# CS 124 Programming Assignment 3: Spring 2022

**Your name(s) (up to two):** Burak Ufuktepe

**Collaborators:** None

No. of late days used on previous psets: 19

No. of late days used after including this pset: ?????????

## DP Solution to the Number Partition Problem

Consider the Number Partition problem where we have a sequence of non-negative integers such that the sequence of terms in sum up to some number . We would like to find a sequence of signs such that the *residue*

is minimized. In other words, we would like to split , into two subsets and such that the absolute value of the difference of their sums is minimized. That is:

This expression is minimized when we have . So, the difference can be expressed as follows:

If is even, then so we will have:

If is odd the minimum difference can be 1. That is, so we will have:

Therefore, the problem becomes identifying a subset of whose sum of elements is or as close to as possible.

Let be a 2D boolean array that indicates whether there exists a subset of such that the sequence of terms in sum up to where and .

We initialize every element of to and then we set for all . This is because all sets have a subset (namely the empty set) whose elements sum to 0. Then we can populate the array by iterating over each row from to and in each iteration we iterate over each column from to . To determine we use the following recurrence relation:

To calculate the *residue*, we first identify the largest such that is . This value is essentially the sum of the elements of , that is . Hence, the sum of the elements of will be . So, the *residue* can be calculated as follows:

**Correctness:** we show that our recurrence relation correctly calculates . That is, for a given value of and where and , it correctly identifies whether there exists a subset of such that the sequence of terms in sum up to .

The proof is by induction on .

**Base Case:** when and when we have an arbitrary value of such that 1 there are three cases:

**Case 1:** if that means there does not exist a subset of whose terms sum up to . In this case, our recurrence relation sets to . Since is initialized to we will have , which is correct.

**Case 2:** if that means there exists a subset of whose terms sum up to , namely the set . In this case, our recurrence relation sets to or where . Since is initialized to we will have , which is correct.

**Case 3:** if that means there does not exist a subset of whose terms sum up to . In this case, our recurrence relation sets to or . Since both and are initialized to we will have , which is correct.

So, the base case holds.

**Inductive Hypothesis:** Assume that our recurrence relation correctly calculates where and 1 .

**Inductive Step:** We show that our recurrence relation correctly calculates . In the iteration there are two cases:

**Case 1:** if then we cannot include in the solution as its value exceeds . That is, we need to consider whether there exists a subset of whose terms sum up to . By the Inductive Hypothesis this was correctly calculated and stored in . Hence, we set to .

**Case 2:** if then we consider the following two conditions:

* If there exists a subset of whose terms sum up to , then we can use that subset as our solution. Hence, we can set to . By the Inductive Hypothesis, must be . Therefore, will be .
* If there exists a subset of whose terms sum up to -, then we can use that subset and include so that the sum of values is -. Hence, we can set to . By the Inductive Hypothesis, must be . Therefore, will be .
* If both and are then will be .

Since these are the only possibilities and, in each case, is correctly calculated we conclude that our recurrence relation is correct.

we can either include in the subset or

## Implementation

### Data Layout Optimization

Splitting matrices takes up a significant portion of the actual runtime in Strassen’s algorithm. To speed up this process, instead of using a standard row-major ordering, Morton ordering is used to represent matrices. Morton ordering takes a 2D array stored in row-major order and arranges the matrix in block arrays where is the size of the “base case”. This is illustrated in Figure 1 for and Figure 2 for .

Table

Description automatically generated

Figure 1: Row-Major Ordering vs Morton Ordering (2 x 2 blocks)

Diagram

Description automatically generated with medium confidence

Figure 2: Row-Major Ordering vs Morton Ordering (3 x 3 blocks)

This way, we end up with a 1D array and each quadrant is stored contiguously in memory as shown in Figure 3. To partition an matrix into four matrices we can simply slice the 1D array into four parts without having to iterate over each element.

Text

Description automatically generated with low confidence

Figure 3: 1D Array Representation of a Matrix using Morton Ordering

To compare the performance of Morton Ordering to Row-Major Ordering, several trials are conducted using random matrices where each entry is randomly selected to be 0, 1 or 2. Matrices of sizes and are tested using various cross-over points. Test results are given in Table 1. Furthermore, for each matrix size, runtime vs cross-over point plots are shown in Figure 4, Figure 5, and Figure 6 where the minimum runtimes are highlighted with red dots.

Table 1: Morton-Ordering vs Row-Major Ordering Runtime Comparison

|  |  |  |  |
| --- | --- | --- | --- |
| **n** | **Cross-Over Point** | **Runtime (s)** | |
| **Morton-Ordering** | **Row-Major Ordering** |
| 1536 | 6 | 243 | 337 |
| 1536 | 12 | 187 | 236 |
| 1536 | 24 | 176 | 202 |
| 1536 | 48 | 181 | 196 |
| 1536 | 96 | 192 | 202 |
| 1792 | 7 | 347 | 467 |
| 1792 | 14 | 278 | 343 |
| 1792 | 28 | 273 | 308 |
| 1792 | 56 | 277 | 303 |
| 1792 | 112 | 288 | 317 |
| 2048 | 4 | 704 | 1036 |
| 2048 | 8 | 475 | 624 |
| 2048 | 16 | 393 | 470 |
| 2048 | 32 | 398 | 422 |
| 2048 | 64 | 414 | 445 |
| 2048 | 128 | 451 | 468 |

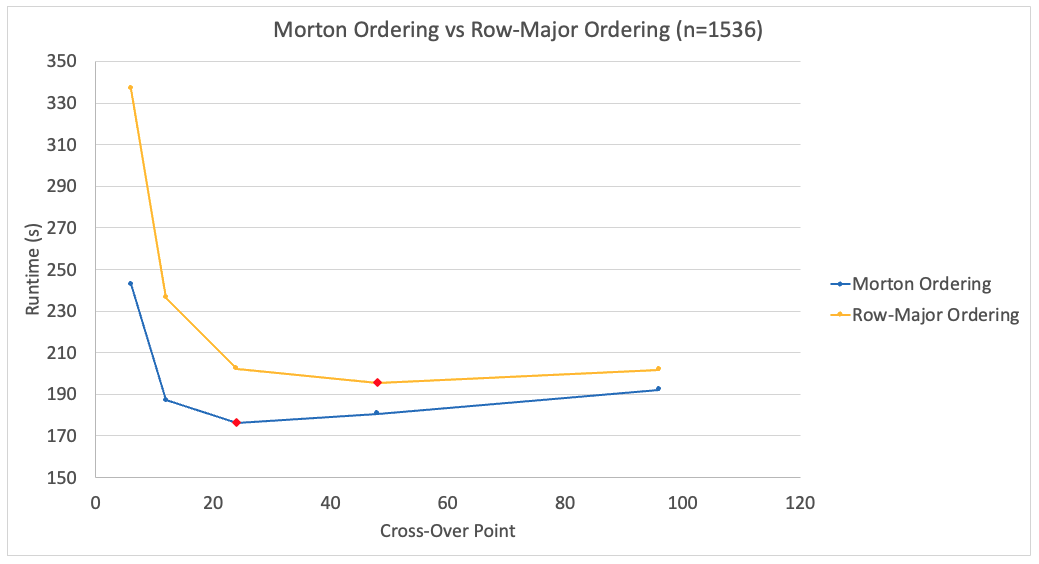


Figure 4: Morton Ordering vs Row-Major Ordering (n=1536)

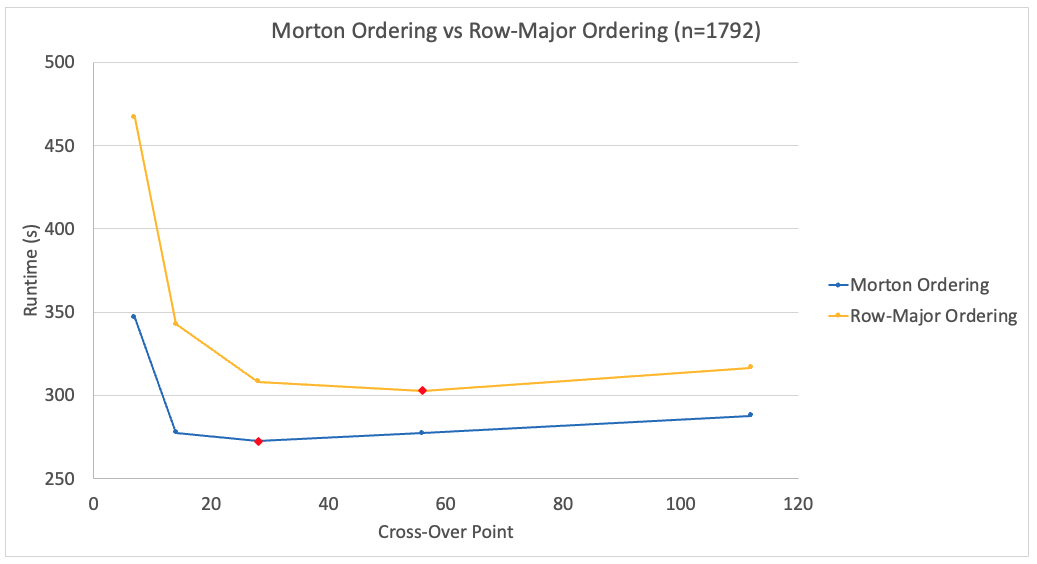


Figure 5: Morton Ordering vs Row-Major Ordering (n=1792)

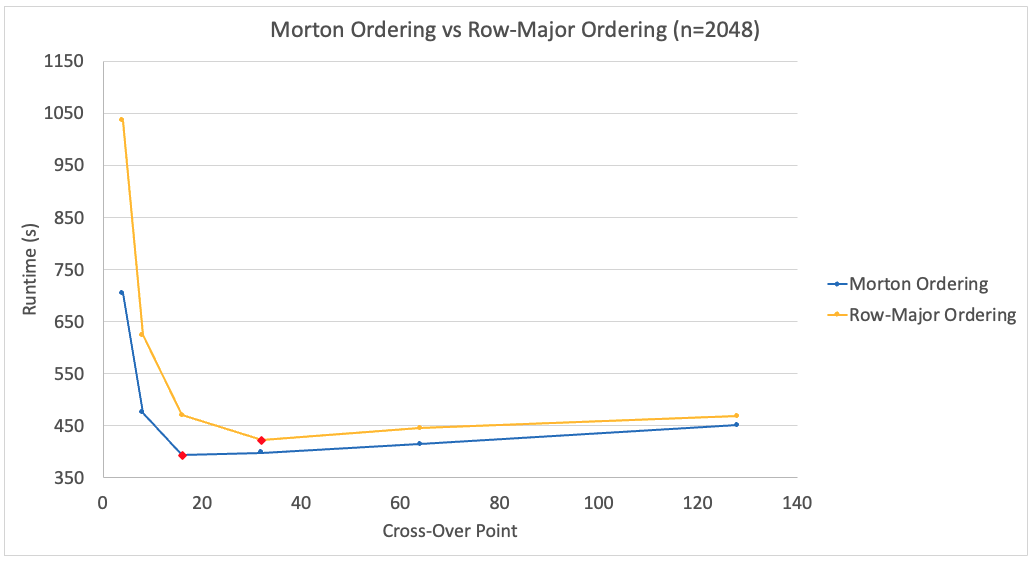


Figure 6: Morton Ordering vs Row-Major Ordering (n=2048)

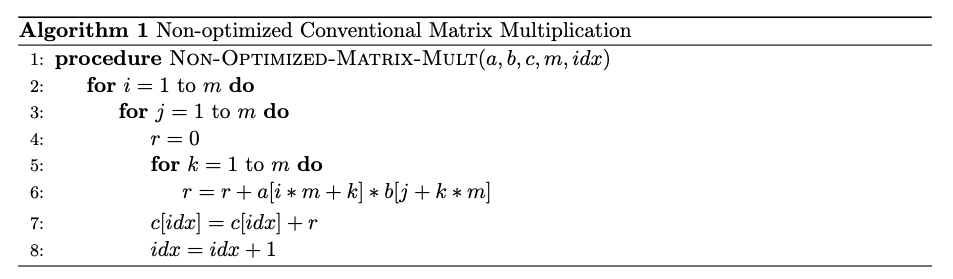
Morton Ordering shows a noticeable performance gain over Row-Major Ordering (up to 32%). The largest performance gain is observed when the cross-over point is small. This is because using a smaller cross-over point results in more recursive calls in Strassen’s algorithm, as a result the number of matrix partition operations increases. Since splitting up a matrix in Morton Ordering is a lot cheaper compared to Row-Major Ordering, Morton Ordering yields better results in terms of runtime.

Also note that the optimum cross-over point for Morton Ordering is lower than the optimum cross-over point for Row-Major Ordering. This is because Strassen’s algorithm runs more efficiently when the matrices are laid out in Morton Ordering. Whereas, when we use Row-Major ordering, the algorithm spends too much time tying to split up matrices.

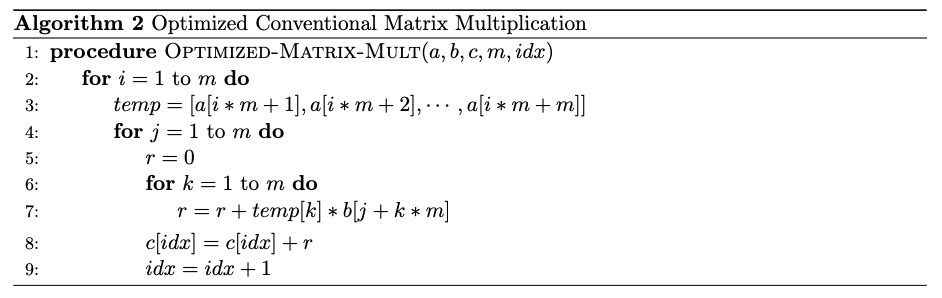
### Conventional Matrix Multiplication Optimization

In order to multiply two matrices that are Morton-ordered, the following procedure is used where and are matrices to be multiplied, is the resultant Morton-ordered matrix that is initially filled with zeros, is the size of the matrix, and is the current index of that will be populated. Note that this function is part of a recursive function, therefore we need an index argument () to identify the starting index for .

As you can see in the for loop of lines 3-8, the procedure accesses multiple times and at each time, it performs a computation to find the relevant index of . Furthermore, due to Morton ordering, the procedure jumps between contiguous blocks of memory which slows down the process.



When we bring a block of data into the cache, we would like it to contain as much useful data as possible and perform as much useful work as possible on it before removing it from the cache. Therefore, to optimize this procedure, the frequently accessed elements of are stored in a temporary array as shown in line 3 of the following procedure. Then, in line 7, this array is used to access the relevant element of . Since is a much smaller array it provides better cache performance (due to index locality). Furthermore, there is no computation involved to find the relevant index hence the runtime is further reduced.



Several trials have been conducted to compare the runtimes of these two procedures. Random matrices of sizes from up to are used and the results are given in Figure 7. It is observed that the optimized procedure runs 21% to 24% faster than the non-optimized procedure.

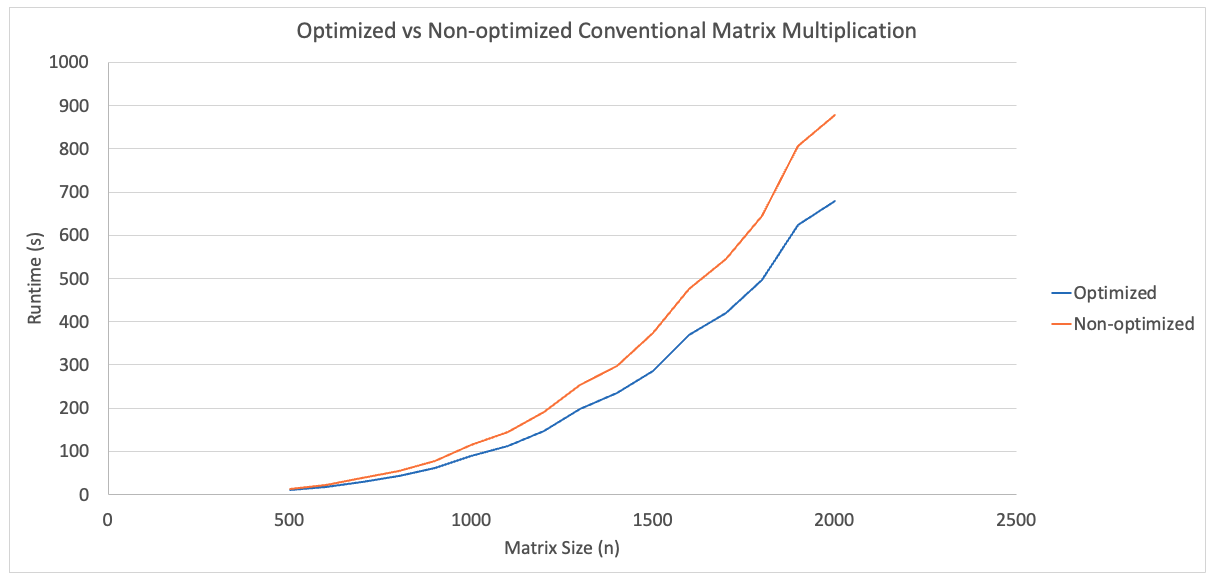


Figure 7: Optimized vs Non-optimized Conventional Matrix Multiplication

### Padding

Strassen’s algorithm recursively divides matrices into four matrices. That means at each recursive call must be divisible by 2. The obvious method to resolve this issue would be to pad the original matrix to the next power of 2 with zero rows and zero columns. However, this would be an expensive approach because we would have to almost double the size of the original matrix if its original size is just over a power of 2 (i.e ).

Instead, we use the following approach: we first find the size of the “base case” for our Morton-ordered array. Then we keep doubling that size until we reach or exceed the size of our original matrix to find the minimum required size that we need to perform Strassen’s algorithm. That is:

1. Let . Repeatedly divide in half, each time taking the ceiling, until is less than or equal to the cross-over point. This will be equal to the size of the “base case” for our Morton-ordered array.
2. Then repeatedly double until .
3. Pad the original matrix with zero rows and zero columns until we obtain a matrix of size .

In other words, we are padding the matrix just enough so that Strassen’s algorithm can reach the “base case”. One of the main advantages of this approach is, since the padding is done upfront, we do not have to worry about odd-sized matrices while performing Strassen’s algorithm.

For example, assume and the cross-over point is 37. We find the size of the base case as follows:

🡪 🡪 🡪 🡪

Then, to find the minimum required matrix size, we repeatedly double the base case until we reach or exceed :

🡪 🡪 🡪 🡪

Finally, we pad our original matrix until we reach a size of .

## Mulltiprocessing

To reduce the overall runtime, Python’s multiprocessing module is utilized while performing the experimental analysis to find the optimum cross-over point. The analysis is run on a laptop with a 10-core Apple M1 Pro CPU. For the initial trials, 8 parallel processes were used. However, due to overloading the CPU a high variance was observed in runtimes. Therefore, the experimental analysis was conducted using 4 parallel processes.

## Experimental Analysis

An experimental analysis is performed to find the optimum cross-over point. As explained in the previous section, our implementation pads the given matrices (if necessary) before performing Strassen’s algorithm. Therefore, the size of the matrices is always converted from to where is the size of the “base case” such that and is the smallest integer that satisfies . In other words, it doesn’t matter if we multiply odd-sized matrices or even-sized matrices because our algorithm always converts the given matrix to an even-sized matrix by padding the matrix before executing Strassen’s algorithm.

Moreover, it is observed that this conversion process (padding) takes only a fraction of the total runtime. Hence, to simplify the analysis, padding is avoided by selecting matrices of sizes where is the cross-over point.

Based on our analytical analysis, the actual optimum cross-over point is anticipated to be in the range of 10-30. However, to better understand the behavior of our implementation, cross-over points at various increments from 4 to 160 are considered as shown below.

Cross-Over Points (): 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 18, 20, 22, 24, 26, 28, 30, 32, 36, 40, 44, 48, 52, 56, 60, 64, 72, 80, 88, 96, 104, 112, 120, 128, 144, 160.

To test these cross-over points, the following 16 matrix dimensions are used.

Matrix Dimensions (): 768, 832, 896, 960, 1024, 1152, 1280, 1408, 1536, 1664, 1792, 1920, 2048, 2304, 2560, 2816.

For each dimension, different cross-over points are considered. An example is presented in Table 2 for

Table 2: Cross-Over Points Considered for n=1536

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

The runtime vs cross-over point graphs are shown in Figure 8 and Figure 9 for and , respectively. In these graphs, the minimum runtime for each matrix size is highlighted with a red dot. The results are also tabulated in the Appendix section (see Table 5) and the optimum cross-over point for each matrix size is given in Table 3.

Table 3: Optimum Cross-Over Points

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **n** | **Optimum Cross-Over Point** |  | **n** | **Optimum Cross-Over Point** |
| 768 | 24 |  | 1536 | 24 |
| 832 | 26 |  | 1664 | 26 |
| 896 | 28 |  | 1792 | 28 |
| 960 | 30 |  | 1920 | 30 |
| 1024 | 16 |  | 2048 | 16 |
| 1152 | 18 |  | 2304 | 18 |
| 1280 | 20 |  | 2560 | 20 |
| 1408 | 22 |  | 2816 | 22 |

Note that the size of the matrices in Figure 8 are doubled in Figure 9. In both graphs we see the same trends. That is, the optimum cross-over point for a matrix of size and are the same. Hence, we would expect to obtain the same optimum cross-over points if we were to double the matrices even further.

In both graphs we observe that, for every matrix size, the minimum cross-over point that is larger than yields the minimum runtime. More specifically, as shown in Table 3, the optimal cross-over point varies in the range of to . Hence, based on the experimental results, we can define the optimal cross-over point as which is close to our analytical cross-over point of (for even-sized matrices).

Also, we observe that for and the runtime difference between the optimum cross-over point and the next smaller cross-over point is negligible. Therefore, the experimental cross-over point may further be decreased by optimizing the memory usage of Strassen’s algorithm.

Chart, box and whisker chart

Description automatically generated

Figure 8: Runtime vs Cross-Over Point (n < 1500)

Chart, bar chart, box and whisker chart

Description automatically generated

Figure 9: Runtime vs Cross-Over Point (n > 1500)

## Triangle in Random Graphs

Random graphs on vertices are generated where in each graph edges are included with probabilities and . Strassen’s algorithm is used to count the number of triangles in each of these graphs, and the results are compared to the expected number of triangles, which is . For each probability, 10 graphs are generated, and the average number of triangles are calculated. The results are given in Table 4.

Table 4: Expected vs Actual Number of Triangles

|  |  |  |
| --- | --- | --- |
| **p** | **Expected Number of Triangles in Graph** | **Avg. Number of Triangles in Graph** |
| 0.01 | 178 | 179 |
| 0.02 | 1427 | 1454 |
| 0.03 | 4818 | 4825 |
| 0.04 | 11420 | 11515 |
| 0.05 | 22304 | 22327 |

We see that the maximum difference between the expected and average number of triangles is less than 2%. The maximum difference may further be reduced by perfoming more trials.

## Appendix

Table 5: Experimental Results

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **n** | **Cross-Over Point** | **Runtime (s)** |  | **n** | **Cross-Over Point** | **Runtime (s)** |
| 768 | 6 | 35 |  | 1536 | 6 | 243 |
| 768 | 12 | 27 |  | 1536 | 12 | 187 |
| 768 | 24 | 25 |  | 1536 | 24 | 176 |
| 768 | 48 | 26 |  | 1536 | 48 | 181 |
| 768 | 96 | 27 |  | 1536 | 96 | 192 |
| 832 | 13 | 33 |  | 1664 | 13 | 229 |
| 832 | 26 | 31 |  | 1664 | 26 | 221 |
| 832 | 52 | 32 |  | 1664 | 52 | 228 |
| 832 | 104 | 35 |  | 1664 | 104 | 244 |
| 896 | 7 | 49 |  | 1792 | 7 | 347 |
| 896 | 14 | 40 |  | 1792 | 14 | 278 |
| 896 | 28 | 39 |  | 1792 | 28 | 273 |
| 896 | 56 | 40 |  | 1792 | 56 | 277 |
| 896 | 112 | 43 |  | 1792 | 112 | 288 |
| 960 | 15 | 47.4 |  | 1920 | 15 | 334 |
| 960 | 30 | 47.1 |  | 1920 | 30 | 331 |
| 960 | 60 | 49 |  | 1920 | 60 | 344 |
| 960 | 120 | 53 |  | 1920 | 120 | 371 |
| 1024 | 4 | 100 |  | 2048 | 4 | 704 |
| 1024 | 8 | 68 |  | 2048 | 8 | 475 |
| 1024 | 16 | 56.0 |  | 2048 | 16 | 393 |
| 1024 | 32 | 56.5 |  | 2048 | 32 | 398 |
| 1024 | 64 | 59 |  | 2048 | 64 | 414 |
| 1024 | 128 | 64 |  | 2048 | 128 | 451 |
| 1152 | 9 | 90 |  | 2304 | 9 | 636 |
| 1152 | 18 | 78 |  | 2304 | 18 | 551 |
| 1152 | 36 | 81 |  | 2304 | 36 | 557 |
| 1152 | 72 | 83 |  | 2304 | 72 | 586 |
| 1152 | 144 | 90 |  | 2304 | 144 | 633 |
| 1280 | 5 | 161 |  | 2560 | 5 | 1139 |
| 1280 | 10 | 117 |  | 2560 | 10 | 822 |
| 1280 | 20 | 106 |  | 2560 | 20 | 728 |
| 1280 | 40 | 107 |  | 2560 | 40 | 754 |
| 1280 | 80 | 113 |  | 2560 | 80 | 795 |
| 1280 | 160 | 123 |  | 2560 | 160 | 872 |
| 1408 | 11 | 141 |  | 2816 | 11 | 1049 |
| 1408 | 22 | 130 |  | 2816 | 22 | 969 |
| 1408 | 44 | 133 |  | 2816 | 44 | 973 |
| 1408 | 88 | 150 |  | 2816 | 88 | 1002 |