}^{(1)} = k_{\mathbf{V},1}^{(1)} + k_{\mathbf{V},2}^{(1)}$, and finally

$$\mathbf{T}_1^{(1),k_{\mathbf{V}}^{(1)}\times k_{\mathbf{U}}^{(1)}} = \mathbf{T}_2^{(2),k_{\mathbf{V}}^{(1)}\times k_{\mathbf{U},1}^{(2)}}\mathbf{V}_{\mathbf{U}}^{(2),k_{\mathbf{U},1}^{(2)}\times k_{\mathbf{U}}^{(1)}}, \tag{10.9}$$

where $k_{\mathbf{U}}^{(1)} = k_{\mathbf{U},1}^{(1)} + k_{\mathbf{U},2}^{(1)}$. We note that there is an equally sized contribution to (10.9) obtained by following the other recursive path. Moving now to level $l = 1$, and having already computed the intermediate product $\mathbf{T}_1^{(1)}$, we can immediately compute $\mathbf{T}_2^{(1)}$, which is

$$\mathbf{T}_2^{(1),k_{\mathbf{V}}^{(0)}\times k_{\mathbf{U}}^{(1)}} = \mathbf{U}_{\mathbf{V}}^{(1),k_{\mathbf{V}}^{(0)}\times k_{\mathbf{V}}^{(1)}}\mathbf{T}_1^{(1),k_{\mathbf{V}}^{(1)}\times k_{\mathbf{U}}^{(1)}}, \tag{10.10}$$

and finally

$$\mathbf{T}_1^{(0),k_{\mathbf{V}}^{(0)}\times k_{\mathbf{U}}^{(0)}} = \mathbf{T}_1^{k_{\mathbf{A}}\times k_{\mathbf{B}}} = \mathbf{T}_2^{(1),k_{\mathbf{V}}^{(0)}\times k_{\mathbf{U}}^{(1)}}\mathbf{V}_{\mathbf{U}}^{(1),k_{\mathbf{U}}^{(1)}\times k_{\mathbf{U}}^{(0)}}, \tag{10.11}$$

where

$$k_{\mathbf{V}}^{(0)} = k_{\mathbf{A}} = k_{\mathbf{V},1}^{(0)} + k_{\mathbf{V},2}^{(0)} + k_{\mathbf{V},3}^{(0)} + k_{\mathbf{V},4}^{(0)}, \tag{10.12}$$

and

$$k_{\mathbf{U}}^{(0)} = k_{\mathbf{B}} = k_{\mathbf{U},1}^{(0)} + k_{\mathbf{U},2}^{(0)} + k_{\mathbf{U},3}^{(0)} + k_{\mathbf{U},4}^{(0)}\,. \tag{10.13}$$

We note that at each step in the reconstruction, the dimensions of the intermediate products comprise the rank sum of matrices on that level, or on the parent level. As each product is relatively small, they require much less workspace and can be computed very quickly.

Having now computed $\mathbf{T}_1^{k_{\mathbf{A}}\times k_{\mathbf{B}}}$ in (10.2), the choice of whether to perform the left or right product first is made by minimizing the storage of $\mathbf{T}_2^{k_{\mathbf{A}}\times n_2}$ and $\mathbf{T}_3^{m_1\times k_{\mathbf{B}}}$. As the top-level groups were sorted by size, each diagonal is of size $N_b \times N_b$, increasing to $N_{max} \times N_{max}$, and so we are always guaranteed maximum sizes of $m_1 = N_b$ and $n_2 = N_{max}$, or vice versa, depending on whether we are updating trailing **U** blocks or **L** blocks, respectively. Thus, if $m_1 > n_2$, we take the first path and compute $\mathbf{T}_2^{k_{\mathbf{A}}\times n_2}$, otherwise we take the second path and compute $\mathbf{T}_3^{m_1\times k_{\mathbf{B}}}$.

10.2.3 MLACA Block-RHS Solution

For problems having many right-hand sides, we can again apply Algorithm 8 with a few changes as summarized in Algorithm 13. Prior to the back-substitutions in steps 4 and 10, we first reconstruct $[\mathbf{LU}]_{bb}$ to full size, which is fast and adds little overhead. Though the blocks of the **LU** matrix are MLACA-compressed, in this case we will use the single-level ACA to compress the RHS blocks. Thus, there will be matrix products between a U or V-type MLACA compressed **LU** matrix block and single-level ACA compressed RHS matrix block, as illustrated in Figures 10.9a and 10.9b, respectively. In both cases, the middle product is carried out first, yielding

$$\mathbf{C}^{k_\mathbf{A} \times k_\mathbf{B}} = \mathbf{A}_v^{k_\mathbf{A} \times n_1} \mathbf{B}_u^{m_2 \times k_\mathbf{B}}. \tag{10.14}$$

If **A** is U-type, whether $\mathbf{A}_u\mathbf{C}$ or $\mathbf{C}\mathbf{B}_v$ is performed next depends on which path results in the fewest total operations. If **A** is instead V-type, we perform the leftmost product $\mathbf{A}_u\mathbf{C}$ first as $\mathbf{A}_u$ is a sparse block matrix. This yields

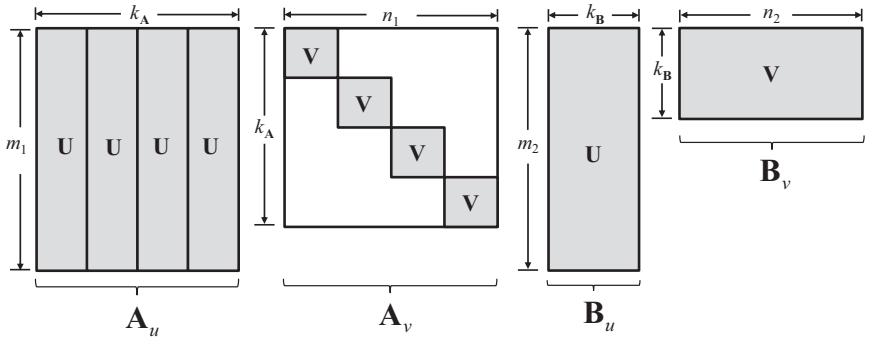
$$\mathbf{D}^{m_1 \times k_\mathbf{B}} = \mathbf{A}_u^{m_1 \times k_\mathbf{A}} \mathbf{C}^{k_\mathbf{A} \times k_\mathbf{B}} \tag{10.15}$$

and then finally

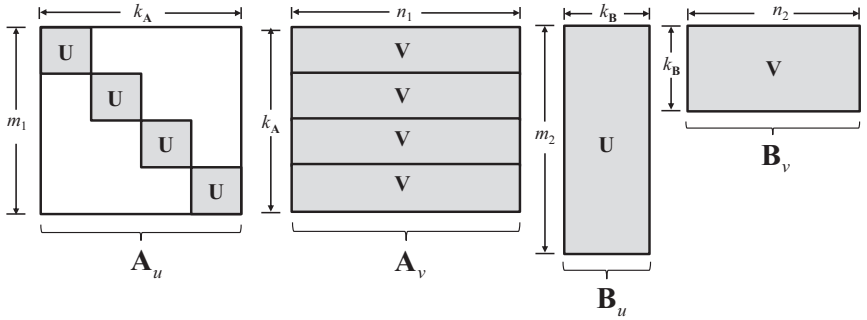
$$[\mathbf{AB}]^{m_1 \times n_2} = \mathbf{D}^{m_1 \times k_\mathbf{B}} \mathbf{B}_v^{k_\mathbf{B} \times n_2}. \tag{10.16}$$

Algorithm 13 MLACA-Compressed Block-RHS Solution

1: Given $\mathbf{LU}\ \mathbf{X} = \mathbf{B}$, first solve $\mathbf{UX} = \mathbf{L}^{-1}\mathbf{B}$:
2: **for** $b = 1$ to M **do**
3: Reconstruct $[\mathbf{LU}]_{bb}$
4: $\mathbf{B}_b = \mathbf{L}_{bb}^{-1}\left[[\mathbf{B}_u\mathbf{B}_v]_b - \sum_{s=1}^{b-1}[\mathbf{L}_u\mathbf{L}_v]_{bs}[\mathbf{B}_u\mathbf{B}_v]_s\right]$
5: Compress $\mathbf{B}_b$ via ACA and store updated $[\mathbf{B}_u\mathbf{B}_v]_b$
6: **end for**
7: Then solve $\mathbf{X} = \mathbf{U}^{-1}\left[\mathbf{L}^{-1}\mathbf{B}\right]$
8: **for** $b = M$ to 1 **do**
9: Reconstruct $[\mathbf{LU}]_{bb}$
10: $\mathbf{X}_b = \mathbf{U}_{bb}^{-1}\left[[\mathbf{B}_u\mathbf{B}_v]_b - \sum_{s=b+1}^{M}[\mathbf{U}_u\mathbf{U}_v]_{bs}[\mathbf{X}_u\mathbf{X}_v]_s\right]$
11: Compress $\mathbf{X}_b$ via ACA and store $[\mathbf{X}_u\mathbf{X}_v]_b$
12: **end for**



(a) U-type MLACA Matrix and ACA Matrix



(b) V-type MLACA Matrix and ACA Matrix

Figure 10.9: Product of MLACA Matrix and ACA Matrix

10.3 Software Implementation Notes

In this section, we discuss how MLACA support was implemented in our *Serenity* MoM solver. Much of what is discussed here builds upon what was done previously with the ACA in Section 9.6, and as before, what is presented here consists only of programming suggestions based on the author's experience.

MLACA support was added to *Serenity* several years after the ACA capability was originally developed. Almost all of the existing software architecture and C++ class library were reused as-is, or with only minor changes and additions. The new code provides class support for the MLACA matrix and compression operations, as well as MLACA matrix products on the CPU and GPU.

10.3.1 Software Class Support

In this section, we discuss the software classes that were implemented or updated in support of MLACA operations.

10.3.1.1 Abstract Matrix Class

We note that in addition to full rows and columns, on levels $l < L$ the U- and V-type MLACA algorithms require *partial* rows and columns of a matrix, respectively, when compressing sub-blocks. Thus, the `AbstractMatrix`, previously defined in Section 9.6.1.2.1, was updated to provide two new functions:

1. `getPartialRow`(r, c_1, c_2): Returns columns c_1 to c_2 from row r of a matrix block.

2. `getPartialColumn`(c, r_1, r_2): Returns rows r_1 to r_2 from column c of a matrix block.

These functions are implemented by the `ElementEngine` class or the `MLACATranslator` class, as discussed in the following sections.

10.3.1.2 Element Engine Class

The `ElementEngine` class, previously discussed in Section 9.6.1.1, now implements `getPartialRow`() and `getPartialColumn`(). During matrix filling, we note that only a subset of the triangles in the source group are needed to carry out these calculations, and these are known *a priori* due to the hierarchical clustering of basis functions in each group (Section 10.1.2). Thus, at initialization, the `ElementEngine` assembles for each group a list of triangles needed to populate a sub-row or sub-column when that group is a source group, minimizing compute time during the matrix fill.

10.3.1.3 MLACA Translator Class

The `ACAMatrixTranslator` class, described previously in Section 9.6.1.2.5, provides an `AbstractMatrix` class interface to the ACA driver while interfacing with the element engine. For MLACA-based matrix filling, we implement a

`MLACAMatrixTranslator` class, which derives from `ACAMatrixTranslator` and implements the `getPartialRow` and `getPartialColumn` methods. Top-level ACA functions, such as the `getRow` and `getColumn` methods, simply use existing base class methods.

10.3.1.4 MLACAMatrix and MLACANode Classes

The `ACAMatrix` class, discussed previously in Section 9.6.1.2.3, is a container class that stores the **U** and **V** components of a single-level ACA compressed matrix block. Objects of this class are generated as a product of the ACA+QR/SVD compression (Section 9.2). Compression of a block using MLACA is a recursive operation, and as a result there is information at each node in the binary tree required to store and reconstruct the original block. Thus, in support of MLACA operation, we implement two new classes. The `MLACANode` class represents a node in the MLACA binary tree. This class has functions that permit it to construct and store its compressed, notional $\mathbf{A}^{(l)}$ and $\mathbf{B}^{(l)}$ blocks from the matrix on the previous level. The `MLACAMatrix` class is a top-level container class, implementing all of the virtual functions of the `AbstractMatrix` class (see Section 9.6.1.2.1), and stores the root-level `MLACANode`.

Filling of off-diagonal blocks using the MLACA is similar to the single-level ACA. During the MLACA recursion, upper-level `MLACANode` objects in the MLACA tree store only **U** or **V** matrices, for V- or U-type MLACA, respectively, whereas the leaf nodes store both **U** and **V**. These are stored in a single array in memory, with each `MLACANode` storing integer offsets into this array.

10.3.1.5 HMatrix Class

The `BlockMatrix` class, discussed previously in Section 9.6.1.2.4, is a high-level class responsible for storing pointers to all matrix blocks in row-major format. In that class, the diagonal blocks comprised `FullMatrix` blocks, and off-diagonals `ACAMatrix` blocks. Though this behavior was hard-coded, it was sufficient to support ACA operations.

In the MLACA, we now compress the diagonal blocks. Each diagonal block is subdivided recursively, with `MLACAMatrix` off-diagonal sub-blocks on each level, and `FullMatrix` diagonal sub-blocks on the finest level. The diagonal sub-blocks on coarser levels, however, remained undefined. For these, a generalized block matrix container class similar to `BlockMatrix` is needed. As a result, the `BlockMatrix` class was refactored and replaced by the `HMatrix` class, where restrictions on the types of blocks that can be stored at any position have been removed.

10.3.2 Shared Memory Processing

For computing and solving the MLACA-compressed matrix system on shared memory systems, we derive from the same CPU and GPU thread classes implemented previously for the ACA, reusing most existing functionality. A few changes are needed, as well as the addition of new code, as discussed in Sections 10.3.2.2 and 10.3.2.3.

10.3.2.1 Reconstruction of Blocks and Intermediate Products

Prior to updating a MLACA block, we must first reconstruct the top-level $\mathbf{U}_0^{(1)}$ (U-type) or $\mathbf{V}_0^{(1)}$ (V-type) matrix. This is done on the CPU, following Section 10.2.1. Similarly, prior to any matrix product, the top-level intermediate product $\mathbf{T}_1$ in (10.2) is also computed on the CPU following Section 10.2.2.2.

Note that these reconstructions comprise a large number of small matrix products. On the GPU, however, matrix multiplication is most efficient for larger products executed in a single kernel. Consequently, reconstruction on the GPU was found to be slower than on the CPU, and so these steps are always done on the CPU.

10.3.2.2 MLACA CPU Thread Class

The MLACA CPU thread class inherits from the ACA CPU thread class, reusing all of the existing functions for matrix filling, LU factorization, and block-RHS solving. New functions are added for the matrix products outlined in (10.3-10.6) and (10.14-10.16).

10.3.2.3 MLACA GPU Thread Class

Similarly, the MLACA GPU thread class inherits from the ACA GPU thread class, reusing many of the existing functions, with new functions added to support the computation of MLACA matrix products on the GPU. The key differences involve workspace management, as discussed next.

10.3.2.3.1 Matrix Filling In MLACA, the diagonal blocks are compressed. Thus, each thread needs a device workspace of size $S_{max}^L = N_{max}^L \times N_{max}^L$ for storing full diagonal blocks on level $l = L$, where N_{max}^L is the largest sub-block size on that level. Additionally, each thread also needs a workspace of size N_{max} for the rows and columns computed for all off-diagonal blocks on the top level $l = 0$. Thus, a single workspace of size $\max(S_{max}^L, N_{max})$ is allocated for matrix-filling operations on the device.

10.3.2.3.2 LU Factorization Staging of prior $\mathbf{L}$ and $\mathbf{U}$ blocks on the GPU is still done, however, because $\mathbf{T}_1$ is computed on the CPU, only the top-level $\mathbf{V}_0^{(1)}$ (U-type) and $\mathbf{U}_0^{(1)}$ (V-type) block-matrix portions are staged. As before, this is disabled if GPU workspace becomes limited.

For each thread, device workspaces of size $N_b \times N_{max}$ are again needed to store the full-size $\mathbf{L}_{sb}$ and $\mathbf{U}_{bs}$ blocks. Workspaces are also needed for the top-level portions of the product matrices to be multiplied (or one, if the other was staged), and the intermediate products $\mathbf{T}_1$, $\mathbf{T}_2$ and $\mathbf{T}_3$. These are again sized by searching all matrix blocks above and to the right of the diagonal, and all blocks to the left and below the diagonal. In this case, we find the maximum top-level block-matrix size $S_{max} = \max\left(\max(\mathbf{U}_0^{(1)}), \max(\mathbf{V}_0^{(1)})\right)$ and the maximum rank sum K_{max}. Two spaces of size S_{max} are reserved for the top-level portions (or one, if staging), as well as a space of size $K_{max} \times K_{max}$ for $\mathbf{T}_1$ and a space of size $N_b \times K_{max}$ for $\mathbf{T}_2$ and $\mathbf{T}_3$.

10.3.3 Distributed Memory Processing

After implementing the code changes needed for MLACA-based shared memory processing, it was found that our existing distributed memory implementation for the single-level ACA, as discussed previously in Section 9.6.3, required very few code changes to support MLACA. The only new code implemented specifically for MPI operation was for serialization and deserialization of the MLACA matrix, required for message passing of that data type.

10.4 Numerical Examples

In this section we compare the performance of the MLACA algorithm to the single-level ACA in computing the radar cross section of several test articles. We will again look at the conducting, dielectric, and coated sphere, benchmark targets with measured data, and the monoconic RV with dielectric nose considered previously in Section 9.7. The electrical size of these problems is again larger than in prior chapters, taking advantage of the extra compression afforded by MLACA. For each test case, our *Serenity* solver code is used, and triangle meshes are constructed with at least 10 unknowns per wavelength in each dielectric region. The CFIE is applied on closed conductors with $\alpha = 0.5$, and PMCHWT applied to dielectric interfaces.

10.4.1 Compute Platform

Calculations in this section were carried out using two versions of the *Serenity* solver. The first is the shared memory, GPU-accelerated version, executed on the Dell workstation described previously in Section 9.7.1. The second is a distributed memory, CPU-only *Serenity* solver that uses MPI for inter-process communication. The MPI data distribution and parallelization strategy was discussed previously in Section 9.6.3. The distributed memory system comprised a 50-node compute cluster, each node having dual twelve-core Intel Xeon CPUs (E5-2680 v3) at 2.50GHz (1200 total CPUs) and 128 GB RAM per node (6.4 TB total RAM), connected via a 10 gigabit Ethernet network fabric, running Red Hat Enterprise Linux.

10.4.2 Conducting Spheres

We first compare the memory and CPU requirements of the single-level ACA and MLACA, in both matrix filling and LU factorization, for perfectly electrically conducting (PEC) spheres of radius a ranging from 1 to 32 λ. The GPU-accelerated *Serenity* solver is used. The number of triangles N_T and unknowns N for each sphere are summarized in Table 10.1. The target size of top-level basis function groups was $N_g = 4000$ unknowns, and $\tau_{ACA} = 10^{-4}$ for both ACA and MLACA. This τ_{ACA} was chosen so that the examples (except $a = 32\lambda$) would fit in available RAM. A much smaller value ($\tau_{ACA} \leq 10^{-5}$) would be required for an accurate solution. In this study, we only present results from MLACA levels up to $L = 4$, as the compression achieved for levels $L \geq 5$ was not significantly greater than for $L = 4$ and $N_g = 4000$. We will examine the effect of increasing the group size further in a later section.

Table 10.1: PEC Spheres: Dimensions and Number of Unknowns

$a(\lambda)$	1	2	4	8	16	32
N_T	5120	20480	81920	327680	1310720	5242880
N	7680	30720	122880	491520	1966080	7864320

Table 10.2: PEC Spheres: Memory ($M_{\mathbf{Z}}$, GB), Fill Time ($T_{\mathbf{Z}}$, seconds)

	ACA		$L=1$		$L=2$		$L=3$		$L=4$	
$a(\lambda)$	$M_{\mathbf{Z}}$	$T_{\mathbf{Z}}$	$M_{\mathbf{Z}}$	$T_{\mathbf{Z}}$	$M_{\mathbf{Z}}$	$T_{\mathbf{Z}}$	$M_{\mathbf{Z}}$	$T_{\mathbf{Z}}$	$M_{\mathbf{Z}}$	$T_{\mathbf{Z}}$
1	.093	1.2	.067	1.6	.052	2.1	.047	3.0	.051	4.4
2	1.0	5.6	.68	5.4	.44	6.4	0.32	8.1	.26	10.9
4	5.9	19	3.7	19	2.5	21	1.8	28	1.4	38
8	30	90	20	90	14	110	11	158	10	246
16	174	634	135	715	112	954	97	1472	81	2345
32*	1560	4680	1373	6120	1125	9000	896	12960	748	23400

* Results Estimated.

Table 10.3: PEC Spheres: Memory ($M_{\mathbf{LU}}$, GB), LU Factorization Time ($T_{\mathbf{LU}}$, seconds)

	ACA		$L=1$		$L=2$		$L=3$		$L=4$	
$a(\lambda)$	$M_{\mathbf{LU}}$	$T_{\mathbf{LU}}$	$M_{\mathbf{LU}}$	$T_{\mathbf{LU}}$	$M_{\mathbf{LU}}$	$T_{\mathbf{LU}}$	$M_{\mathbf{LU}}$	$T_{\mathbf{LU}}$	$M_{\mathbf{LU}}$	$T_{\mathbf{LU}}$
1	.093	.5	.067	.9	.053	.8	.048	1.0	.052	1.2
2	1.0	7.6	.69	7.5	.46	7.7	.34	7.7	.27	9.4
4	6.5	41	4.1	51	2.8	46	2.1	48	1.7	53
8	37	562	25	566	18	563	14	568	12	579
16	248	21485	191	19594	151	21028	126	21333	110	21840

Table 10.2 summarizes the matrix fill time ($T_{\mathbf{Z}}$, in seconds) and the memory requirements of the MoM matrix $\mathbf{Z}$ ($M_{\mathbf{Z}}$, in GB). The matrix for $a = 32\lambda$ would not fit in available RAM, however the final matrix size and fill time can be accurately estimated by *Serenity* during the fill operation. Thus, the results for this case were obtained by allowing *Serenity* to run until memory was exhausted.

Similarly, Table 10.3 summarizes the LU factorization time and the memory requirements of the **LU** matrix ($M_{\mathbf{LU}}$, in GB). To show the differences more clearly, Table 10.4 are summarized the percent difference (decrease) in the sizes of the $\mathbf{Z}$ and **LU** matrix when using MLACA versus the single-level ACA. The additional compression obtained using MLACA is substantial, with its effectiveness increasing with level. From these results, we note two details: For $a \geq 4$, MLACA compresses the LU blocks to a greater degree than it does the $\mathbf{Z}$ blocks. We also see that the increase in compression peaks for $2 \leq a \leq 4$ and then decreases somewhat. This is due to the diagonal blocks, which comprise an increasingly smaller fraction of all matrix blocks as the problem size and number of groups increase.

Table 10.4: Percent Difference (decrease) in $M_{\mathbf{Z}}$ and $M_{\mathbf{LU}}$: MLACA versus Single-Level ACA

	$M_{\mathbf{Z}}$						$M_{\mathbf{LU}}$				
$a(\lambda)$	1	2	4	8	16	32	1	2	4	8	16
$L=1$	32	38	47	40	25	13	32	37	45	59	26
$L=2$	57	78	82	73	43	32	55	74	80	88	49
$L=3$	66	103	108	93	57	54	64	99	102	107	65
$L=4$	138	117	124	100	73	70	57	115	117	117	77

Table 10.5: Percent Difference (increase) in Fill Time $T_{\mathbf{Z}}$ and LU Factorization Time $T_{\mathbf{LU}}$: MLACA versus Single-Level ACA

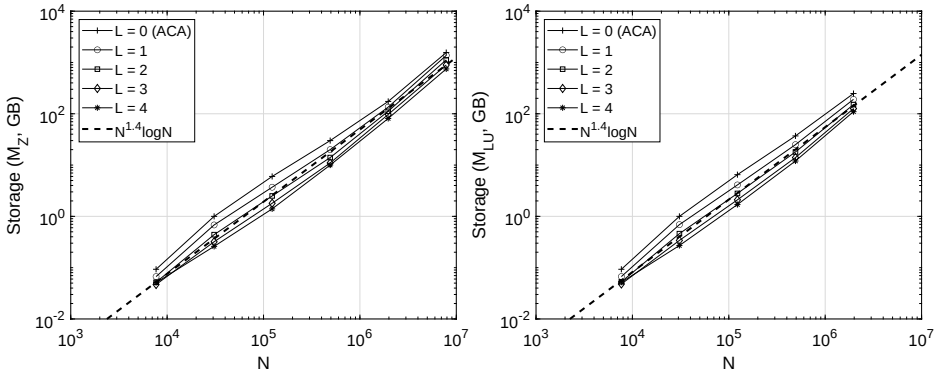
	$T_{\mathbf{Z}}$				$T_{\mathbf{LU}}$		
$a(\lambda)$	4	8	16	32	4	8	16
$L=1$	0	0	12	27	22	1	9
$L=2$	10	20	40	63	11	0	2
$L=3$	38	55	80	94	16	1	1
$L=4$	67	93	115	133	26	3	2

Table 10.5 summarizes the percent difference (increase) in matrix fill time $T_{\mathbf{Z}}$ and LU factorization time $T_{\mathbf{LU}}$. We have omitted results for $a < 4$ as these runs were so short that system overhead and other effects caused large variances in total run time. The increases in $T_{\mathbf{Z}}$ is as expected, where the additional compression steps add significantly to the run time. However the increases in $T_{\mathbf{LU}}$ are far more moderate, particularly for $a = 8\lambda$ and $a = 16\lambda$ where the increase in run time is only a few percent for all L. This suggests that the block updates (prior to recompression) consume the most time in the LU factorization, and that our MLACA block LU factorization is almost as efficient as our single-level ACA factorization.

In Figures 10.10a and 10.10b is shown the memory requirements $M_{\mathbf{Z}}$ and $M_{\mathbf{LU}}$ versus the number of unknowns. In both figures, we observe that for larger problem sizes ($a \geq 8$), the ACA and MLACA both have a complexity of roughly $O(N^{1.4}\log N)$, somewhat higher than the $O(N^{1.28}\log N)$ complexity observed for the ACA with smaller problems in Section 9.7.8.

10.4.2.1 Variation of Target Group Size

We next examine the performance of MLACA versus the target group size N_g for the 16λ sphere having $N = 1966080$ unknowns. Summarized in Table 10.6 are the group sizes considered, and the resulting minimum (N_{min}), maximum (N_{max}), and average (N_{avg}) number of basis functions per group. We note that while $N_g \approx N_{avg}$ in each case, there is some variation about the average. This occurs because the algorithm we used to generate the sphere does not create facets of exactly the same size, and because



(a) $M_{\mathbf{Z}}$ Storage versus N (b) $M_{\mathbf{LU}}$ Storage versus N

Figure 10.10: Storage versus Number of Unknowns N for PEC Spheres

Table 10.6: 16λ PEC Sphere: Target Group Size (N_g), Minimum (N_{min}), Maximum (N_{max}), Average (N_{avg})

N_g	N_{min}	N_{max}	N_{avg}
2500	1796	3516	2498
5000	3746	6853	4990
7500	6175	10165	7475
10000	8119	13732	9980
12500	10215	16083	12443

the K-means algorithm is iterative and does not guarantee the same number of basis functions in each group.

Table 10.7 summarizes the memory requirements $M_{\mathbf{Z}}$ and $M_{\mathbf{LU}}$ for the single-level ACA and MLACA for levels $L \leq 5$. Several trends are notable in these data. For small N_g, the compression improves more slowly (or degrades) with increasing L, and similar behavior occurs for small L and increasing N_g. However, compression increases dramatically as both N_g and L are increased. In particular, we note that for $N_g = 12500$, $M_{\mathbf{Z}} = 69$ GB and $M_{\mathbf{LU}} = 127$ GB for the $L = 5$ MLACA, respectively, versus $M_{\mathbf{Z}} = 327$ GB and $M_{\mathbf{LU}} = 499$ GB for the single-level ACA.

Table 10.7: 16λ PEC Sphere: Memory ($M_{\mathbf{Z}}$, GB), Memory ($M_{\mathbf{LU}}$, GB) versus Target Group Size

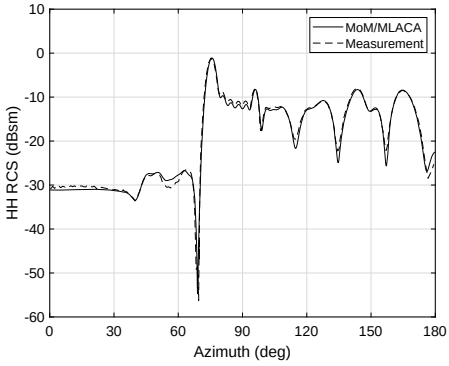
	ACA		$L=1$		$L=2$		$L=3$		$L=4$		$L=5$	
N_g	$M_{\mathbf{Z}}$	$M_{\mathbf{LU}}$	$M_{\mathbf{Z}}$	$M_{\mathbf{LU}}$	$M_{\mathbf{Z}}$	$M_{\mathbf{LU}}$	$M_{\mathbf{Z}}$	$M_{\mathbf{LU}}$	$M_{\mathbf{Z}}$	$M_{\mathbf{LU}}$	$M_{\mathbf{Z}}$	$M_{\mathbf{LU}}$
2500	284	582	214	480	180	400	174	355	181	340	209	368
5000	247	525	190	408	152	317	117	262	103	233	105	229
7500	269	487	185	362	140	271	112	215	89	182	79	171
10000	295	487	197	347	136	251	106	193	87	158	71	142
12500	327	499	210	346	139	243	104	182	82	145	69	127

10.4.3 Polyethylene Cone-Sphere

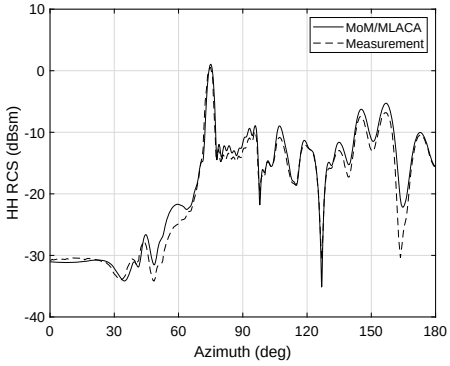
Next, we consider the RCS of a polyethylene conesphere, another benchmark target fabricated and measured in [6]. This object has the same dimensions as the EMCC cone-sphere described in Section 7.9.2.3, with an error of ± 0.005 inches. The average permittivity $\epsilon_r \approx 2.35$, and only the HH-polarized RCS could be extracted from the figures in [6]. As with the polyethylene ogive in Section 9.7.5.2, a single facet model was constructed and used at all frequencies. This model had 371936 triangles, resulting in 557904 edges and 1115808 unknowns. For the MoM solution, an $L = 2$ level MLACA was used, with a target group size of $N_g = 5000$ unknowns. The memory requirements $M_{\mathbf{Z}}$ and $M_{\mathbf{LU}}$ versus frequency are summarized in Table 10.8. As was observed with the polyethylene ogive, the compression here also decreases versus frequency. The computed and measured results are compared for horizontal polarization in Figures 10.11a–10.11h, where tip-on incidence corresponds to an azimuth angle of 180 degrees. The comparison is fairly good at all frequencies and incident angles.

Table 10.8: Polyethylene Cone-Sphere: Memory ($M_{\mathbf{Z}}$, GB), Memory ($M_{\mathbf{LU}}$, GB)

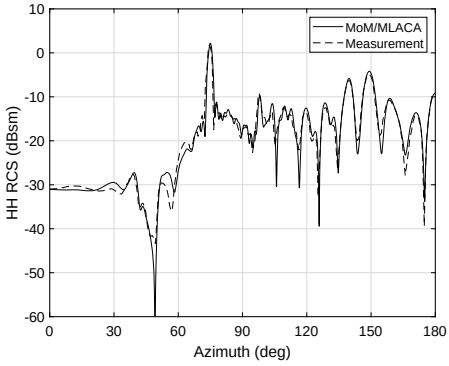
f_0	4.0	6.0	8.0	10.0	12.0	14.0	16.0	18.0
$M_{\mathbf{Z}}$	88	91	93	96	98	101	102	105
$M_{\mathbf{LU}}$	105	119	136	153	170	188	206	224



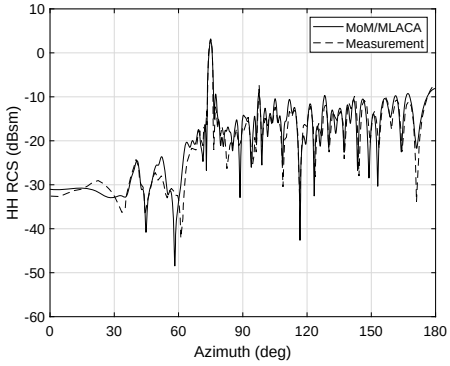
(a) 4.0 GHz, HH-Polarization



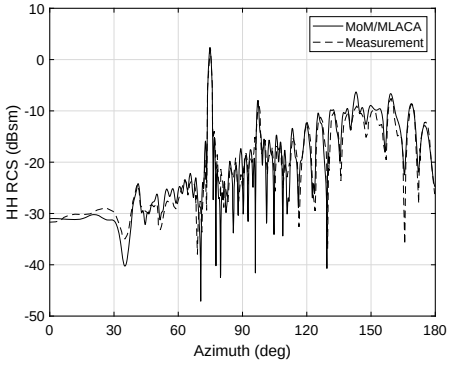
(b) 6.0 GHz, HH-Polarization



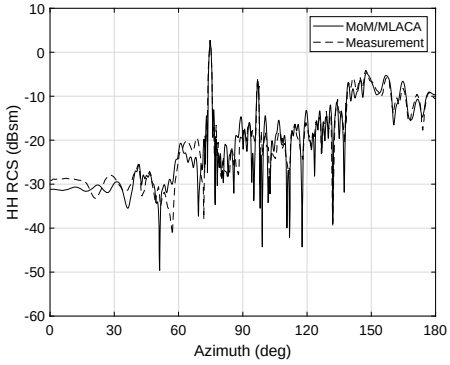
(c) 8.0 GHz, HH-Polarization



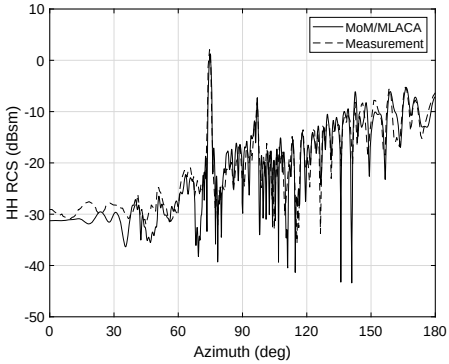
(d) 10.0 GHz, HH-Polarization



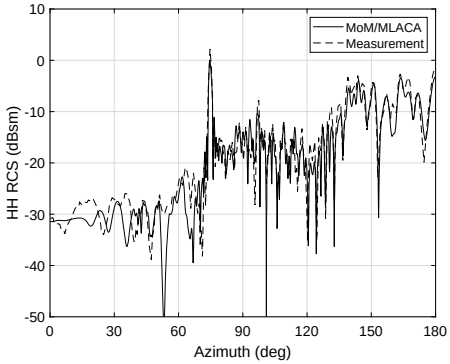
(e) 12.0 GHz, HH-Polarization



(f) 14.0 GHz, HH-Polarization



(g) 16.0 GHz, HH-Polarization



(h) 18.0 GHz, HH-Polarization

Figure 10.11: Polyethylene Cone-Sphere: RCS

10.4.4 Monoconic Reentry Vehicle

Next, we will examine the monoconic reentry vehicle considered previously in Sections 7.10.2.4, 8.8.8, and 9.7.7. In this case, we consider only the RV with dielectric nose, and we will compare the results from the single-level ACA to MLACA. We compute the monostatic RCS at 12.0, 14.0, 16.0, 20.0, and 24.0 GHz, where the number of triangles N_T, electric basis functions N_J, magnetic basis functions N_M, and total number of unknowns N in each model are summarized in Table 10.9. For each example, $\tau_{ACA} = 10^{-5}$, and the target group size is $N_g = 10000$ unknowns. The results from the single-level ACA and $L = 5$ level MLACA are compared for vertical and horizontal polarizations across all frequencies in Figures 10.12a–10.12j, The comparison is very good.

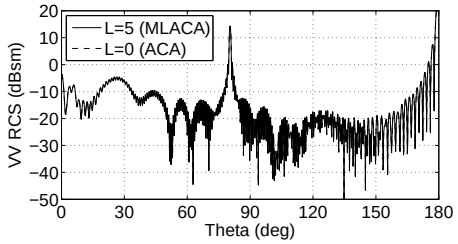
Table 10.9: Monoconic Reentry Vehicle: Number of Triangles and Unknowns

f_0(GHz)	12.0	14.0	16.0	20.0	24.0
N_T	787100	1065932	1167268	1842152	3252068
N_J	1180548	1598780	1750780	2763076	4877898
N_M	33348	46988	48220	80212	144420
N	1213896	1645768	1799000	2843288	5022318

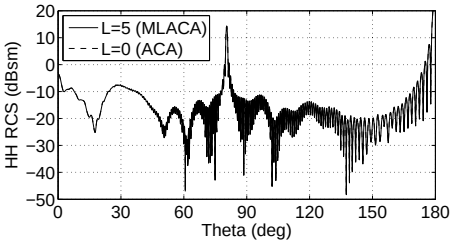
Table 10.10 summarizes the memory requirements $M_{\mathbf{Z}}$ and $M_{\mathbf{LU}}$ for the single-level ACA and MLACA for levels $L \leq 5$. Using MLACA again results in a dramatic improvement in total memory usage versus the single-level ACA, particularly at higher frequencies when using more MLACA levels. To visualize these trends, the data are plotted in Figures 10.13a and 10.13b for $M_{\mathbf{Z}}$ and $M_{\mathbf{LU}}$, respectively. We note that for this particular problem, the $L = 5$ level MLACA yields the greatest compression, however the additional improvement decreases significantly between $L = 4$ and $L = 5$. It is unlikely that an $L = 6$ level MLACA would yield any additional benefit for this value of N_g.

Table 10.10: Monoconic Reentry Vehicle: Memory ($M_{\mathbf{Z}}$, GB), Memory ($M_{\mathbf{LU}}$, GB)

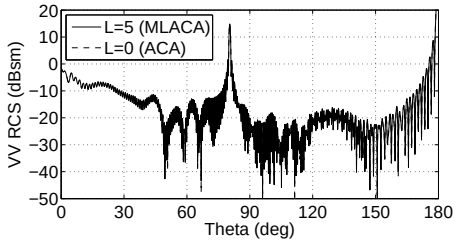
f_0	ACA		$L=1$		$L=2$		$L=3$		$L=4$		$L=5$	
	$M_{\mathbf{Z}}$	$M_{\mathbf{LU}}$	$M_{\mathbf{Z}}$	$M_{\mathbf{LU}}$	$M_{\mathbf{Z}}$	$M_{\mathbf{LU}}$	$M_{\mathbf{Z}}$	$M_{\mathbf{LU}}$	$M_{\mathbf{Z}}$	$M_{\mathbf{LU}}$	$M_{\mathbf{Z}}$	$M_{\mathbf{LU}}$
12.0	151	184	97	124	66	88	49	67	39	55	34	49
14.0	222	284	144	196	98	141	74	108	59	89	52	80
16.0	245	328	161	231	112	170	85	133	69	110	61	101
20.0	429	666	293	491	211	372	165	299	137	253	125	234
24.0	884	1841	634	1156	473	888	377	720	316	608	291	554



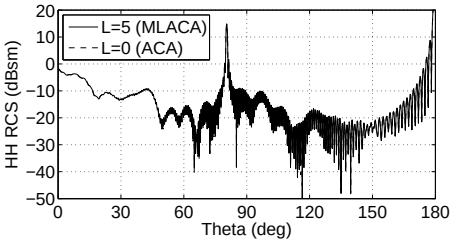
(a) 12.0 GHz, VV-Polarization



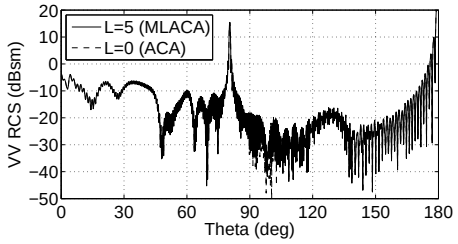
(b) 12.0 GHz, HH-Polarization



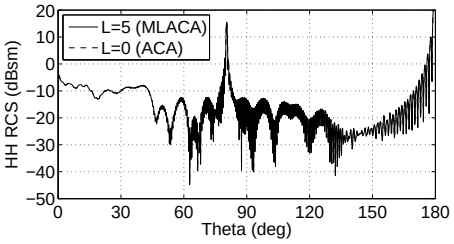
(c) 14.0 GHz, VV-Polarization



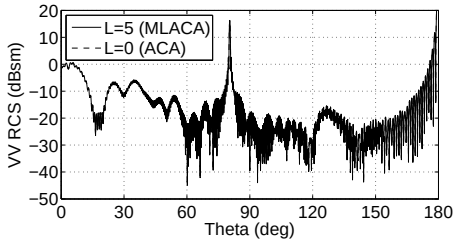
(d) 14.0 GHz, HH-Polarization



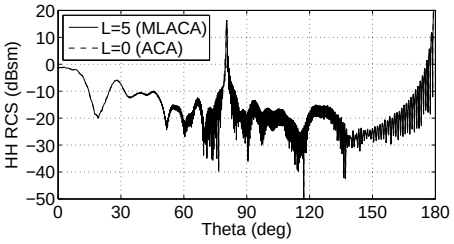
(e) 16.0 GHz, VV-Polarization



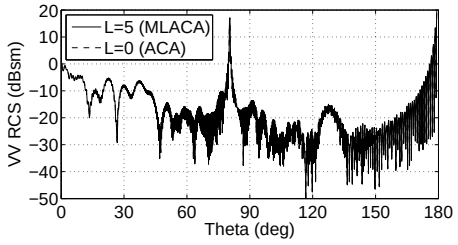
(f) 16.0 GHz, HH-Polarization



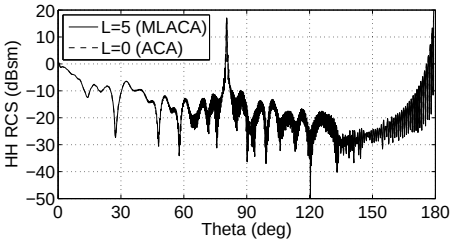
(g) 20.0 GHz, VV-Polarization



(h) 20.0 GHz, HH-Polarization

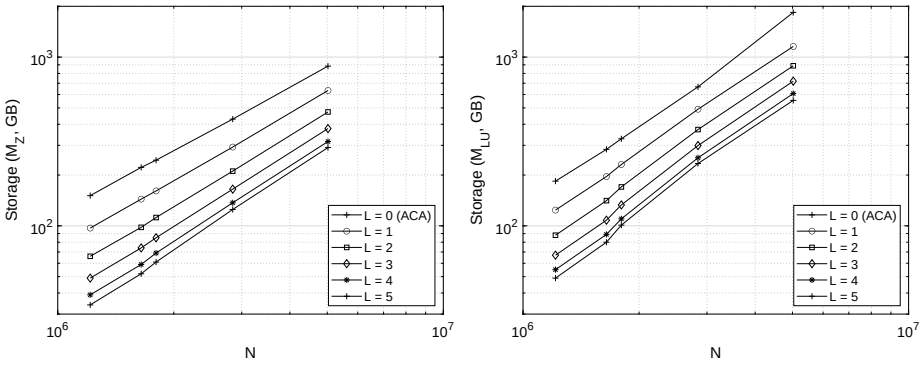


(i) 24.0 GHz, VV-Polarization



(j) 24.0 GHz, HH-Polarization

Figure 10.12: RV with Lossless Nose: RCS at 12.0, 14.0, 16.0, 20.0 and 24.0 GHz

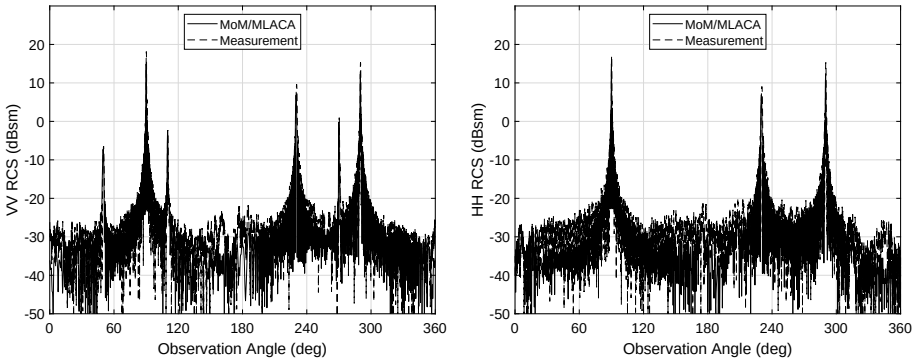


(a) $M_{\mathbf{Z}}$ Storage versus N (b) $M_{\mathbf{LU}}$ Storage versus N

Figure 10.13: Storage versus N For the Monoconic Reentry Vehicle

10.4.4.1 EMCC Prism

Finally, we will again consider the EMCC Prism from Section 9.7.4.7, this time at a frequency of 9.2 GHz. For the prism, a facet model with 1481992 triangles was constructed, yielding a total of 2222988 unknowns. For the MoM solution, an $L = 3$ level MLACA was used, with a target group size of $N_g = 5000$ unknowns. The results from the MLACA are compared to the EMCC measurements in Figures 10.14a and 10.14b, for vertical and horizontal polarizations, respectively. The comparison is fairly good in vertical polarization, however the MoM underpredicts the RCS in horizontal polarization at many angles. The cause for this disagreement is unknown, and no similar comparison was available in [7] for cross-referencing.



(a) VV-Polarization (b) HH-Polarization

Figure 10.14: EMCC Prism: MoM/MLACA vs. Measurement at 9.2 GHz

References

[1] E. Michielssen and A. Boag, "A multilevel matrix decomposition algorithm for analyzing scattering from large structures," *IEEE Trans. Antennas Propagat.*, vol. 44, pp. 1086–1093, August 1996.

[2] J. Tamayo, A. Heldring, and J. Rius, "Multilevel adaptive cross approximation (MLACA)," *IEEE Trans. Antennas Propagat.*, vol. 59, pp. 4600–4608, December 2011.

[3] J. Tamayo, *Multilevel Adaptive Cross Approximation and Direct Evaluation Method for Fast and Accurate Discretization of Electromagnetic Integral Equations*. PhD thesis, Polytechnic University of Catalonia, 2011.

[4] W. C. Gibson, "Efficient solution of electromagnetic scattering problems using multilevel adaptive cross approximation (MLACA) and LU factorization," *IEEE Trans. Antennas Propagat.*, vol. 68, pp. 3815–3823, May 2020.

[5] X. Chen, C. Gu, A. Heldring, Z. Li, and Q. Cao, "Error bound of the multilevel adaptive cross approximation (MLACA)," *IEEE Trans. Antennas Propagat.*, vol. 64, pp. 374–378, January 2016.

[6] J. M. Parks, "Scattering from Dielectric Bodies," Master's thesis, Air Force Institute of Technology, December 1997.

[7] A. Greenwood, "Electromagnetic Code Consortium Benchmarks," Tech. Rep. AFRL-DE-TR-2001-1086, Air Force Research Laboratory, December 2001.